UNIVERSIDAD TECNICA FEDERICO SANTA MARIA

Repositorio Digital USM **https://repositorio.usm.cl**

Tesis USM TESIS de Pregrado de acceso ABIERTO

2020-03

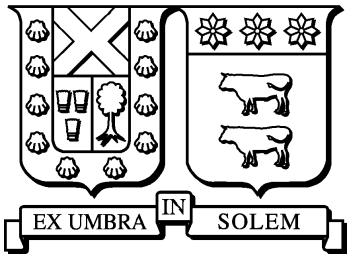
IMPLEMENTACIÓN DE UNA ESTRUCTURA DE DATOS SUCINTA PARA VECTORES DE BITS UTILIZANDO COMPRESIÓN HÍBRIDA S18

CALQUÍN VALDÉS, MANUEL ARMANDO

https://hdl.handle.net/11673/48793

Repositorio Digital USM, UNIVERSIDAD TECNICA FEDERICO SANTA MARIA

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO – CHILE



IMPLEMENTACIÓN DE UNA ESTRUCTURA DE DATOS SUCINTA PARA VECTORES DE BITS UTILIZANDO COMPRESIÓN HÍBRIDA S18

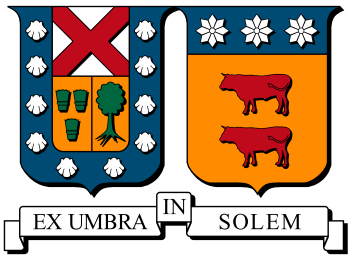
MANUEL ARMANDO CALQUÍN VALDÉS

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL INFORMÁTICO

PROFESOR GUÍA: DIEGO GASTÓN ARROYUELO BILLIARDI

MARZO 2020

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO – CHILE



IMPLEMENTACIÓN DE UNA ESTRUCTURA DE DATOS SUCINTA PARA VECTORES DE BITS UTILIZANDO COMPRESIÓN HÍBRIDA S18

MANUEL ARMANDO CALQUÍN VALDÉS

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL INFORMÁTICO

PROFESOR GUÍA: DIEGO GASTÓN ARROYUELO BILLIARDI
PROFESOR CORREFERENTE: JOSÉ LUIS MARTÍ LARA

MARZO 2020

MATERIAL DE REFERENCIA, SU USO NO INVOLUCRA RESPONSABILIDAD DEL AUTOR O LA INSTITUCIÓN

Agradecimientos

A mi familia, espacialmente a mis padres, Delia y Renato por entregarme una buena educación y por siempre apoyarnos en estudiar lo que quisiéramos. A mis hermanos(as) por haber acompañado durante todo este tiempo.

También agradezo a Fabián, por haberme apoyado mientras escribía este documento estresado en las primeras iteraciones, cuando aún no lograba buenos resultados.

A mis abuelos por haberse preocupado que en los momentos difíciles pudiera seguir estudiando.

A todos los profesores que se preocupaban de que aprendiera, y no sólo de impartir los ramos para ganar su sueldo. Ustedes saben quienes son.

A mis amigos y compañeros de la universidad, quienes se dieron el trabajo de explicarme las cosas que no entendía y con quienes pude contar durante todo el proceso.

Muchas gracias a todos.

Para Fabián.

Sin tu apoyo no habría terminado.

Te quiero mucho.

Resumen

Los avances en la tecnología le han permitido al ser humano generar en la última década más datos de los que había generado en toda la historia. Procesar estos datos se ha vuelto un reto incluso en *hardware* moderno.

Por suerte, las estructuras de datos sucintas permiten procesar estos datos y proveen de las mismas operaciones que sus contrapartes clásicas pero usando menos espacio.

En este documento se estudia una nueva estructura de datos sucinta para vectores de bits. Se bosquejan las ideas claves y se presenta una implementación concreta de la misma. Finalmente se muestra que ésta es competitiva en la teoría y en la práctica, contra estructuras descritas en el estado del arte.

Abstract

The advances in technology have allowed humankind to generate, in just the last decade, more data than the generated in all history until then. Processing this huge datasets has become a challenge even in modern hardware.

Fortunately, succinct data structures allow processing data using a compact representation while providing the same operations as their classic counterparts.

In this document a new succinct data structure for bit vectors is studied. The key ideas are outlined, and a concrete implementation is presented. Finally, it is shown that this structure is competitive in theory and practice against other state-of-the-art succinct structures.

Tabla de Contenidos

Agradecimientos v

Resumen ix

Abstract xi

Tabla de Contenidos xiii

Lista de Tablas xvii

Lista de Figuras xxi

Introducción 1

1. Marco Conceptual 3

1.1. El Problema de Recuperación de Documentos en Motores de Búsqueda de Texto 4

1.1.1. Índices Invertidos 4

1.1.2. El Vector de Bits 5

1.1.3. Vectores de Bits Sucintos 6

1.1.4. Conjuntos Sucintos 7

1.2. El Vector de Bits como Estructura Subyacente 8

1.3. Métricas de Rendimiento para Vectores de Bits 9

1.3.1. Métricas de Tiempo 9

1.3.2. Métricas de Espacio 9

1.4. La *Succinct Data Structure Library* 12

1.5. Objetivos de la Memoria 12

1.5.1. Objetivo General 12

1.5.2. Objetivos Específicos 13

2. Estado del Arte **15**

2.1. Métodos basados en *Gap Encoding* 15

2.1.1. Vector V-Byte 16

2.1.2. Vector RRR 16

2.1.3. Vector SD 17

2.1.4. Vector PForDelta 18

2.2. Métodos Híbridos 19

2.2.1. Vector WAH 19

2.2.2. Vector Hyb 20

2.2.3. Vector Simple 9 21

2.2.4. Vector HPForDelta 21

3. Propuesta **23**

3.1. Estructura General del Método 23

3.2. Compresión *Greedy* de una Pasada 24

3.2.1. Palabras incompletas 25

3.3. Unión de los casos C16 y C18 25

3.4. Índice de bloques de palabras 26

3.5. Un segundo nivel de indirección 27

3.6. Saltos rápidos en *runs* 27

3.7. Resumen 28

4. Implementación **31**

4.1. Metodología 31

4.1.1. Hardware de Pruebas 33

4.1.2. Software de Pruebas 33

4.2. Resultados y observaciones destacables 33

4.2.1. Respecto de la cota inferior teórica 34

4.2.2. Respecto del vector de bits 34

4.2.3. Respecto de `s9_vector` 34

4.2.4. Respecto de `hyb_vector` 35

4.2.5. Respecto de `rrr_vector` 36

4.2.6. Respecto de `sd_vector` 36

Conclusiones **39**

Referencias **43**

Índice de Términos **45**

A. Código Fuente de S18 **45**

B. Datos de Pruebas de Rendimiento **59**

B.1. Access 60

B.2. Rank 90

B.3. Select 120

B.4. Successor 150

Lista de Tablas

B.1. Rendimiento de Access para $P = 0.01$ 61

B.2. Rendimiento de Access para $P = 0.02$ 63

B.3. Rendimiento de Access para $P = 0.03$ 65

B.4. Rendimiento de Access para $P = 0.04$ 67

B.5. Rendimiento de Access para $P = 0.05$ 69

B.6. Rendimiento de Access para $P = 0.1$ 71

B.7. Rendimiento de Access para $P = 0.2$ 73

B.8. Rendimiento de Access para $P = 0.3$ 75

B.9. Rendimiento de Access para $P = 0.4$ 77

B.10. Rendimiento de Access para $P = 0.5$ 79

B.11. Rendimiento de Access para $P = 0.6$ 81

B.12. Rendimiento de Access para $P = 0.7$ 83

B.13. Rendimiento de Access para $P = 0.8$ 85

B.14. Rendimiento de Access para $P = 0.9$ 87

B.15. Rendimiento de Access para $P = 0.95$ 89

B.16. Rendimiento de Rank para $P = 0.01$ 91

B.17. Rendimiento de RANK para $P = 0.02$ 93

B.18. Rendimiento de RANK para $P = 0.03$ 95

B.19. Rendimiento de RANK para $P = 0.04$ 97

B.20. Rendimiento de RANK para $P = 0.05$ 99

B.21. Rendimiento de RANK para $P = 0.1$ 101

B.22. Rendimiento de RANK para $P = 0.2$ 103

B.23. Rendimiento de RANK para $P = 0.3$ 105

B.24. Rendimiento de RANK para $P = 0.4$ 107

B.25. Rendimiento de RANK para $P = 0.5$ 109

B.26. Rendimiento de RANK para $P = 0.6$ 111

B.27. Rendimiento de RANK para $P = 0.7$ 113

B.28. Rendimiento de RANK para $P = 0.8$ 115

B.29. Rendimiento de RANK para $P = 0.9$ 117

B.30. Rendimiento de RANK para $P = 0.95$ 119

B.31. Rendimiento de SELECT para $P = 0.01$ 121

B.32. Rendimiento de SELECT para $P = 0.02$ 123

B.33. Rendimiento de SELECT para $P = 0.03$ 125

B.34. Rendimiento de SELECT para $P = 0.04$ 127

B.35. Rendimiento de SELECT para $P = 0.05$ 129

B.36. Rendimiento de SELECT para $P = 0.1$ 131

B.37. Rendimiento de SELECT para $P = 0.2$ 133

B.38. Rendimiento de SELECT para $P = 0.3$ 135

B.39. Rendimiento de SELECT para $P = 0.4$ 137

B.40. Rendimiento de SELECT para $P = 0.5$ 139

B.41. Rendimiento de SELECT para $P = 0.6$ 141

B.42. Rendimiento de SELECT para $P = 0.7$ 143

B.43. Rendimiento de SELECT para $P = 0.8$ 145

B.44. Rendimiento de SELECT para $P = 0.9$ 147

B.45. Rendimiento de SELECT para $P = 0.95$ 149

B.46. Rendimiento de SUCCESSOR para $P = 0.01$ 151

B.47. Rendimiento de SUCCESSOR para $P = 0.02$ 153

B.48. Rendimiento de SUCCESSOR para $P = 0.03$ 155

B.49. Rendimiento de SUCCESSOR para $P = 0.04$ 157

B.50. Rendimiento de SUCCESSOR para $P = 0.05$ 159

B.51. Rendimiento de SUCCESSOR para $P = 0.1$ 161

B.52. Rendimiento de SUCCESSOR para $P = 0.2$ 163

B.53. Rendimiento de SUCCESSOR para $P = 0.3$ 165

B.54. Rendimiento de SUCCESSOR para $P = 0.4$ 167

B.55. Rendimiento de SUCCESSOR para $P = 0.5$ 169

B.56. Rendimiento de SUCCESSOR para $P = 0.6$ 171

B.57. Rendimiento de SUCCESSOR para $P = 0.7$ 173

B.58. Rendimiento de SUCCESSOR para $P = 0.8$ 175

B.59. Rendimiento de SUCCESSOR para $P = 0.9$ 177

B.60. Rendimiento de SUCCESSOR para $P = 0.95$ 179

Lista de Figuras

1.1. Índice invertido 4

1.2. Índice invertido con bitmaps 6

1.3. Representación de un Wavelet Tree 8

2.1. Representación gráfica de bloques de bits RRR. 17

2.2. Representación gráfica de un vector comprimido con RRR. 17

2.3. Representación gráfica de un bloque comprimido con PForDelta. . . . 19

2.4. Representación gráfica de un vector comprimido con WAH. 20

2.5. Representación gráfica de un vector comprimido con S9. 21

3.1. Representación gráfica de la implementación S18 29

4.1. Rendimiento de Rank para Hyb y S18 en vectores densos 35

4.2. Rendimiento de operaciones SD y S18 para vector con densidad 1 % . 37

4.3. Rendimiento de operaciones SD y S18 para vector con densidad $3\,\%$. 38

B.1. Rendimiento de Access para $P = 0.01$ 60

B.2. Rendimiento de Access para $P = 0.02$ 62

B.3. Rendimiento de Access para $P = 0.03$ 64

B.4. Rendimiento de Access para $P = 0.04$ 66

B.5. Rendimiento de Access para $P = 0.05$ 68

B.6. Rendimiento de Access para $P = 0.1$ 70

B.7. Rendimiento de Access para $P = 0.2$ 72

B.8. Rendimiento de Access para $P = 0.3$ 74

B.9. Rendimiento de Access para $P = 0.4$ 76

B.10. Rendimiento de Access para $P = 0.5$ 78

B.11. Rendimiento de Access para $P = 0.6$ 80

B.12. Rendimiento de Access para $P = 0.7$ 82

B.13. Rendimiento de Access para $P = 0.8$ 84

B.14. Rendimiento de Access para $P = 0.9$ 86

B.15. Rendimiento de Access para $P = 0.95$ 88

B.16. Rendimiento de Rank para $P = 0.01$ 90

B.17. Rendimiento de Rank para $P = 0.02$ 92

B.18. Rendimiento de Rank para $P = 0.03$ 94

B.19. Rendimiento de Rank para $P = 0.04$ 96

B.20. Rendimiento de Rank para $P = 0.05$ 98

B.21. Rendimiento de Rank para $P = 0.1$ 100

B.22. Rendimiento de Rank para $P = 0.2$ 102

B.23. Rendimiento de Rank para $P = 0.3$ 104

B.24. Rendimiento de Rank para $P = 0.4$ 106

B.25. Rendimiento de Rank para $P = 0.5$ 108

B.26. Rendimiento de RANK para $P = 0.6$ 110

B.27. Rendimiento de RANK para $P = 0.7$ 112

B.28. Rendimiento de RANK para $P = 0.8$ 114

B.29. Rendimiento de RANK para $P = 0.9$ 116

B.30. Rendimiento de RANK para $P = 0.95$ 118

B.31. Rendimiento de SELECT para $P = 0.01$ 120

B.32. Rendimiento de SELECT para $P = 0.02$ 122

B.33. Rendimiento de SELECT para $P = 0.03$ 124

B.34. Rendimiento de SELECT para $P = 0.04$ 126

B.35. Rendimiento de SELECT para $P = 0.05$ 128

B.36. Rendimiento de SELECT para $P = 0.1$ 130

B.37. Rendimiento de SELECT para $P = 0.2$ 132

B.38. Rendimiento de SELECT para $P = 0.3$ 134

B.39. Rendimiento de SELECT para $P = 0.4$ 136

B.40. Rendimiento de SELECT para $P = 0.5$ 138

B.41. Rendimiento de SELECT para $P = 0.6$ 140

B.42. Rendimiento de SELECT para $P = 0.7$ 142

B.43. Rendimiento de SELECT para $P = 0.8$ 144

B.44. Rendimiento de SELECT para $P = 0.9$ 146

B.45. Rendimiento de SELECT para $P = 0.95$ 148

B.46. Rendimiento de SUCCESSOR para $P = 0.01$ 150

B.47. Rendimiento de SUCCESSOR para $P = 0.02$ 152

B.48. Rendimiento de Successor para $P = 0.03$ 154

B.49. Rendimiento de Successor para $P = 0.04$ 156

B.50. Rendimiento de Successor para $P = 0.05$ 158

B.51. Rendimiento de Successor para $P = 0.1$ 160

B.52. Rendimiento de Successor para $P = 0.2$ 162

B.53. Rendimiento de Successor para $P = 0.3$ 164

B.54. Rendimiento de Successor para $P = 0.4$ 166

B.55. Rendimiento de Successor para $P = 0.5$ 168

B.56. Rendimiento de Successor para $P = 0.6$ 170

B.57. Rendimiento de Successor para $P = 0.7$ 172

B.58. Rendimiento de Successor para $P = 0.8$ 174

B.59. Rendimiento de Successor para $P = 0.9$ 176

B.60. Rendimiento de Successor para $P = 0.95$ 178

Introducción

El actual crecimiento y disponibilidad de cantidades masivas de datos creada y recuperada por aplicaciones ha cambiado los requerimientos algorítmicos de las tareas de procesamiento y minería de datos. Esto ha generado una explosión en la investigación de las Estrucuras de Datos Sucintas (SDS, por sus siglas en inglés) durante la última década. La implementación de SDS eficientes requiere una mezcla de ideas del área algorítmica así como de compresión.

Las SDS son aquellas que ocupan una cantidad de memoria asintóticamente cercana a la cota inferior teórica y aún así permiten realizar operaciones en un tiempo eficiente. El concepto fue acuñado 3 décadas atrás [Jacobson, 1988] y desde entonces ha dado pie al estudio de los métodos de compresión subyacentes a dichas estructuras, así como a métodos que permiten realizar operaciones eficientes sobre las mismas.

Destaca el estudio de las SDS de tipo *conjunto* o *diccionario*, donde el problema es fundamentalmente representar un subconjunto $S \subseteq U$ de un universo de enteros

$$U = \{0,\, 1,\, 2,\, 3,\, \ldots,\, u-1\}$$

el cual puede ser representado como un vector de bits $B\,[1 \ldots u)$, donde

$$B[i] = \begin{cases} 1 & i \in S \\ 0 & i \notin S \end{cases}$$

El estudio y desarrollo de mejores SDS le permite a las aplicaciones que hacen uso de ellas poder almacenar en memoria principal representaciones de datos cuyo tamaño sería prohibitivo si se usaran las contrapartes clásicas de dichas estructuras. Es decir, el aporte que se pueda realizar en el campo de las SDS se ramifica a otras áreas de las

tecnologías de la información, como puede ser la rama de Recuperación de Información (IR, por sus siglas en inglés). Asimismo, el espacio adicional que se consigue como resultado de la compresión implícita puede ser usado para almacenar estructuras auxiliares que soportan operaciones menos triviales sobre el conjunto de datos.

En el presente documento se estudiarán los métodos de compresión de vectores de bits para SDS, con el fin de proponer una nueva implementación para conjuntos, basada en el modelo de compresión *Simple-18* (S18). S18 es un modelo de compresión híbrido basado en *Gap Encoding* (GE) y *Run-Length Encoding* (RLE), que trata de capturar lo mejor de ambos modelos y que ha mostrado tener buenos resultados en cuanto a tasa de compresión así como en velocidad de procesamiento de consultas.

La implementación será realizada como una extensión de la *Succinct Data Structure Library* (SDSL), la cual provee la implementación de SDS en distintos niveles de abstracción, considerando puntos claves mencionados en más de 40 publicaciones.

Asimismo, se evaluará la competitividad (en cuanto a uso de memoria como eficiencia de sus operaciones) de la implementación aplicada a los problemas reales como

- Compresión de conjuntos de enteros.
- Representación de mapas de bits.
- Representar y operar (realizar consultas) sobre índices invertidos.

El documento se estructura de la siguiente forma: En el Capítulo 1 se define el marco conceptual del problema, indicando qué aristas se deben considerar a la hora de evaluar la competitividad de la solución. A continuación, en el Capítulo 2 se realiza un estudio y análisis de las propuestas existentes en la literatura actual para resolver el problema, destacando sus fortalezas y debilidades. Luego, en el Capítulo 3 se presenta la propuesta de una nueva solución al problema. A esto le sigue la validación de la propuesta, en el Capítulo 4, donde se evalúa la competitividad de la misma. Finalmente se presentan las conclusiones del trabajo realizado y se propone trabajo a futuro para continuar con el estudio del problema.

Capítulo 1

Marco Conceptual

El campo de estudio de las SDS ha evolucionado rápidamente en la última década. Una serie de estructuras nuevas han sido propuestas, desarrolladas, y ha sido mostrado que son muy versátiles. Las SDS proveen la misma funcionalidad que sus contrapartes clásicas, pero usando una cantidad de espacio asintóticamente cercana a la cota inferior teórica necesaria para almacenar los datos subyacentes.

Usualmente la complejidad de tiempo de las operaciones de las SDS es idéntica o cercana a la de su implementación clásica. Sin embargo, en la práctica las SDS tienden a ser más lentas que sus contrapartes, debido a que los patrones de acceso a la información subyacente son más complejos. Esto quiere decir que el uso de SDS es deseable sólo en contextos donde la representación de la implementación clásica tiene un uso de memoria prohibitivo.

También existen otros escenarios donde el uso de SDS es deseable. Por ejemplo, para las tecnologías en la nube. El poder de procesamiento se puede comprar bajo demanda. El espacio de almacenamiento se puede comprar por tamaño y es cobrado por tiempo de uso. Es decir, existen casos en los cuales es más rentable contar con una representación compacta de los datos y ahorrar costos en espacio de almacenamiento. A cambio se debe pagar por más poder de procesamiento, pero sólo cuando es requerido.

1.1. El Problema de Recuperación de Documentos en Motores de Búsqueda de Texto

Una de las aplicaciones de las estructuras sucintas es en el problema de recuperación de información, particularmente en motores de búsqueda de texto, donde el problema es fundamentalmente encontrar un subconjunto de documentos dentro de una colección, tal que contengan una o una serie de palabras de un vocabulario. Notar que el problema de recuperación de documentos no se limita sólo a documentos y texto, sino que puede aplicarse a cualquier entidad que posee un conjunto de atributos asociados, los cuales pueden ser compartidos.

1.1.1. Índices Invertidos

La estructura de datos clásica que permite resolver eficientemente el problema de recuperación de documentos en los motores de búsqueda de texto es el índice invertido [Zobel *et al.*, 1998]. La idea fundamental de dichos índices es asociar una lista a cada palabra del vocabulario, la cual contenga los identificadores de todos los documentos que la contienen. Un ejemplo de índice invertido es el propuesto en la Figura 1.1.

Doc 1
Me gusta el pan con palta.

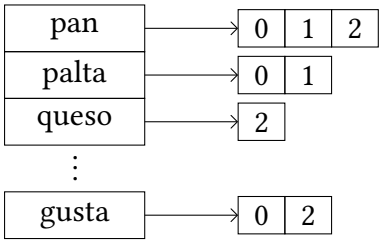
Doc 2
A Manuel le encanta el pan con palta.

Doc 3
A Fabián le gusta el pan con queso.

(a) Documentos a ser indizados usando un índice invertido.

Documento	Identificador
Doc 1	0
Doc 2	1
Doc 3	2

(b) Tabla de asociación entre documentos de (a) y sus identificadores.



(c) Índice invertido para los documentos de (a) usando los identificadores definidos en (b).

Figura 1.1: Representación gráfica de un índice invertido.

Con estas estructuras, encontrar la serie de documentos que contienen una única palabra p_i es fácil, dado que ya se tiene una lista L_j asociada que contiene los identificadores d_k de cada documento tal que $p_i \in d_k$.

Asimismo, si se quisiera encontrar los documentos que contienen una serie de palabras p_1, p_2, p_3, $\ldots$, p_n sólo se debe realizar una intersección entre las listas asociadas a cada palabra ($L_1 \cap L_2 \cap \ldots \cap L_n$). Análogo es el problema de encontrar documentos que contienen una o más de una serie de palabras, en cuyo caso se debe unir dichas listas ($L_1 \cup L_2 \cup \ldots \cup L_n$). Se debe tener en consideración que la estructura que se ocupe para representar dichas listas define la eficiencia de las operaciones de unión e intersección.

Dentro de las estructuras que destacan para representar las listas de asociación a documentos están los vectores de enteros, *hashing*, listas enlazadas, entre otras, siendo la más relevante para este documento (por su facilidad de uso y alta compresibilidad) el vector de bits[1].

1.1.2. El Vector de Bits

El vector de bits es una estructura de datos que almacena los valores 0 o 1 de forma compacta (i.e. almacena cada valor usando un único bit).

Esta estructura permite mapear desde un dominio de alta cardinalidad (e.g. un conjunto de enteros) a otro que consiste sólo en los valores 0 y 1. Esto es útil para representar conjuntos. Por ejemplo, un conjunto $S \subseteq U$ tal que

$$U = \{0,\ 1,\ 2,\ 3,\ \ldots,\ u-1\} \tag{1.1}$$

puede ser mapeado a un vector de bits $B[0 \ldots u)$ definiendo

$$B\,[i] = \begin{cases} 1 & i \in S \\ 0 & i \notin S \end{cases} \tag{1.2}$$

En la Figura 1.2 se puede observar que cada vector de bits debe tener un largo igual a la cantidad de documentos. Es decir, para un motor que almacena un conjunto D de

[1]Conocido también como mapa de bits, conjunto de bits o arreglo de bits.

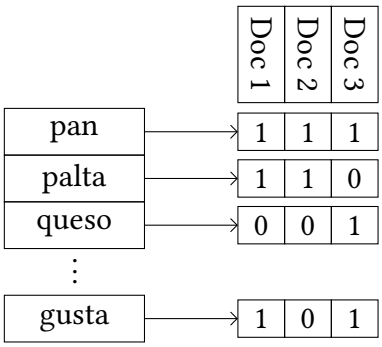


Figura 1.2: Índice invertido para los documentos
presentados en la Figura 1.1 usando vectores de bits.

documentos, e indiza un vocabulario (conjunto) V de palabras, se tiene que los vectores de bits del índice invertido hacen uso de $|D| \times |V|$ bits.

En la práctica los tamaños de los vocabularios de los índices invertidos suelen ser (relativamente) estáticos, a diferencia del conjunto de documentos que suele estar en constante crecimiento. Cada documento que se agrega al índice tiene un costo de $|V|$ bits. El diccionario *Oxford* tiene aproximadamente $172{,}000$ palabras, lo que significa que para un motor de búsqueda basado en el vocabulario anglosajón, agregar un documento tiene un costo de $172{,}000[\text{b}] \approx 21.5[\text{KB}]$. Esto quiere decir que almacenar índices invertidos es costoso en cuanto al espacio requerido, ¡Para un millón de documentos sólo el índice del motor pesaría aproximadamente $2.2[\text{GB}]$!.

1.1.3. Vectores de Bits Sucintos

Dado el rápido crecimiento que pueden tener los índices invertidos en cuanto a uso de memoria es deseable contar con una implementación sucinta del vector de bits. Existe una serie de modelos de compresión para vectores de bits. Los aspectos más importantes a ser considerados son

- La capacidad de compresión del modelo usado.
- La posibilidad de uso (i.e. realizar operaciones y/o consultas) de los datos en su estado compacto. Por ejemplo, existen modelos como WAH [Wu *et al.*, 2002] que permiten realizar operaciones *bitwise* sin la necesidad de descomprimir los datos.

- El *tradeoff* entre el uso de espacio y la eficiencia de las operaciones sobre la estructura sucinta.

Por otro lado, es posible modelar cada una de las listas L_i del índice invertido como un conjunto que contiene los documentos asociadas a cada palabra. Es decir, las operaciones relevantes para resolver el problema son aquellas asociadas a las estructuras de tipo *conjunto*.

1.1.4. Conjuntos Sucintos

Los conjuntos sucintos son las contrapartes compactas de los conjuntos. Estas estructuras permiten representar un subconjunto $S \subseteq U$ de un universo U (1.1). Además deben contar con las operaciones fundamentales sobre las cuales se pueden construir otras más complejas, que permiten tener a disposición la implementación subyacente en la cual se apoyan otras estructuras más abstractas, como por ejemplo un índice invertido. Estas operaciones son:

- INSERT: Permite insertar un nuevo elemento en S.
- DELETE: Permite eliminar un elemento de S.
- ACCESS: Permite acceder al i-ésimo elemento en U, de lo cual se puede deducir la pertenencia en S.
- RANK_q: Permite conocer la cantidad de elementos en S que son iguales al indicado (q), considerando sólo hasta una cierta posición.
- SELECT_q: Permite encontrar la i-ésima ocurrencia de un elemento q en S.

Cuando los datos almacenados en el conjunto S son invariantes, sólo importan las últimas tres operaciones. Asimismo, cuando los datos almacenados dentro de el conjunto son de baja cardinalidad, como lo es para el vector de bits, sólo son necesarias las operaciones RANK_1 y SELECT_1 con las cuales se pueden construir todas las operaciones de los conjuntos clásicos (i.e. pertenencia, acceso, unión, intersección).

1.2. El Vector de Bits como Estructura Subyacente

Basta contar con vectores de bits sucintos en un espacio de baja cardinalidad para ser capaces de representar conjuntos de alta cardinalidad. Por ejemplo, usando conjuntos sucintos basados e vectores de bits se pueden generalizar las operaciones $\textsc{Rank}_q$ y $\textsc{Select}_q$

Una estructura sucinta que hace justamente esto es el *Wavelet Tree* [Grossi *et al.*, 2003] y está implementado en base a vectores de bits sucintos. La motivación por realizar esta generalización escapa de los contenidos a repasar en este documento, pero es un buen ejemplo para mostrar la relevancia de contar con buenas implementaciones de vectores de bits sucintos, dado que pueden ser usados para armar estructuras más complejas (además el *Wavelet Tree*, permite implementar la contraparte sucinta del Arreglo de Sufijos).

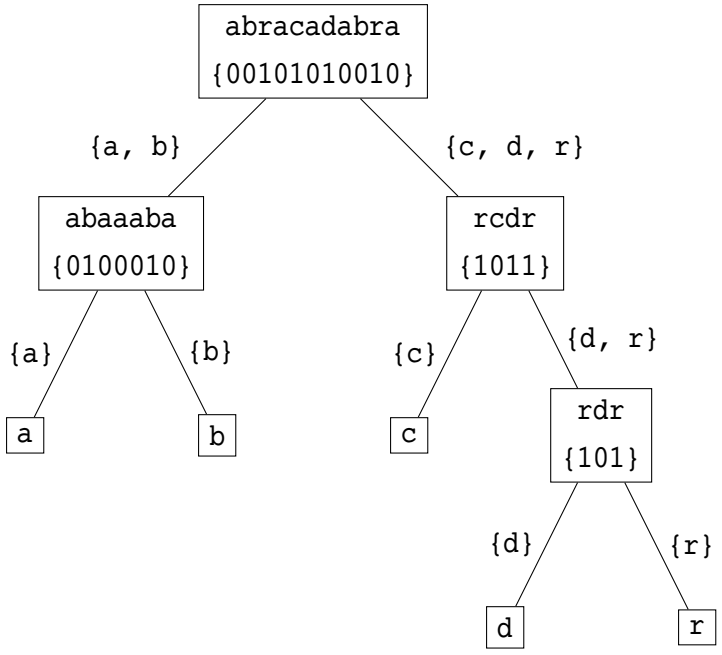


Figura 1.3: Representación gráfica de un *Wavelet Tree*. Cada nodo particiona la cadena de símbolos y un vector de bits representa en qué partición se proyecta cada uno. La implementación sucinta usa vectores de bits sucintos.

1.3. Métricas de Rendimiento para Vectores de Bits

Dado que existen diversos modelos para codificar vectores de bits de forma sucinta, se deben definir métricas para determinar las propiedades de cada implementación.

Como características relevantes se definen métricas para la velocidad de las operaciones, así como el uso de espacio de los datos en su estado sucinto.

1.3.1. Métricas de Tiempo

Operaciones por unidad de tiempo

Como se define en la Sección 1.2, el vector de bits suele ser la implementación subyacente bajo otra estructura de dato. Lo relevante para un usuario es qué tan rápidas son las operaciones de dicha estructura. El usuario final suele estar interesado en la latencia de las operaciones sobre la estructura, es decir, una medida adecuada es el tiempo promedio de respuesta[2]:

$$L = \frac{\text{Tiempo transcurrido}[ns]}{\text{Consultas realizadas}[u]} \qquad (\textit{Latencia})$$

Si bien además se suele hacer la distinción entre tiempo de compresión, descompresión y consulta, en el caso de las SDS no tiene sentido. Esto porque las consultas se hacen sobre el espacio comprimido, por lo tanto son la misma métrica aplicada a cada tipo de consulta.

1.3.2. Métricas de Espacio

Ratio de compresión (Bits por bit)

Una métrica particularmente relevante es aquella que mide el uso de espacio de los datos en su estado sucinto, dado que la motivación por usar SDS es justamente permitir

[2]Las operaciones sobre vectores de bits suelen tener una duración del orden de magnitud de los nanosegundos en los procesadores modernos (*circa* 2020).

almacenar en memoria la representación de una mayor cantidad de datos. La medida que suele ser usada en la literatura es el ratio de compresión, usualmente llamado como "bits por bit", es decir:

$$R = \frac{\text{Tamaño de datos comprimidos [bit]}}{\text{Tamaño de datos [bit]}} \qquad \text{(Ratio de compresión)}$$

Por ejemplo, si decimos que una estructura logra una tasa de 2[bits por bit], quiere decir que en promedio cada bit en el estado sucinto es capaz de almacenar la información de 2[bit] en la contraparte clásica.

Cota Inferior de Compresión

Es importante considerar que dependiendo del modelo usado para la compresión, se puede lograr un uso de espacio distinto. Cada modelo de compresión $\mathbb{M}$ tiene asociada una cota inferior $\Omega(\mathbb{M})$. En la práctica se busca que el uso de espacio empírico $\mathbb{S}$ sea lo más cercano a la cota inferior:

$$R_\Omega = \frac{\mathbb{S}}{\Omega(\mathbb{M})} \qquad \text{(Cota inferior de compresión)}$$

Por ejemplo, para modelos que usan *Gap Encoding* (GE), su cota inferior es la Entropía Gap

Definición 1.3.1. Si una secuencia $\mathcal{X}$ puede ser codificada en términos de *gaps* $\mathcal{G}(\mathcal{X})$, entonces se requiere

$$\mathbb{G}(\mathcal{X}) = \sum_{g_i \in \mathcal{G}(\mathcal{X})} 1 + \log_2 g_i \qquad \text{(Entropía Gap)}$$

bits para codificarla usando un modelo de compresión basado en GE.

Por otra parte, S18 es un modelo de compresión híbrido localmente adaptativo que detecta regiones que son compresibles con GE y las diferencia de aquellas compresibles con RLE, por lo que la cota inferior está definida por la Entropía Localmente Adaptativa (LAC).

Definición 1.3.2. Si una secuencia $\mathcal{X}$ puede ser codificada en términos de *gaps* $\mathcal{G}(\mathcal{X})$ y *runs* $\mathcal{R}(\mathcal{X})$, entonces se requiere

$$\mathbb{L}(\mathcal{X}) = \left(\sum_{g_i \in \mathcal{G}(\mathcal{X})} 1 + \log_2 g_i \right) + \left(\sum_{r_i \in \mathcal{R}(\mathcal{X})} 1 + \log_2 r_i \right) \qquad \text{(Entropía LAC)}$$

La entropía LAC se obtiene al comprimir de forma perfecta todos los *gaps* como *gaps* y los *runs* como *runs*. Esto considera sólo un bit de espacio adicional que puede ser usado para guardar marcadores indicando si lo comprimido es un *gap* o *run*, empaquetar esa información en la menor cantidad de bits es tarea de quien diseña el método de compresión. Por ejemplo, el vector de bits

$$\boxed{0001000010000111111001000111111111111001011111}$$

es codificado perfectamente como

$$\underbrace{\boxed{0001}}_{g_1}\underbrace{\boxed{00001}}_{g_2}\underbrace{\boxed{00001}}_{g_3}\underbrace{\boxed{11111}}_{r_1}\underbrace{\boxed{001}}_{g_4}\underbrace{\boxed{0001}}_{g_5}\underbrace{\boxed{1111111111}}_{r_2}\underbrace{\boxed{001}}_{g_6}\underbrace{\boxed{01}}_{g_7}\underbrace{\boxed{1111}}_{r_3}$$

y el vector que almacena la información adicional es

$$\boxed{0001001001}$$

indicando con 0 cuando un bloque es un *gap* y con un 1 en caso contrario.

Notar además que un mejor modelo de compresión permite representar los mismos datos en menos espacio.

Teorema 1.3.1. La Entropía LAC está acotada superiormente por la Entropía Gap.

$$\mathbb{L}(\mathcal{X}) \leq \mathbb{G}(\mathcal{X}) \tag{1.3}$$

Demostración. Un vector de bits es codificado con GE, de tal modo que genera *gaps*

$$G = \{g_1,\, g_2,\, g_3,\, \cdots,\, g_n\}$$

en un espacio

$$\sum_{g_i \in G} 1 + \log_2 g_i$$

Si el vector no tiene *runs*, entonces el tamaño LAC es el mismo. En cambio, si posee un único *run* de largo 2, entonces hay 2[bit] de dos *gaps* distintos de largo 1 que se convierten en 2[bit] de un *run* de largo 2 codificado en $1 + \log_2 2$.

Para todos los otros casos, donde existe al menos un *run* de largo $l \geq 2$, entonces existen l *gaps* de largo 1[bit] cada una, que se convierten en un *run* de

$$1 + \log_2 l \leq l \qquad\qquad \forall l > 2$$

bits, donde sigue que una serie de *gaps* son empaquetados en un *run* cuyo código es mas corto. ■

Corolario 1.3.0.1. Los modelos localmente adaptativos tienen un mejor potencial de compresión que los modelos basados en GE.

1.4. La *Succinct Data Structure Library*

Implementar estructuras de datos sucintas es una tarea altamente no-trivial y que requiere de un gran esfuerzo. Por este motivo, Simon Gog implementó la *Succinct Data Structure Library* [Gog *et al.*, 2014] (SDSL) que implementa SDS en distintos niveles de abstracción y que permiten a cualquier usuario usarlas sin conocer nada sobre la implementación subyacente. Asimismo, la librería rescata puntos claves mencionados en más de 40 publicaciones, por lo tanto su desarrollo va de la mano con lo existente en el estado del arte.

La SDS implementa `hyb_vector` [Karkkainen *et al.*, 2014], un vector híbrido, es decir, se aprovecha de las propiedades que tiene localmente el vector de bits y decide de forma automática qué regiones codificar con GE y cuáles con RLE. Dada la naturaleza intrínsecamente híbrida de este algoritmo, es necesario comparar su rendimiento con el de la solución que será propuesta en este documento en adición a los métodos existentes en el estado del arte.

1.5. Objetivos de la Memoria

Teniendo en consideración el marco teórico expuesto en el presente capítulo, se definen los objetivos generales de esta memoria a continuación.

1.5.1. Objetivo General

Obtener un producto de *software* derivado de la SDSL que soporte compresión híbrida S18 para vectores de bits.

1.5.2. Objetivos Específicos

- Diseñar e implementar una Estructura de Datos Sucinta para vectores de bits, basada en la compresión híbrida S18, que soporte las operaciones Access, Rank y Select.
- Evaluar la competitividad de la SDS implementada respecto de otros métodos de compresión para vectores de bits, particularmente enfocado a la compresión de índices de mapas de bits e índices invertidos.
- Comparar el uso de espacio de la SDS implementada respecto de la cota inferior teórica.

Capítulo 2

Estado del Arte

Existe un amplia gama de estructuras sucintas para compresores de bits. El problema de recuperar documentos de una colección en base a consultas sobre el contenido de los mismos es clásico, por lo tanto despierta el interés de los investigadores por encontrar nuevas formas de poder consultar este tipo de colecciones, las cuales sólo se han vuelto más grandes con el paso de los años.

En este capítulo se describirán brevemente algunos de los métodos que cuentan con una implementación práctica disponible para el público, o bien cuyas ideas son relevantes para la implementación a definir en el Capítulo 3.

2.1. Métodos basados en *Gap Encoding*

El modelo clásico para comprimir vectores de bits es el que codifica un vector de *gaps* en un espacio comprimido. Cada *gap* corresponde a la distancia sucesiva entre unos (o ceros) del vector de bits. Es decir, codifican la información respecto de cada elemento existente dentro del conjunto de forma incremental.

2.1.1. Vector V-Byte

Esta implementación propuesta por [Williams y Zobel, 1999] destaca por ser extremadamente simple, pero efectiva para los casos en los cuales los *gaps* tienen un largo cuya representación en binario es en promedio del mismo tamaño que un byte.

V-Byte codifica cada *gap* en una palabra de 1[byte], de los cuales utiliza los 7[bit] inferiores para codificar el *gap* en sí, y reserva el bit más significativo para marcar una palabra como continuación de la anterior en caso de que su representación no quepa en 7[bit], en cuyo caso los mismos son repartidos en más de una palabra.

Por supuesto, la mayor desventaja de este método es que desperdicia mucho espacio de almacenamiento cuando la mayoría de los *gaps* son pequeños (i.e. pueden ser codificados en menos de 7[bit]).

2.1.2. Vector RRR

RRR, por las siglas de sus autores [Raman *et al.*, 2002]. La idea fundamental de RRR es dividir el vector de bits en bloques de tamaño homogéneo, por ejemplo b. A cada bloque se le asigna una clase C correspondiente a la cantidad de bits iguales a 1. Si se tiene el conocimiento de cuantos bits son 1 dentro de un bloque, entonces se conoce de forma implícita todos los valores que puede tener dicho bloque. RRR guarda junto con la clase el *offset* indicando la posición que tendría el bloque si se tuviera una tabla con todos los valores posibles ordenados de forma ascendiente.

Por lo tanto, divide el vector en bloques de b[bit] cada uno, y a cada bloque asigna una clase y el *offset* a la tabla implícita de todas las posibilidades.

Esto permite iterar rápidamente sobre los bloques para llevar la suma acumulativa de bits iguales a 1 (para calcular Rank). Además, para poder responder a dichas consultas en tiempo constante, RRR agrupa los bloques en superbloques de tamaño constante f, y para cada uno almacena en un índice adicional el valor de Rank hasta dicho bloque.

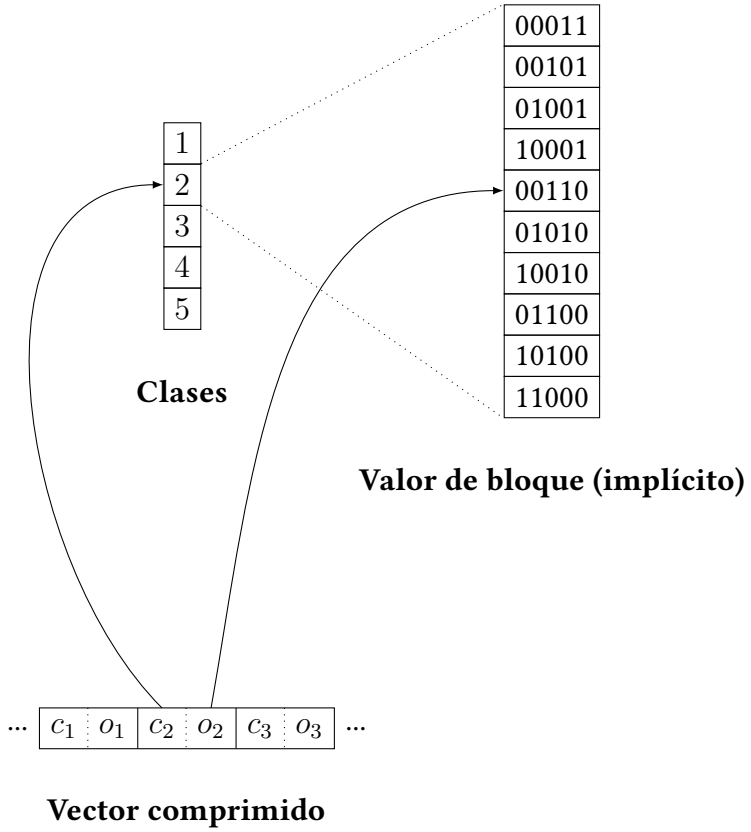


Figura 2.1: Representación gráfica de bloques de bits RRR.

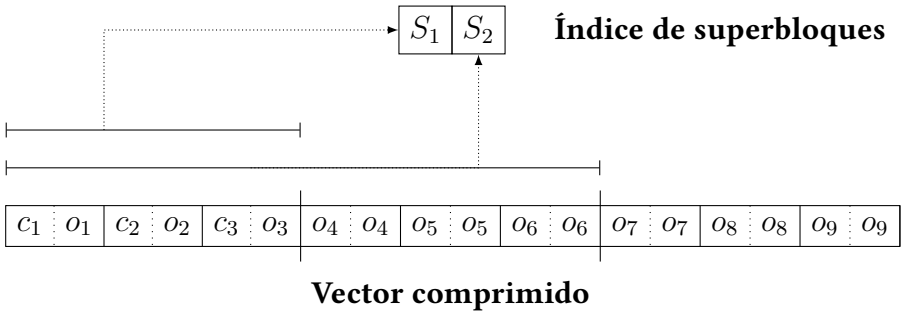


Figura 2.2: Representación gráfica de un vector comprimido con RRR.

2.1.3. Vector SD

El vector SD (*sparse–dense-vector*) es propuesto en [Okanohara y Sadakane, 2007] es capaz de comprimir el vector de bits representando los *gaps* entre bits fijados utilizando codificación Elias–Fano [Elias, 1974, Fano, 1971].

En resumen, el método codifica las posiciones del bit más común de forma sucinta. Por este motivo define dos estrategias de codificación, dependiendo de la densidad de unos.

`sarray` para conjuntos dispersos

El método consiste en codificar todas las posiciones de los bits fijados (en total son m) en dos arreglos H, L. El primero guarda implícitamente $\lfloor \log_2 m \rfloor$ bits superiores, marcando con un 1 si existe alguna posición tal que contiene esos bits superiores (*hi*), mientras que el segundo guarda los bits restantes (*lo*).

Para obtener SELECT de la x-ésima posición, se debe buscar si el bit correspondiente a los bits superiores de x se encuentra fijado en H y de estarlo, se debe sumar el `offset` en L a dicho valor.

`darray` para conjuntos densos

En este caso, se particiona H tal que cada bloque contiene la misma cantidad de bits fijados. Luego en L la estrategia es guardar el *offset* desde el comienzo del bloque, o bien, si no es conveniente, guardar directamente la posición absoluta de los bits fijados, lo cual se decide localmente bloque a bloque.

Obtener SELECT del x-ésimo bit es análogo a `sarray`, solo que en algunos casos, no es necesario decodificar dado que el valor se encuentra guardado sin ningún tipo de compresión.

2.1.4. Vector PForDelta

Este método propuesto por [Zukowski *et al.*, 2006] comparte algunas características con VByte, sin embargo define bloques de *gaps* (por ejemplo, 128) y los codifica en la menor cantidad de bits posible. Digamos que el máximo *gap* de un bloque es x, entonces PForDelta codifica todos los *gaps* del bloque en $b = \lceil \log_2 x \rceil$. Para esto debe almacenar una cabecera donde indica el tamaño de cada bloque.

Asimismo, el método asume que la mayoría de los *gaps* son pequeños, por lo que para

calcular b descarta el 10 % de los *gaps* más grandes. En las posiciones donde debiesen estar las excepciones, PForDelta almacena el `offset` a la siguiente excepción. Los valores de las excepciones son guardadas al final del bloque y no son comprimidos.

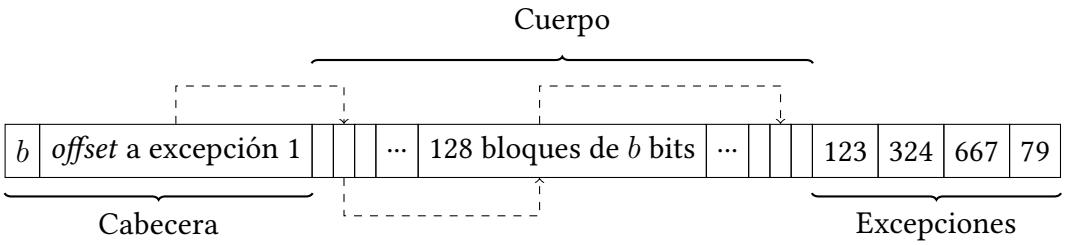


Figura 2.3: Representación gráfica de un bloque comprimido con PForDelta.

2.2. Métodos Híbridos

La compresión de los métodos basados en *gap encoding* está acotada por la Entropía *Gap*, es decir, en presencia de *runs* de unos, deben codificar cada uno como si fuese un *gap*.

Los métodos híbridos sobrepasan este problema agregando información adicional a la codificación, de modo que pueden adaptarse y codificar regiones de *gaps* utilizando GE, mientras que las regiones con *runs* son comprimidas utilizando RLE. Estos métodos tienen una tasa de compresión acotada por la entropía LAC.

2.2.1. Vector WAH

Este método es propuesto en [Wu *et al.*, 2006] y destaca por su simpleza y efectividad. El método particiona el vector de bits en bloques de 31[bit] cada uno. Posteriormente codifica en 32[bit]:

- El bloque literal, en caso de tener una mezcla de 0 y 1, con una cabecera adicional de 1 bit igual a 0
- El largo del *run* de bloques: En caso de que un bloque tenga sólo 0 o sólo 1, se codifica el *run* (en cantidad de bloques) que comparten el mismo patrón. Para

indicar si el patrón es 0 o 1 se fija el segundo bit de la palabra WAH con el valor correspondiente.

Vector de bits separado en bloques

00000001000000000100000000000000
00000000000000000000000000000000
00000000000000000000000000000000
11000000100000000100000010000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
11010000001000011000100000000000

Vector de bits comprimido

000000001000000000100000000000000
100000000000000000000000000000010
011000000100000000100000010000000
100000000000000000000000000000011
011010000001000011000100000000000

Figura 2.4: Representación gráfica de un vector comprimido con WAH.

2.2.2. Vector Hyb

Este método propuesto en [Karkkainen *et al.*, 2014] divide el vector de bits en $\frac{n}{b}$ bloques de tamaño b. Cada bloque es luego codificado usando una cabecera de tamaño fijo a la cual le sigue un cuerpo de tamaño variable. La cabecera define si el cuerpo es uno de tres posibles casos

- Un vector plano de b bits,
- Las posiciones de los bits minoritarios,
- La longitud de *runs* de 0 y 1 de forma alternada, usando a lo mas $\lceil \log_2 b \rceil$[bit] cada uno

2.2.3. Vector Simple 9

Propuesto por [Anh y Moffat, 2005], S9 codifica los *gaps* del vector de bits en palabras de 32 bits. Reserva los 4 bits superiores para guardar información sobre cuál es el tamaño en bits de los *gaps* codificados en los 28 bits restantes de la palabra.

Los casos definidos son

C1 1 *gap* de 28 bits, con la cabecera `0000`

C2 2 *gaps* de 14 bits cada uno, con la cabecera `0001`

C3 3 *gaps* de 9 bits cada uno, con la cabecera `0010`

C4 4 *gaps* de 7 bits cada uno, con la cabecera `0011`

C5 5 *gaps* de 5 bits cada uno, con la cabecera `0100`

C6 7 *gaps* de 4 bits cada uno, con la cabecera `0101`

C7 9 *gaps* de 3 bits cada uno, con la cabecera `0110`

C8 14 *gaps* de 2 bits cada uno, con la cabecera `0111`

C9 28 *gaps* de 1 bit cada uno, con la cabecera `1000`

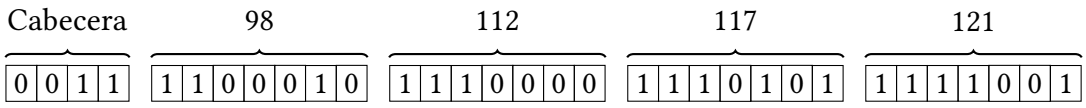


Figura 2.5: Representación gráfica de un vector comprimido con S9.

2.2.4. Vector HPForDelta

Este método propuesto en [Arroyuelo *et al.*, 2018] construye un un esquema basado en PForDelta, aprovechándose que la versión original deja un bit libre en la cabecera. Este bit es utilizado para indicar si el bloque comprime un *gap* o un *run*. En el caso de codificar un *run* en los 31 bits sobrantes se impone como restricción que el mismo debe tener un largo $l \geq 32$, de modo de usar a lo más un bit por cada bit del *run*.

Capítulo 3

Propuesta

El algoritmo implementado se basa en las ideas propuestas en [Arroyuelo *et al.*, 2018], con algunas sutiles modificaciones que hacen la implementación más competitiva.

3.1. Estructura General del Método

La idea original del algoritmo aplica una segunda compresión sobre un vector sucinto comprimido con S9. Lo que busca es aprovechar de usar los bits que algunos casos no utilizan para codificar información sobre el bloque predecesor al actual. Asimismo crea un caso adicional completamente nuevo capaz de codificar *runs* con una longitud mayor a 28[bit].

Al igual que en S9, las palabras son un bloque de 32 bits, sin embargo, la cabecera es de tamaño variable, que va desde los 4 a los 6 bits. En total se definen 18 casos:

C1 1 bloque de 28 bits, con la cabecera `0000`

C2 2 bloques de 14 bits, con la cabecera `0001`

C3 3 bloques de 9 bits, con la cabecera `0010`

C4 4 bloques de 7 bits, con la cabecera `0011`

C5 7 bloques de 4 bits, con la cabecera `0100`

C6 9 bloques de 3 bits, con la cabecera `0101`

C7 14 bloques de 2 bits, con la cabecera `0110`

C8 28 bloques de 1 bit, seguidos de C1, con la cabecera `0111`

C9 28 bloques de 1 bit, seguidos de C2, con la cabecera `1000`

C10 28 bloques de 1 bit, seguidos de C3, con la cabecera `1001`

C11 28 bloques de 1 bit, seguidos de C4, con la cabecera `1010`

C12 28 bloques de 1 bit, seguidos de C5, con la cabecera `1011`

C13 28 bloques de 1 bit, seguidos de C6, con la cabecera `1100`

C14 28 bloques de 1 bit, seguidos de C7, con la cabecera `1101`

C15 28 bloques de 1 bit, seguidos de 5 bloques de 5 bits, con la cabecera `1110`

C16 28 bloques de 1 bit, con la cabecera `11111`

C17 5 bloques de 5 bits, con la cabecera `111100`

C18 Codifica en binario un *run* de $2 \leq l \leq 2^{26}$ bloques de 28 bits, con la cabecera `111101`

Es decir, S18 utiliza las ideas de S9, pero es un algoritmo de compresión híbrida. Asimismo, S18 es capaz de codificar *runs* de 28 bits 1 consecutivos de forma implícita en la cabecera.

3.2. Compresión *Greedy* de una Pasada

En el documento original, el autor propone S18 como un compresor que recibe como entrada un vector comprimido con S9, y que busca palabras consecutivas que forman los nuevos casos propuestos. La implementación que se propone en este documento es capaz de comprimir un vector de bits usando S18 sin requerir un paso de compresión previa usando S9.

Para esto, el algoritmo usa un acercamiento ávido, donde *gap* a *gap* decide si el mismo es codificable dentro de una palabra en memoria. Es decir, a medida que recorre el vector de bits va generando cada una de las palabras S18.

```
/* Create and pack word */
word w = word();
while (std::distance(gaps, end) > 0 and w.add_if_enough_space(*gaps))
    gaps++;
s18_seq[s18_seq_size++] = w.pack();
```

Algoritmo 1: Extracto de la rutina que codifica palabras S18 de forma greedy.

3.2.1. Palabras incompletas

El acercamiento de compresión de una pasada ávida puede producir palabras S18 que no utilizan todos los bloques. Para solucionar este caso especial, se considera que un *gap* de largo 0 termina la palabra. Es decir, en el momento de decodificar, si se encuentra un *gap* de largo igual a cero, la rutina continua con la siguiente palabra.

```
for (size_t i = 0; i < len; i++) {
        uint32_t wi = w.access_fast(i, _case);
        if (wi == 0) break; /* Word was not full */
  /* ... */
```

Algoritmo 2: Extracto de la rutina que decodifica palabras S18.

3.3. Unión de los casos C16 y C18

La propuesta original de S18 indica que el caso C16 codifica un *run* de 28 unos consecutivos de forma implícita y que por lo tanto, no es necesario guardar dichos valores. En el caso de esta implementación, se tiene que si es necesario codificar dichos valores, dado que el vector de bits puede terminar con un *run* de largo $14 < l \leq 28$, el cual sólo puede ser codificado usando una palabra C16.

Por este motivo, se modifica el C16, tal que guarda en binario el largo del *run*. Esto implica que ahora C16 puede codificar un `run` de largo $0 < l \leq 2^{27} - 1 = 134{,}217{,}727$. La implementación asume que un *run* de dicho largo es extremadamente improbable, y que por lo tanto, es muy probable que C16 siempre sea codificado con mayor prioridad que S18.

Por este motivo es que se excluye la codificación de C18 en la implementación, y en su lugar se usa C16. Esto implica también que no es necesario que el *run* codificado sea múltiplo de 28, lo cual hace la implementación más versátil.

Asimismo, unificar ambos casos resulta en una mejora de rendimiento, dado que, el código ya no debe realizar saltos a subrutinas distintas dependiendo del tipo de *run* a codificar, sino que al ser único, el código para procesar dichos *runs* consiste de un único bloque de código sin saltos condicionales.

3.4. Índice de bloques de palabras

Para poder responder rápidamente a consultas Access, Rank, Select, la implementación permite indizar bloques de b palabras (por defecto $b = 1$). Se implementan dos índices

`idx_bits` Cada entrada del índice indica cuantos bits en total preceden al bloque.

`idx_ones` Cada entrada del índice indica cuantos unos en total preceden al bloque.

Esto permite responder a las consultas

Access(i) Buscando $\max j$ tal que `idx_bits`$[j] \leq i$ y decodificar las palabras desde el bloque j-ésimo.

Rank(i) Buscando $\max j$ tal que `idx_bits`$[j] \leq i$ y decodificar las palabras desde el bloque j-ésimo, a lo cual se le adiciona `idx_ones`$[j]$.

Select(i) Buscando $\max j$ tal que `idx_ones`$[j] \leq i$ y decodificar las palabras desde el bloque j-ésimo, a lo cual se le adiciona `idx_bits`$[j]$.

Para realizar la búsqueda de j, se tiene como opción usar búsqueda binaria o lineal. Sin embargo, ambas opciones son relativamente lentas si se considera que las operaciones deben ser competitivas con las propuestas del estado del arte.

3.5. Un segundo nivel de indirección

Para mejorar el rendimiento de la búsqueda de j, la propuesta implementa un segundo nivel de indirección, es decir, se tiene índices adicionales. Si el largo del vector de bits es l, y dicho vector contiene m bits seteados (iguales a 1), entonces, para un vector S18 de w palabras

$$\texttt{l2_bits}[k] = \max j : \texttt{idx_bits}[j] \leq \left\lceil \frac{l}{w} \right\rceil$$

$$\texttt{l2_ones}[k] = \max j : \texttt{idx_ones}[j] \leq \left\lceil \frac{m}{w} \right\rceil$$

es decir, se divide el vector en bloques de misma largo de bits y de unos respectivamente, y se precalculan los resultados de la búsqueda binaria del bloque correspondiente al índice.

Esto permite acceder a un bloque cercano al deseado en tiempo constante, lo cual mejora los tiempos de acceso hasta en un 50 %, particularmente para Select, lo cual es observado de forma empírica durante la implementación.

3.6. Saltos rápidos en *runs*

La última optimización implementada en S18 es la de saltos rápidos en `runs`. Cuando la rutina de decodificación detecta un *run*, es capaz de decodificar todos los bits que empaqueta en tiempo constante. Por ejemplo

- Para Access, si el *offset* que queda por consumir es menor o igual al largo del *run*, el algoritmo retorna inmediatamente 1.
- Para Rank, donde la decodificación implica realizar una suma parcial *gap* a *gap*, cuando se detecta un *run* el algoritmo inmediatamente es capaz de sumar todos los unos del *run*, o bien sólo los que hacen falta para consumir el *offset*.

- Para SELECT, el algoritmo decodifica *runs* completos en tan sólo 4 operaciones aritméticas, sin importar el largo.

3.7. Resumen

Así, se tiene que la implementación final de la estructura usa compresión S18 y es capaz de agrupar las palabras en bloques indizados. Asimismo, los índices cuentan con estructuras auxiliares que hacen la estructura más competitiva en tiempo de cómputo.

En la figura 3.1 se puede apreciar cómo los índices de segundo nivel aproximan los resultados de la búsqueda binaria necesaria para obtener la posición de la casilla requerida de los índices normales. El índice de bits almacena una cuenta de los bits totales que preceden al bloque. Mientras que el índice de unos almacena la cuenta de unos que preceden al bloque. Para la figura se considera un tamaño de bloque igual a dos.

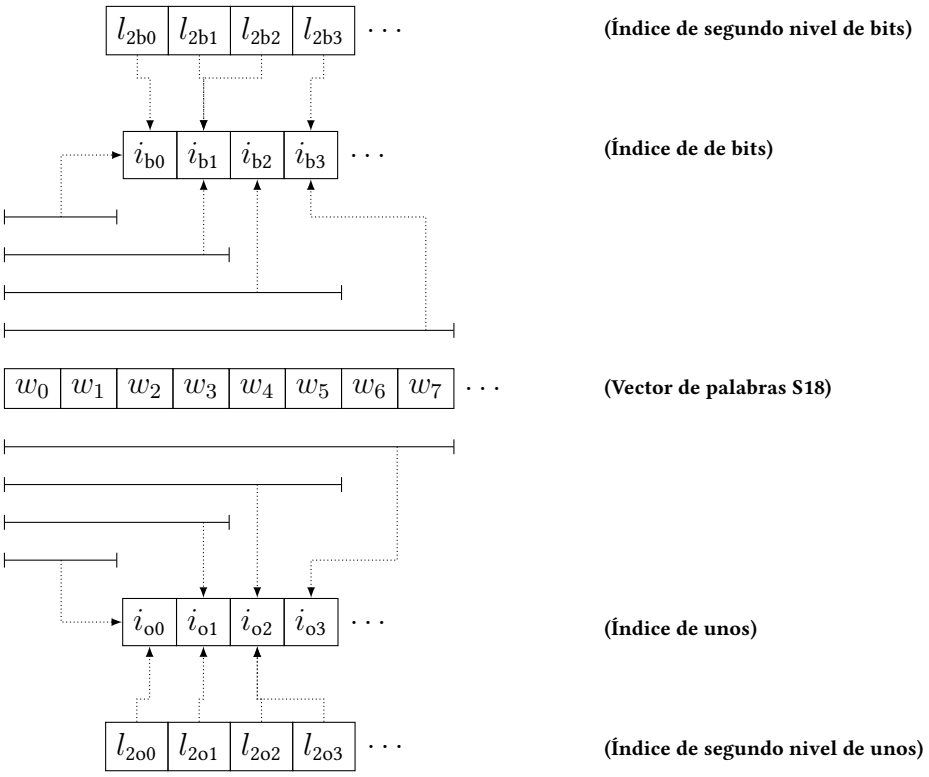


Figura 3.1: Representación gráfica de la implementación S18, incluyendo índices y estructuras auxiliares para asistir las operaciones de dicha SDS.

Capítulo 4

Implementación

La implementación contempla el código fuente necesario para cumplir con los requerimientos de la propuesta en el Capítulo 3, el cual puede ser analizado por el lector refiriéndose al Apéndice A.

En esta sección se presentan pruebas de rendimiento de la implementación para una serie de configuraciones. Para cada configuración se implementan pruebas que permiten hacer análisis de sensibilidad respecto de una única variable. El resto permanece invariante.

4.1. Metodología

Las pruebas de rendimiento se realizan sobre vectores sintéticos de diversa densidad y que contienen un número total predeterminado de *gaps* y *runs*. La probabilidad de que ocurra un *run* durante la construcción del vector de bits es un parámetro configurable. Para tener resultados agnósticos y aislar los efectos de operaciones que no son relevantes para cada prueba, se utiliza la librería de *benchmarks* de Google[1]. Entre las características de esta suite, se tiene que restringe la cantidad de iteraciones de cada prueba de tal modo de garantizar la estabilidad y significancia de los resultados obtenidos. Lo más relevante a destacar es que corre las pruebas por al menos un segundo, descarta las mediciones iniciales y espera a que el valor esperado del tiempo

de ejecución converja.

Cada caso de prueba construye un vector de bits idéntico para cada uno de los compresores a utilizar. Asimismo, cada caso de prueba se aplica sobre una familia de configuraciones para cada método (i.e. se aplica para diversos tamaños de bloques de cada vector sucinto).

Los vectores sintéticos poseen los siguientes atributos parametrizables

Probabilidad de run Fija la proporción de *runs* contra *gaps* presentes en el vector de bits. Para las pruebas se usan las configuraciones .01, .02, .03, .04, .05, .1 .2, .3, .4, .5, .6, .7, .8, .9, .95.

Cantidad de elementos del vector de bits Contempla la cantidad total de *gaps* + *runs* presentes en el vector de bits. Se considera que cada *run* es un único elemento. Durante las pruebas se fija este valor en 10,000.

Tamaño del bloque Para las estructuras que permiten organizar su representación interna en bloques de tamaño variable se prueban distintas configuraciones. Para todos los casos corresponden a las potencias de 2 entre 1 y 256, excepto RRR, para el cual se configuran los antecesores de dichas potencias, debido a que dicha estructura posee un mejor rendimiento para esos parámetros.

Las pruebas de rendimiento contemplan medir la latencia de las operaciones Access, Rank, Select y Successor para posiciones aleatorias del vector de bits.

El detalle completo de los resultados de todos los experimentos se encuentra disponible en el Apéndice B. Durante esta sección sólo se analizan los resultados más relevantes, que indican dónde se obtuvo el mejor y peor rendimiento. Análisis de sensibilidad para indicar bajo qué condiciones se logra un mejor rendimiento que para otros métodos descritos en el estado del arte. Por otro lado se indica si hubo algún resultado inesperado o que contradiga lo presentado como parte del marco teórico.

[1] `https://github.com/google/benchmark`

4.1.1. Hardware de Pruebas

El hardware utilizado para las pruebas tiene las siguientes características

- CPU: Intel(R) Core(TM) i5-6400 CPU @ 2.70[GHz]
- RAM: 16GB DDR4 a 2133[Mhz]
- SSD: 256GB
- HDD: 2TB

4.1.2. Software de Pruebas

El software utilizado para las pruebas corresponde a

- Sistema Operativo: Linux, con kernel en su versión 5.3.15
- Compilador: `g++` versión 9.2.0, el software es compilado usando el nivel de optimización `-O3`, asimismo se habilita el uso de instrucciones nativas para la arquitectura siendo probada (x86).
- Librería de pruebas: Google Benchmark, versión 1.5.0

4.2. Resultados y observaciones destacables

Entre las observaciones generales se destaca que la implementación de S18 presentada en este documento es capaz competir en espacio y/o latencia para una gran familia de operaciones. Si bien S18 se beneficia de *runs* de gran tamaño, se observa que el rendimiento en el caso de vectores de bits de poca densidad se encuentra en la vecindad de los métodos del estado del arte.

4.2.1. Respecto de la cota inferior teórica

Para todos los experimentos se calcula la entropía híbrida. Se tiene que el uso de espacio decrece asintóticamente a la entropía a medida que se aumenta el tamaño del bloque. Para todos los casos se tiene que el vector no gasta más de un orden de magnitud respecto del teórico esperado. Asimismo, se tiene que a medida que aumenta la densidad del vector, la diferencia de espacios teórico y empírico se acentúa más.

Esto último es esperado, dado que los *runs* largos tienen una mejor tasa de compresión en la representación que S18 les asigna (como palabra), que en el espacio que utilizan en los índices, el cual es proporcional a la cantidad de palabras que codifica S18 para representar el vector de bits.

4.2.2. Respecto del vector de bits

Para todas las pruebas se tiene que S18 tiene mayor latencia respecto del vector de bits normal, para todas las operaciones de acceso probadas.

A cambio, S18 provee de una representación compacta de los datos, y para todas las pruebas se obtuvo que S18 requiere de menos espacio de almacenamiento. Incluso considerando índices y estructuras adicionales requeridas por esta estructura sucinta.

4.2.3. Respecto de `s9_vector`

Respecto de S9, se tiene que S18 logra tener una distribución similar en forma respecto de latencia vs compresión, pero al ser equivalente a una compresión aplicada por sobre un vector S9 se tiene que en todos los casos la compresión es mejor. Esto aplica para todas las operaciones probadas.

Asimismo, se tiene que gracias a la familia de optimizaciones implementadas en el algoritmo de compresión descrito en este documento, también se logra una mejor latencia respecto de todas las operaciones en todos los casos de prueba realizados.

Si bien esto implica que S18 provee una mejor compresión, lo cual puede ser mostrado no solo teóricamente, sino que también de forma empírica, no es posible aseverar que

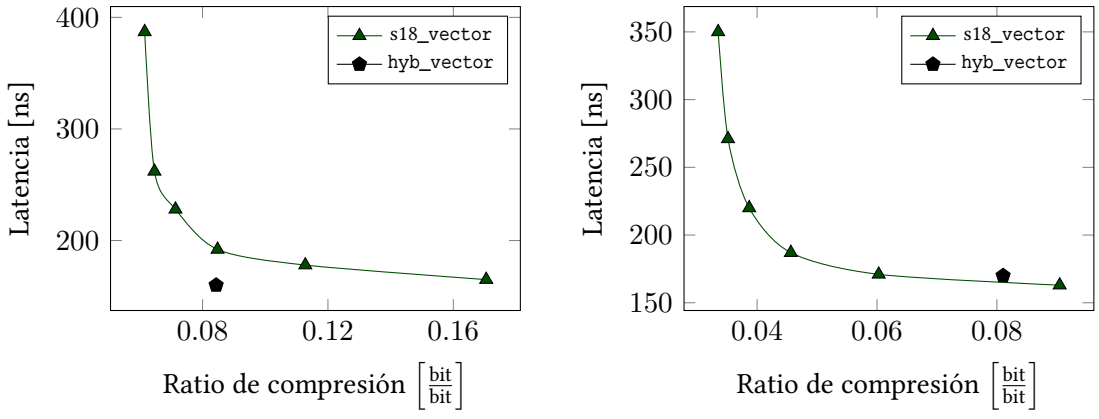
S18 siempre provee mejor latencia que S9, sino que la implementación específica de este documento es más competitiva en este aspecto.

4.2.4. Respecto de `hyb_vector`

La implementación de `hyb_vector` disponible en la SDSL no cuenta con la operación Select, por lo tanto no es posible realizar una comparación de este operador o sus derivados. Dado que dicho vector implementa un compresor híbrido, compite en la misma categoría que S18.

Para los experimentos realizados, se tiene que `hyb_vector` es mejor en todos los casos que prueban la operación Access. Esto es atribuible a que el patrón de acceso a los datos es más simple, dado que la compresión se realiza en bloques que representan la misma cantidad de bits.

Sin embargo, a pesar de que el patrón de acceso a los datos es más simple para `hyb_vector`, se tiene que para vectores densos ($d > 0.94$) S18 es capaz de competir y ganar en espacio y tiempo para la operación Rank. Esto es esperado, dado que en dichos casos, S18 no corta los runs en distintos bloques, lo que le permite no solo comprimir usando menos bloques, sino que calcular el operador en tiempo cuasi-constante (tan solo unas pocas operaciones aritméticas).



(a) Rendimiento de la operación Rank para una densidad menor a 0.94 %.

(b) Rendimiento de la operación Rank para una densidad mayor a 0.94 %.

Figura 4.1: Rendimiento de Rank para Hyb y S18 en vectores densos

4.2.5. Respecto de `rrr_vector`

Respecto de RRR, se tiene que S18 es capaz de competir siempre en al menos una dimensión. Para todas las pruebas realizadas se obtuvo de forma empírica que S18 provee una mejor tasa de compresión.

Para las operaciones Access y Rank se tiene que la curva de S18 se encuentra hacia la izquierda de la curva de RRR (mejor tasa de compresión), y dada la cercanía de las mismas se puede aseverar de forma empírica que S18 es una buena alternativa a RRR cuando se desea una mejor tasa de compresión con una latencia similar para dichas operaciones.

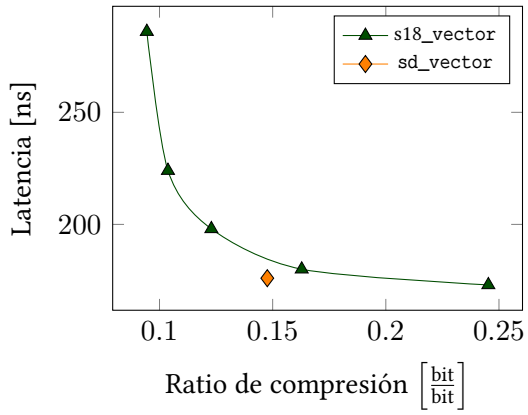
En el caso de Access, se tiene que las curvas de RRR y S18 se acercan a una distancia de aproximadamente 20[ns] en la vecindad de los 180[ns], asimismo se tiene que eso ocurre en una tasa de compresión que es cercana a los 0.3[bits por bit]. En el caso de Rank, se tiene que las curvas de RRR y S18 se juntan en la vecindad de los 190[ns] de latencia a una tasa de 0.25[bits por bit].

Respecto de Select se tiene que el patrón de acceso a los datos que implementa S18 le permite sacar una ventaja mayor a RRR, donde la tasa de compresión siempre es al menos 0.1[bits por bit] mejor y la latencia siempre es al menos 40[ns] más rápido. La mejora respecto de Select es tan sustancial, que incluso permite a S18 sacar ventaja para operaciones combinadas como Successor, donde a pesar de hacer uso de la operación Rank, la curva de S18 es mejor en ambos aspectos para todos los casos.

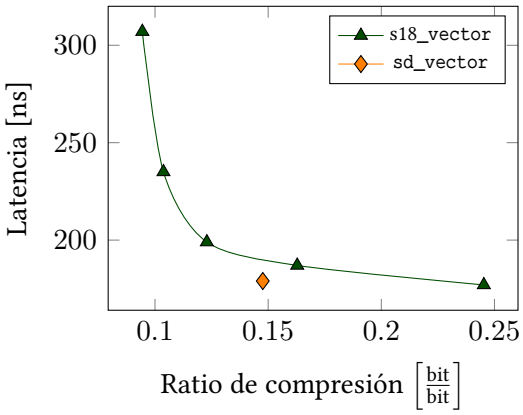
4.2.6. Respecto de `sd_vector`

Dada la naturaleza del funcionamiento de SD, es esperable que éste tenga un mejor rendimiento en cuanto a la tasa de compresión para vectores de baja densidad. Por este motivo resulta particularmente relevante comparar el rendimiento de vectores poco densos entre SD y S18.

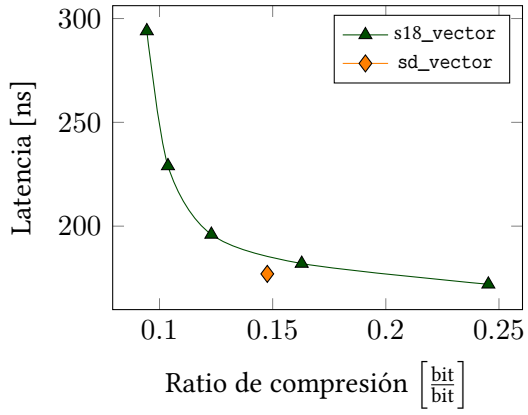
Los resultados con probabilidad de *run* entre 0.01 y 0.05 permiten realizar un análisis de sensibilidad en el cual queda mostrado que S18 es capaz de competir con SD una vez la densidad del vector de bits se vuelve mayor a $2\,\%$. Una vez se tiene un vector con



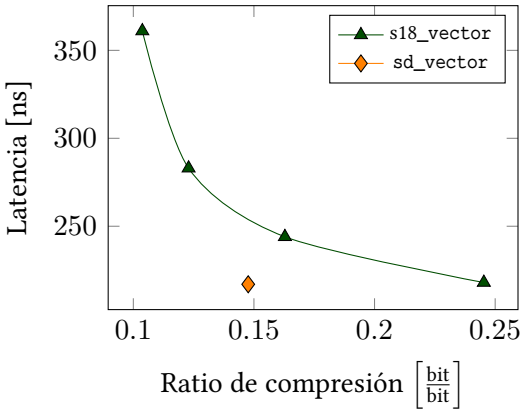
(a) Rendimiento de la operación Access para una densidad menor a 2 %.



(b) Rendimiento de la operación Rank para una densidad menor a 2 %.



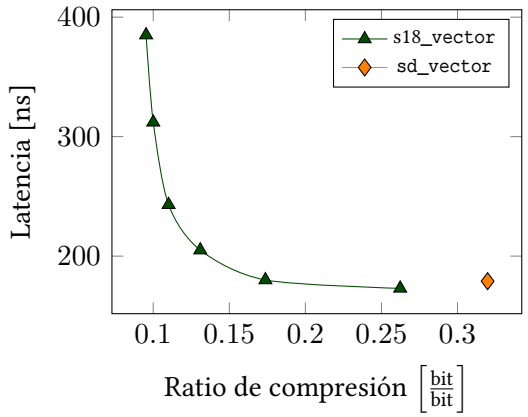
(c) Rendimiento de la operación Select para una densidad menor a 2 %.



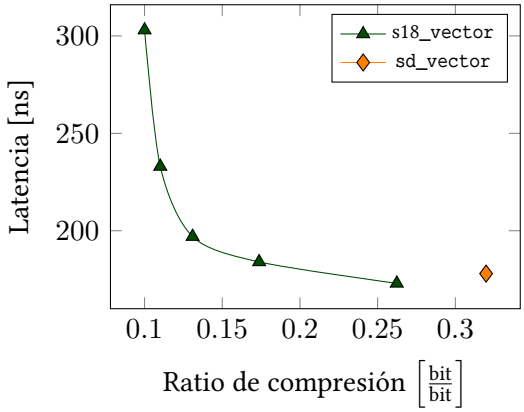
(d) Rendimiento de la operación Successor para una densidad menor a 2 %.

Figura 4.2: Rendimiento de operaciones SD y S18 para vector con densidad 1 %

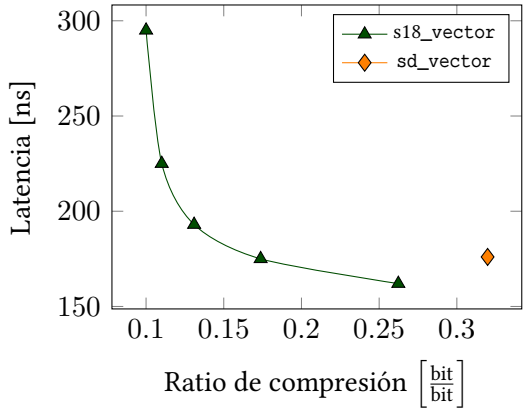
una densidad mayor a esa, la curva de rendimiento de S18 logra ser mejor en términos de latencia y compresión.



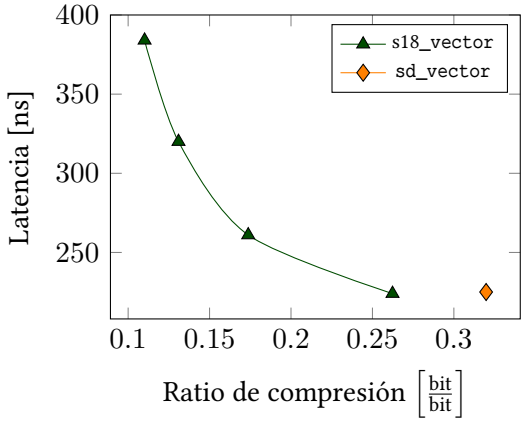
(a) Rendimiento de la operación Access para una densidad mayor a 2 %.



(b) Rendimiento de la operación Rank para una densidad mayor a 2 %.



(c) Rendimiento de la operación Select para una densidad mayor a 2 %.



(d) Rendimiento de la operación Successor para una densidad mayor a 2 %.

Figura 4.3: Rendimiento de operaciones SD y S18 para vector con densidad 3 %

Conclusiones

La gran cantidad de datos que se generan y deben procesar es un problema que requiere de soluciones donde la representación sucinta de los datos sobre los cuales se trabaja puede proveer de un mejor rendimiento, así como abaratar costos en escenarios cada vez más comunes, como lo son el de comprar poder de procesamiento y almacenamiento bajo demanda.

Bajo este escenario es donde se mostró que ya existe una familia de SDS para vectores de bits que ya han sido estudiadas por décadas, pero aún hay espacio para mejoras y nuevas implementaciones creativas. El estado del arte respalda aquello con implementaciones sucintas para el vector de bits que son capaces de ofrecer mejor rendimiento en algunas de las dimensiones relevantes para cada usuario.

Se mostró también ejemplos de problemas clásicos donde resulta relevante contar con implementaciones de vectores de bits, como por ejemplo para su aplicación en estructuras abstractas más complejas que requieren contar con una implementación sucinta del vector de bits.

En este documento se presentó una implementación concreta de el vector sucinto utilizando compresión híbrida S18. El método de compresión S18 provee de una mejor cota inferior teórica de compresión que su contraparte S9. Esto gracias a su capacidad de adaptarse localmente y adaptar el mecanismo de compresión aplicado en cada región del vector. Asimismo se mostró una serie de optimizaciones que pueden ser aplicadas sobre la idea original que permiten mejoras en rendimiento. Particularmente mejorando las latencias de las operaciones de dicha SDS. Todo esto fue realizado como una extensión a la *Succinct Data Structure Library* lo que permite que cualquier persona con conocimientos de programación en C++ pueda hacer uso de la estructura sin tener

ningún conocimiento de la implementación subyacente de la misma.

Se mostró que el uso de espacio de S18 decrece asintóticamente a la entropía híbrida a medida se aumenta el tamaño del bloque. Asimismo, se mostró que el ratio entre ambas no es mayor a un orden de magnitud en ningún caso. Es decir, S18 provee de una mejor tasa potencial de compresión en la teoría y queda respaldado con evidencia empírica.

Asimismo, se mostró de forma empírica que S18 es capaz de entregar una mejor tasa de compresión que otros métodos del estado del arte, y que incluso cuando requiere de estructuras adicionales para implementar las operaciones básicas del vector de bits, es capaz de representar de forma sucinta los datos subyacentes.

Al comparar S18 con sus contrapartes en el estado del arte se pudo observar que en todos los casos existe al menos un escenario donde el uso de esta nueva estructura es competitiva en alguna dimensión. Esto es particularmente relevante dado que amplía el espectro de escenarios donde es deseable usar una estructura de datos sucintas. Destaca por ejemplo que el rendimiento respecto de S9 es mejor en todas las dimensiones y escenarios. Asimismo destaca que S18 es competitivo con RRR para operaciones combinadas. Por otro lado se pudo observar que para vectores extremadamente densos S18 es capaz de entregar un mejor rendimiento en todas las dimensiones respecto de el método Hyb.

Queda como trabajo a futuro mostrar como S18 puede ayudar en la implementación de estructuras más complejas. Asimismo revisar las optimizaciones implementadas en el código, dado que si bien existe un trabajo creativo de por medio que mejora el rendimiento de las operaciones de acceso no se ha probado con alternativas que no requieran del uso de estructuras auxiliares.

Se propone también extender el alcance de esta implementación y adaptarlo para su uso en motores de bases de datos. Destacan por ejemplo aquellos que cuentan con implementaciones no sucintas de índices invertidos, como por ejemplo Lucene, ElasticSearch o el índice GIN de Postgres RDBMS. Si bien la implementación de este documento es capaz de representar datos de forma sucinta, no maneja control de acceso concurrente ni mecanismos que protegen la durabilidad e integridad de los datos (no es el objetivo ni está dentro del alcance posible en un trabajo como este).

Finalmente, dado el contexto en el que se desarrolla esta memoria, para optar al

título de Ingeniero Civil Informático de la Universidad Técnica Federico Santa María, es importante concluir que existe una serie de cursos que son un aporte directo al desarrollo de este documento.

Destacan por ejemplo, los cursos de Bases de Datos (INF-239) y Bases de Datos Avanzadas (INF-325) donde se enseña cómo distintos tipos de motores de base de datos son capaces de resolver distintos problemas de recuperación de información, y cómo éstos son capaces de indizar de forma transparente al usuario final los datos. Es en este curso donde se muestra la importancia de contar con estas estructuras, las cuales permiten recuperar la información rápidamente.

Por otro lado, en el curso de Compresión de Texto (INF-314) se enseñan los fundamentos y teoría de compresión, fundamentales para entender el estado del arte en Estructuras de Datos Sucintas. Asimismo entrega las herramientas necesarias para ser capaz de implementar una estructura novedosa como la presentada en este documento.

Asimismo, destacan los cursos de Estructura de Datos (INF-134) y Análisis de Algoritmos (INF-221), donde se muestra que un índice es una estructura de datos especializada en proveer de mecanismos para encontrar datos, y se enseña de forma práctica que la indirección a los datos permite implementar algorimos eficientes para procesar estas estructuras.

Finalmente, la rama de cursos de Ingeniería de Software que culmina con Feria de Software (INF-360), donde se enseña a gestionar los recursos para cumplir con el objetivo de desarollar un producto de software que resuelve un problema específico. Es gracias a esta línea de cursos que el autor logra medir sus avances en el tiempo, definir tareas y lograr terminar la implementación del producto a pesar de la gran carga y estrés que conlleva redactar una memoria.

Referencias

[Anh y Moffat, 2005] Anh, V. N. y Moffat, A. (2005). Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166.

[Arroyuelo *et al.*, 2018] Arroyuelo, D., Oyarzún, M., González, S., y Sepulveda, V. (2018). Hybrid compression of inverted lists for reordered document collections. *Information Processing & Management*.

[Elias, 1974] Elias, P. (1974). Efficient storage and retrieval by content and address of static files. *Journal of the ACM (JACM)*, 21(2):246–260.

[Fano, 1971] Fano, R. M. (1971). On the number of bits required to implement an associative memory. memorandum 61. *Computer Structures Group, Project MAC, MIT, Cambridge, Mass., nd*.

[Gog *et al.*, 2014] Gog, S., Beller, T., Moffat, A., y Petri, M. (2014). From theory to practice: Plug and play with succinct data structures. En *International Symposium on Experimental Algorithms*, páginas 326–337. Springer.

[Grossi *et al.*, 2003] Grossi, R., Gupta, A., y Vitter, J. S. (2003). High-order entropy-compressed text indexes. En *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, páginas 841–850. Society for Industrial and Applied Mathematics.

[Jacobson, 1988] Jacobson, G. J. (1988). *Succinct Static Data Structures*. Tesis doctoral, Carnegie Mellon University, Pittsburgh, PA, USA. AAI8918056.

[Karkkainen *et al.*, 2014] Karkkainen, J., Kempa, D., y Puglisi, S. J. (2014). Hybrid compression of bitvectors for the fm-index. En *Data Compression Conference (DCC), 2014*, páginas 302–311. IEEE.

[Okanohara y Sadakane, 2007] Okanohara, D. y Sadakane, K. (2007). Practical entropy-compressed rank/select dictionary. En *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, páginas 60–70. Society for Industrial and Applied Mathematics.

[Raman *et al.*, 2002] Raman, R., Raman, V., y Rao, S. S. (2002). Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. En *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, páginas 233–242. Society for Industrial and Applied Mathematics.

[Williams y Zobel, 1999] Williams, H. E. y Zobel, J. (1999). Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201.

[Wu *et al.*, 2002] Wu, K., Otoo, E. J., y Shoshani, A. (2002). Compressing bitmap indexes for faster search operations. En *Proceedings 14th International Conference on Scientific and Statistical Database Management*, páginas 99–108. IEEE.

[Wu *et al.*, 2006] Wu, K., Otoo, E. J., y Shoshani, A. (2006). Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems (TODS)*, 31(1):1–38.

[Zobel *et al.*, 1998] Zobel, J., Moffat, A., y Ramamohanarao, K. (1998). Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems (TODS)*, 23(4):453–490.

[Zukowski *et al.*, 2006] Zukowski, M., Heman, S., Nes, N., y Boncz, P. (2006). Super-scalar ram-cpu cache compression. En *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, páginas 59–59. IEEE.

Apéndice A

Código Fuente de S18

```
/* s18/head/constants.hpp */

#ifndef INCLUDED_SDSL_S18_CONSTANTS
#define INCLUDED_SDSL_S18_CONSTANTS

#define MASK_HEADER4 (0xF0000000)
#define MASK_HEADER5 (0xF8000000)
#define MASK_BODY4   (~MASK_HEADER4)
#define MASK_BODY5   (~MASK_HEADER5)

#define CASE01 (0x00000000)
#define CASE02 (0x10000000)
#define CASE03 (0x20000000)
#define CASE04 (0x30000000)
#define CASE05 (0x40000000)
#define CASE06 (0x50000000)
#define CASE07 (0x60000000)
#define CASE08 (0x70000000)
#define CASE09 (0x80000000)
#define CASE10 (0x90000000)
#define CASE11 (0xA0000000)
#define CASE12 (0xB0000000)
#define CASE13 (0xC0000000)
#define CASE14 (0xD0000000)
#define CASE15 (0xE0000000)
#define CASE16 (0xF0000000)
#define CASE17 (0xF8000000)

#define MASK_CASE01_CHUNK (0b00001111111111111111111111111111)
#define MASK_CASE02_CHUNK (0b00001111111111111100000000000000)
#define MASK_CASE03_CHUNK (0b00000111111111100000000000000000)
#define MASK_CASE04_CHUNK (0b00001111111000000000000000000000)
#define MASK_CASE05_CHUNK (0b00001111000000000000000000000000)
#define MASK_CASE06_CHUNK (0b00000111000000000000000000000000)
#define MASK_CASE07_CHUNK (0b00001100000000000000000000000000)
#define MASK_CASE08_CHUNK (0b00001111111111111111111111111111)
#define MASK_CASE09_CHUNK (0b00001111111111111100000000000000)
#define MASK_CASE10_CHUNK (0b00000111111111100000000000000000)
#define MASK_CASE11_CHUNK (0b00001111111000000000000000000000)
#define MASK_CASE12_CHUNK (0b00001111000000000000000000000000)
#define MASK_CASE13_CHUNK (0b00000111000000000000000000000000)
```

```
#define MASK_CASE14_CHUNK (0b00001100000000000000000000000000)
#define MASK_CASE15_CHUNK (0b00000001111100000000000000000000)
#define MASK_CASE16_CHUNK (0b00000111111111111111111111111111)
#define MASK_CASE17_CHUNK (0b00000001111100000000000000000000)

#define BITS_CASE01 (28)
#define BITS_CASE02 (14)
#define BITS_CASE03 ( 9)
#define BITS_CASE04 ( 7)
#define BITS_CASE05 ( 4)
#define BITS_CASE06 ( 3)
#define BITS_CASE07 ( 2)
#define BITS_CASE08 (28)
#define BITS_CASE09 (14)
#define BITS_CASE10 ( 9)
#define BITS_CASE11 ( 7)
#define BITS_CASE12 ( 4)
#define BITS_CASE13 ( 3)
#define BITS_CASE14 ( 2)
#define BITS_CASE15 ( 5)
#define BITS_CASE17 ( 5)

#define CHUNKS_CASE01 ( 1)
#define CHUNKS_CASE02 ( 2)
#define CHUNKS_CASE03 ( 3)
#define CHUNKS_CASE04 ( 4)
#define CHUNKS_CASE05 ( 7)
#define CHUNKS_CASE06 ( 9)
#define CHUNKS_CASE07 (14)
#define CHUNKS_CASE08 ( 1)
#define CHUNKS_CASE09 ( 2)
#define CHUNKS_CASE10 ( 3)
#define CHUNKS_CASE11 ( 4)
#define CHUNKS_CASE12 ( 7)
#define CHUNKS_CASE13 ( 9)
#define CHUNKS_CASE14 (14)
#define CHUNKS_CASE15 ( 5)
#define CHUNKS_CASE17 ( 5)

#endif
```

```
/* s18/head/s18_vector.hpp */

/*
 * s18::vector: An implementation for S18 compressed bitvectors
 * Copyright (c) 2019  Manuel Weitzman

 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, version 3 of the License.

 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.

 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <https://www.gnu.org/licenses/>.
 */
#ifndef INCLUDED_SDSL_S18_VECTOR
#define INCLUDED_SDSL_S18_VECTOR

```

```
#include <algorithm>
#include <cassert>
#include <cstdint>
#include <iterator>

#include <sdsl/int_vector.hpp>
#include <sdsl/vlc_vector.hpp>
#include <sdsl/util.hpp>

#include "constants.hpp"


namespace sdsl
{
namespace s18
{

/*
 * Forward declarations
 */

/* Access */
template<uint16_t b_s = 256, class vector_type = int_vector<32>>
class access_support;

/* Rank */
template<uint8_t q = 1, uint16_t b_s = 256, class vector_type = int_vector<32>>
class rank_support;

/* Select */
template<uint8_t q = 1, uint16_t b_s = 256, class vector_type = int_vector<32>>
class select_support;

/* S18 word */
class word;

/* S18 vector */
template<uint16_t b_s = 256, class vector_type = int_vector<32>>
class vector;


/*
 * S18 word
 */
class word
{
private:
        bool                 already_packed;
        bool                 processing_lead_1s;
        uint32_t             leading_1s;
        std::vector<uint32_t> pending_gaps;

        static size_t const   BIT_PAD[33];
        static size_t const   BITS_TO_CHUNKS[29];
        static size_t const   DECODER_CHUNKS[17];
        static size_t const   DECODER_BITS[17];
        static uint32_t const DECODER_MASK[17][14];
        static uint32_t const DECODER_SHIFT[17][14];
public:
        uint32_t value;
        size_t   chunk_size;
        word(void)
                : already_packed(false)
                , processing_lead_1s(true)
```

```
                , leading_1s(0)
                , pending_gaps()
                , value(0)
                , chunk_size(1)
        {}

        word(uint32_t const w)
                : already_packed(true)
                , processing_lead_1s(false)
                , leading_1s(0)
                , pending_gaps()
                , value(w)
                , chunk_size(0)
        {}

public:
        std::tuple<uint32_t, uint32_t, uint32_t> metadata(void) const
        {
                // Return leading 1s, case as int and len w/o leading ones
                switch (value & MASK_HEADER4) {
                        case CASE01: return std::make_tuple(0,  0, CHUNKS_CASE01);
                        case CASE02: return std::make_tuple(1,  0, CHUNKS_CASE02);
                        case CASE03: return std::make_tuple(2,  0, CHUNKS_CASE03);
                        case CASE04: return std::make_tuple(3,  0, CHUNKS_CASE04);
                        case CASE05: return std::make_tuple(4,  0, CHUNKS_CASE05);
                        case CASE06: return std::make_tuple(5,  0, CHUNKS_CASE06);
                        case CASE07: return std::make_tuple(6,  0, CHUNKS_CASE07);
                        case CASE08: return std::make_tuple(0, 28, CHUNKS_CASE01);
                        case CASE09: return std::make_tuple(1, 28, CHUNKS_CASE02);
                        case CASE10: return std::make_tuple(2, 28, CHUNKS_CASE03);
                        case CASE11: return std::make_tuple(3, 28, CHUNKS_CASE04);
                        case CASE12: return std::make_tuple(4, 28, CHUNKS_CASE05);
                        case CASE13: return std::make_tuple(5, 28, CHUNKS_CASE06);
                        case CASE14: return std::make_tuple(6, 28, CHUNKS_CASE07);
                        case CASE15: return std::make_tuple(16, 28, CHUNKS_CASE17);
                        default: switch (value & MASK_HEADER5) {
                                case CASE16: return std::make_tuple(15, (value & MASK_BODY5), 0);
                                case CASE17: return std::make_tuple(16, 0, CHUNKS_CASE17);
                                default: throw std::invalid_argument("word::metadata: "
                                        "Invalid case");
                        }
                }
        }

        uint32_t access_fast(size_t key, size_t _case) const
        {
                assert(_case != 15);
                return (value & DECODER_MASK[_case][key]) >> DECODER_SHIFT[_case][key];
        }

        bool add_if_enough_space(uint32_t gap)
        {
                assert(not already_packed);
                processing_lead_1s = processing_lead_1s and gap == 1;

                if (processing_lead_1s) {
                        if (leading_1s < 0x07FFFFFF) leading_1s += 1;
                        return leading_1s < 0x07FFFFFF;
                }

                if (leading_1s < 28) while (leading_1s) {
                        pending_gaps.push_back(1);
                        leading_1s--;
                }
```

```
                else if (leading_1s > 28) return false;

                size_t new_pending_size = pending_gaps.size() + 1;
                size_t gap_size = BIT_PAD[bits::hi(gap) + 1];
                size_t new_chunk_size = std::max(gap_size, chunk_size);
                if (new_pending_size * new_chunk_size > 28)
                        return false;

                pending_gaps.push_back(gap);
                chunk_size = std::max(gap_size, chunk_size);
                return true;
        }

        uint32_t pack(void)
        {
                assert(not already_packed);
                already_packed = true;

                /* Handler for C16 */
                if (chunk_size == 1 and leading_1s)
                        return (value = CASE16 | leading_1s);
                if (chunk_size == 1 and pending_gaps.size())
                        return (value = CASE16 | static_cast<uint32_t>(pending_gaps.size()));

                for (uint32_t gap : pending_gaps) {
                        value <<= chunk_size;
                        value |= gap;
                }
                value <<= chunk_size * (BITS_TO_CHUNKS[chunk_size] - pending_gaps.size());
                pending_gaps.clear();

                switch (chunk_size) {
                        case 28: value |= !leading_1s ? CASE01 : CASE08; break;
                        case 14: value |= !leading_1s ? CASE02 : CASE09; break;
                        case  9: value |= !leading_1s ? CASE03 : CASE10; break;
                        case  7: value |= !leading_1s ? CASE04 : CASE11; break;
                        case  4: value |= !leading_1s ? CASE05 : CASE12; break;
                        case  3: value |= !leading_1s ? CASE06 : CASE13; break;
                        case  2: value |= !leading_1s ? CASE07 : CASE14; break;
                        case  5: value |= !leading_1s ? CASE17 : CASE15; break;
                        default: throw std::invalid_argument("word::pack: Invalid case");
                }

                return value;
        }

#if DEBUG
        size_t size(void) const {
                switch (value & MASK_HEADER4) {
                        case CASE01: return CHUNKS_CASE01;
                        case CASE02: return CHUNKS_CASE02;
                        case CASE03: return CHUNKS_CASE03;
                        case CASE04: return CHUNKS_CASE04;
                        case CASE05: return CHUNKS_CASE05;
                        case CASE06: return CHUNKS_CASE06;
                        case CASE07: return CHUNKS_CASE07;
                        case CASE08: return CHUNKS_CASE08 + 28;
                        case CASE09: return CHUNKS_CASE09 + 28;
                        case CASE10: return CHUNKS_CASE10 + 28;
                        case CASE11: return CHUNKS_CASE11 + 28;
                        case CASE12: return CHUNKS_CASE12 + 28;
                        case CASE13: return CHUNKS_CASE13 + 28;
                        case CASE14: return CHUNKS_CASE14 + 28;
                        case CASE15: return CHUNKS_CASE15 + 28;
```

```
                        default: switch (value & MASK_HEADER5) {
                                case CASE16: return value & MASK_BODY5;
                                case CASE17: return CHUNKS_CASE17;
                                default: throw std::invalid_argument("word::size: Invalid case");
                        }
                }
        }

        uint32_t operator[](size_t key) const
        {
                switch (value & MASK_HEADER4) {
                        case CASE01:
                                return (value & (MASK_CASE01_CHUNK >> (key * BITS_CASE01)))
                                        >> (BITS_CASE01 * (CHUNKS_CASE01 - key - 1));
                        case CASE02:
                                return (value & (MASK_CASE02_CHUNK >> (key * BITS_CASE02)))
                                        >> (BITS_CASE02 * (CHUNKS_CASE02 - key - 1));
                        case CASE03:
                                return (value & (MASK_CASE03_CHUNK >> (key * BITS_CASE03)))
                                        >> (BITS_CASE03 * (CHUNKS_CASE03 - key - 1));
                        case CASE04:
                                return (value & (MASK_CASE04_CHUNK >> (key * BITS_CASE04)))
                                        >> (BITS_CASE04 * (CHUNKS_CASE04 - key - 1));
                        case CASE05:
                                return (value & (MASK_CASE05_CHUNK >> (key * BITS_CASE05)))
                                        >> (BITS_CASE05 * (CHUNKS_CASE05 - key - 1));
                        case CASE06:
                                return (value & (MASK_CASE06_CHUNK >> (key * BITS_CASE06)))
                                        >> (BITS_CASE06 * (CHUNKS_CASE06 - key - 1));
                        case CASE07:
                                return (value & (MASK_CASE07_CHUNK >> (key * BITS_CASE07)))
                                        >> (BITS_CASE07 * (CHUNKS_CASE07 - key - 1));
                        case CASE08:
                                return key < 28 ? 1 :
                                        (value & (MASK_CASE08_CHUNK >> ((key - 28) * BITS_CASE08)))
                                        >> (BITS_CASE08 * (CHUNKS_CASE08 - (key - 28) - 1));
                        case CASE09:
                                return key < 28 ? 1 :
                                        (value & (MASK_CASE09_CHUNK >> ((key - 28) * BITS_CASE09)))
                                        >> (BITS_CASE09 * (CHUNKS_CASE09 - (key - 28) - 1));
                        case CASE10:
                                return key < 28 ? 1 :
                                        (value & (MASK_CASE10_CHUNK >> ((key - 28) * BITS_CASE10)))
                                        >> (BITS_CASE10 * (CHUNKS_CASE10 - (key - 28) - 1));
                        case CASE11:
                                return key < 28 ? 1 :
                                        (value & (MASK_CASE11_CHUNK >> ((key - 28) * BITS_CASE11)))
                                        >> (BITS_CASE11 * (CHUNKS_CASE11 - (key - 28) - 1));
                        case CASE12:
                                return key < 28 ? 1 :
                                        (value & (MASK_CASE12_CHUNK >> ((key - 28) * BITS_CASE12)))
                                        >> (BITS_CASE12 * (CHUNKS_CASE12 - (key - 28) - 1));
                        case CASE13:
                                return key < 28 ? 1 :
                                        (value & (MASK_CASE13_CHUNK >> ((key - 28) * BITS_CASE13)))
                                        >> (BITS_CASE13 * (CHUNKS_CASE13 - (key - 28) - 1));
                        case CASE14:
                                return key < 28 ? 1 :
                                        (value & (MASK_CASE14_CHUNK >> ((key - 28) * BITS_CASE14)))
                                        >> (BITS_CASE14 * (CHUNKS_CASE14 - (key - 28) - 1));
                        case CASE15:
                                return key < 28 ? 1 :
                                        (value & (MASK_CASE15_CHUNK >> ((key - 28) * BITS_CASE15)))
                                        >> (BITS_CASE15 * (CHUNKS_CASE15 - (key - 28) - 1));
```

```
                        default: switch (value & MASK_HEADER5) {
                                case CASE16:
                                        return key < (value & MASK_CASE16_CHUNK);
                                case CASE17:
                                        return (value & (MASK_CASE17_CHUNK >> (key * BITS_CASE17)))
                                                >> (BITS_CASE17 * (CHUNKS_CASE17 - key - 1));
                                default:
                                        throw std::invalid_argument("word::operator[]: "
                                                "Invalid case");
                        }
                }
        }
#endif
};

size_t const word::BIT_PAD[33] = { 1, 1, 2, 3, 4, 5, 7, 7, 9, 9, 14, 14, 14, 14, 14,
        28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28 };
size_t const word::BITS_TO_CHUNKS[29] =
        {0,28,14,9,7,5,0,4,0,3,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,1};
size_t const word::DECODER_CHUNKS[17] = {1,2,3,4,7,9,14,1,2,3,4,7,9,14,5,0,5};
size_t const word::DECODER_BITS[17] = {28,14,9,7,4,3,2,28,14,9,7,4,3,2,5,0,5};
uint32_t const word::DECODER_MASK[17][14] = {
        {MASK_CASE01_CHUNK >> (0 * BITS_CASE01)},
        {MASK_CASE02_CHUNK >> (0 * BITS_CASE02), MASK_CASE02_CHUNK >> (1 * BITS_CASE02)},
        {MASK_CASE03_CHUNK >> (0 * BITS_CASE03), MASK_CASE03_CHUNK >> (1 * BITS_CASE03),
                MASK_CASE03_CHUNK >> (2 * BITS_CASE03)},
        {MASK_CASE04_CHUNK >> (0 * BITS_CASE04), MASK_CASE04_CHUNK >> (1 * BITS_CASE04),
                MASK_CASE04_CHUNK >> (2 * BITS_CASE04), MASK_CASE04_CHUNK >> (3 * BITS_CASE04)},
        {MASK_CASE05_CHUNK >> (0 * BITS_CASE05), MASK_CASE05_CHUNK >> (1 * BITS_CASE05),
                MASK_CASE05_CHUNK >> (2 * BITS_CASE05), MASK_CASE05_CHUNK >> (3 * BITS_CASE05),
                MASK_CASE05_CHUNK >> (4 * BITS_CASE05), MASK_CASE05_CHUNK >> (5 * BITS_CASE05),
                MASK_CASE05_CHUNK >> (6 * BITS_CASE05)},
        {MASK_CASE06_CHUNK >> (0 * BITS_CASE06), MASK_CASE06_CHUNK >> (1 * BITS_CASE06),
                MASK_CASE06_CHUNK >> (2 * BITS_CASE06), MASK_CASE06_CHUNK >> (3 * BITS_CASE06),
                MASK_CASE06_CHUNK >> (4 * BITS_CASE06), MASK_CASE06_CHUNK >> (5 * BITS_CASE06),
                MASK_CASE06_CHUNK >> (6 * BITS_CASE06), MASK_CASE06_CHUNK >> (7 * BITS_CASE06),
                MASK_CASE06_CHUNK >> (8 * BITS_CASE06)},
        {MASK_CASE07_CHUNK >> ( 0 * BITS_CASE07), MASK_CASE07_CHUNK >> ( 1 * BITS_CASE07),
                MASK_CASE07_CHUNK >> ( 2 * BITS_CASE07), MASK_CASE07_CHUNK >> ( 3 * BITS_CASE07),
                MASK_CASE07_CHUNK >> ( 4 * BITS_CASE07), MASK_CASE07_CHUNK >> ( 5 * BITS_CASE07),
                MASK_CASE07_CHUNK >> ( 6 * BITS_CASE07), MASK_CASE07_CHUNK >> ( 7 * BITS_CASE07),
                MASK_CASE07_CHUNK >> ( 8 * BITS_CASE07), MASK_CASE07_CHUNK >> ( 9 * BITS_CASE07),
                MASK_CASE07_CHUNK >> (10 * BITS_CASE07), MASK_CASE07_CHUNK >> (11 * BITS_CASE07),
                MASK_CASE07_CHUNK >> (12 * BITS_CASE07), MASK_CASE07_CHUNK >> (13 * BITS_CASE07)},
        {MASK_CASE08_CHUNK >> (0 * BITS_CASE08)},
        {MASK_CASE09_CHUNK >> (0 * BITS_CASE09), MASK_CASE09_CHUNK >> (1 * BITS_CASE09)},
        {MASK_CASE10_CHUNK >> (0 * BITS_CASE10), MASK_CASE10_CHUNK >> (1 * BITS_CASE10),
                MASK_CASE10_CHUNK >> (2 * BITS_CASE10)},
        {MASK_CASE11_CHUNK >> (0 * BITS_CASE11), MASK_CASE11_CHUNK >> (1 * BITS_CASE11),
                MASK_CASE11_CHUNK >> (2 * BITS_CASE11), MASK_CASE11_CHUNK >> (3 * BITS_CASE11)},
        {MASK_CASE12_CHUNK >> (0 * BITS_CASE12), MASK_CASE12_CHUNK >> (1 * BITS_CASE12),
                MASK_CASE12_CHUNK >> (2 * BITS_CASE12), MASK_CASE12_CHUNK >> (3 * BITS_CASE12),
                MASK_CASE12_CHUNK >> (4 * BITS_CASE12), MASK_CASE12_CHUNK >> (5 * BITS_CASE12),
                MASK_CASE12_CHUNK >> (6 * BITS_CASE12)},
        {MASK_CASE13_CHUNK >> (0 * BITS_CASE13), MASK_CASE13_CHUNK >> (1 * BITS_CASE13),
                MASK_CASE13_CHUNK >> (2 * BITS_CASE13), MASK_CASE13_CHUNK >> (3 * BITS_CASE13),
                MASK_CASE13_CHUNK >> (4 * BITS_CASE13), MASK_CASE13_CHUNK >> (5 * BITS_CASE13),
                MASK_CASE13_CHUNK >> (6 * BITS_CASE13), MASK_CASE13_CHUNK >> (7 * BITS_CASE13),
                MASK_CASE13_CHUNK >> (8 * BITS_CASE13)},
        {MASK_CASE14_CHUNK >> ( 0 * BITS_CASE14), MASK_CASE14_CHUNK >> ( 1 * BITS_CASE14),
                MASK_CASE14_CHUNK >> ( 2 * BITS_CASE14), MASK_CASE14_CHUNK >> ( 3 * BITS_CASE14),
                MASK_CASE14_CHUNK >> ( 4 * BITS_CASE14), MASK_CASE14_CHUNK >> ( 5 * BITS_CASE14),
                MASK_CASE14_CHUNK >> ( 6 * BITS_CASE14), MASK_CASE14_CHUNK >> ( 7 * BITS_CASE14),
                MASK_CASE14_CHUNK >> ( 8 * BITS_CASE14), MASK_CASE14_CHUNK >> ( 9 * BITS_CASE14),
```

```
                MASK_CASE14_CHUNK >> (10 * BITS_CASE14), MASK_CASE14_CHUNK >> (11 * BITS_CASE14),
                MASK_CASE14_CHUNK >> (12 * BITS_CASE14), MASK_CASE14_CHUNK >> (13 * BITS_CASE14)},
        {MASK_CASE15_CHUNK >> (0 * BITS_CASE15), MASK_CASE15_CHUNK >> (1 * BITS_CASE15),
                MASK_CASE15_CHUNK >> (2 * BITS_CASE15), MASK_CASE15_CHUNK >> (3 * BITS_CASE15),
                MASK_CASE15_CHUNK >> (4 * BITS_CASE15)},
        {},
        {MASK_CASE17_CHUNK >> (0 * BITS_CASE17), MASK_CASE17_CHUNK >> (1 * BITS_CASE17),
                MASK_CASE17_CHUNK >> (2 * BITS_CASE17), MASK_CASE17_CHUNK >> (3 * BITS_CASE17),
                MASK_CASE17_CHUNK >> (4 * BITS_CASE17)}
};
uint32_t const word::DECODER_SHIFT[17][14] = {
        {0},
        {14,0},
        {18,9,0},
        {21,14,7,0},
        {24,20,16,12,8,4,0},
        {24,21,18,15,12,9,6,3,0},
        {26,24,22,20,18,16,14,12,10,8,6,4,2,0},
        {0},
        {14,0},
        {18,9,0},
        {21,14,7,0},
        {24,20,16,12,8,4,0},
        {24,21,18,15,12,9,6,3,0},
        {26,24,22,20,18,16,14,12,10,8,6,4,2,0},
        {20,15,10,5,0},
        {},
        {20,15,10,5,0},
};


/*
 * S18 Vector
 */
template<uint16_t b_s, class vector_type>
class vector
{
public:
        friend class access_support<b_s>;
        friend class rank_support<0, b_s>;
        friend class rank_support<1, b_s>;
        friend class select_support<0, b_s>;
        friend class select_support<1, b_s>;

        typedef typename vector_type::iterator       iterator_type;
        typedef typename vector_type::const_iterator const_iterator_type;
        typedef typename vector_type::size_type      size_type;

private:
        size_t         m_ones;        // 1 bits in original sequence
        size_t         m_size;        // Lenth of original bit vector
        size_t         s18_seq_size;  // Count of S18 words
        int_vector<32> s18_seq;       // Vector of S18 words
        int_vector<>   idx_bits;      // Total bits before block
        int_vector<>   idx_ones;      // Total 1 bits before block
        int_vector<>   l2_bits;       // Pre-computed idx_bits bisection
        int_vector<>   l2_ones;       // Pre-computed idx_ones bisection
        size_t         l2_bits_div;   // Division constant for l2_bits
        size_t         l2_ones_div;   // Division constant for l2_ones

public:
        /* Default constructor */
        vector(void)=delete;

```

```
406         /* Copy constructor */
407         vector(vector const &other) /* copy */
408                 : m_ones(other.m_ones)
409                 , m_size(other.m_size)
410                 , s18_seq_size(other.s18_seq_size)
411                 , s18_seq(other.s18_seq)
412                 , idx_bits(other.idx_bits)
413                 , idx_ones(other.idx_ones)
414                 , l2_bits(other.l2_bits)
415                 , l2_ones(other.l2_ones)
416                 , l2_bits_div(other.l2_bits_div)
417                 , l2_ones_div(other.l2_ones_div)
418         {} /* end vector::vector */
419
420         /* Move constructor */
421         vector(vector const &&other) /* move */
422         {
423                 *this = std::move(other);
424         } /* end vector::vector */
425
426         /* Constructor from bitvector */
427         vector(bit_vector const &bv)
428                 : m_ones(util::cnt_one_bits(bv))
429                 , m_size(bv.size())
430                 , s18_seq_size(0)
431                 , s18_seq(m_ones, 0)
432                 , idx_bits(m_ones / b_s + 2, 0)
433                 , idx_ones(m_ones / b_s + 2, 0)
434                 , l2_bits(0, 0)
435                 , l2_ones(0, 0)
436                 , l2_bits_div(1)
437                 , l2_ones_div(1)
438         {
439                 /* Get absolute positions */
440                 vector_type absp = vector_type(m_ones, 0);
441                 for (size_t i = 0, j = 0; i < m_size; i++)
442                         if (bv[i]) absp[j++] = static_cast<uint32_t>(i);
443
444                 /* Get gaps from absolute positions */
445                 vector_type gaps = vector_type(m_ones, 0);
446                 for (size_t i = 1; i < m_ones; i++)
447                         gaps[i] = absp[i] - absp[i - 1];
448                 gaps[0] = absp[0] + 1;
449
450                 /* Indexes indices */
451                 size_t size_idx_bits = 1;
452                 size_t size_idx_ones = 1;
453
454                 /* Encode gaps into s18 words */
455                 const_iterator_type gap = gaps.begin();
456                 const_iterator_type const begin = gaps.begin();
457                 const_iterator_type const end = gaps.end();
458                 while (std::distance(gap, end) > 0) {
459                         for (size_t i = 0; i < b_s; i++)
460                                 gap = pack_word(gap, end);
461
462                         size_t gaps_encoded = std::distance(begin, gap);
463                         idx_bits[size_idx_bits++] = absp[gaps_encoded - 1] + 1;
464                         idx_ones[size_idx_ones++] = static_cast<uint32_t>(gaps_encoded);
465                 }
466
467                 /* Get rid of extra unused space */
468                 s18_seq.resize(s18_seq_size);
469                 idx_bits.resize(size_idx_bits);
```

```
                idx_ones.resize(size_idx_ones);

                /* Build L2 index */
                size_t size_l2 = size_idx_bits;
                l2_bits.resize(size_l2);
                l2_ones.resize(size_l2);

                l2_bits_div = m_size / size_l2 + (m_size % size_l2 != 0);
                for (size_t i = 0; i < size_l2; i++) {
                        auto it = std::upper_bound(idx_bits.begin(), idx_bits.end(),
                                i * l2_bits_div);
                        l2_bits[i] = static_cast<uint32_t>(std::distance(idx_bits.begin(), it));
                }

                l2_ones_div = (m_ones + 1) / size_l2 + ((m_ones + 1) % size_l2 != 0);
                for (size_t i = 0; i < size_l2; i++) {
                        auto it = std::upper_bound(idx_ones.begin(), idx_ones.end(),
                                i * l2_ones_div);
                        l2_ones[i] = static_cast<uint32_t>(std::distance(idx_ones.begin(), it));
                }

                util::bit_compress(idx_bits);
                util::bit_compress(idx_ones);
                util::bit_compress(l2_bits);
                util::bit_compress(l2_ones);
        } /* end vector::vector */

        size_t size(void) const
        {
                return m_size;
        }

        uint32_t slow_access(size_t const key) const
        {
                return find_block_nth(
                        s18_seq.begin(),
                        s18_seq.end(),
                        static_cast<uint32_t>(key)
                );
        }

        uint32_t operator[](size_t const key) const
        {
                uint32_t pos = static_cast<uint32_t>(l2_bits[key / l2_bits_div] - 1);
                return find_block_nth(
                        s18_seq.begin() + pos * b_s,
                        s18_seq.end(),
                        static_cast<uint32_t>(key) - static_cast<uint32_t>(idx_bits[pos])
                );
        }

        int_vector<32> const &data(void) const
        {
                return s18_seq;
        }

        size_t serialize(std::ostream& out, structure_tree_node* v=nullptr,
                std::string name="") const
        {
                structure_tree_node* child =
                        structure_tree::add_child(v, name, util::class_name(*this));

                size_t written_bytes = 0;
                written_bytes += write_member(m_ones, out, child, "m_ones");
```

```
534                 written_bytes += write_member(m_size, out, child, "m_size");
535                 written_bytes += write_member(s18_seq_size, out, child, "s18_seq_size");
536                 written_bytes += write_member(l2_bits_div, out, child, "l2_bits_div");
537                 written_bytes += write_member(l2_ones_div, out, child, "l2_ones_div");
538 
539                 written_bytes += s18_seq.serialize(out, child, "s18_seq");
540                 written_bytes += idx_bits.serialize(out, child, "idx_bits");
541                 written_bytes += idx_ones.serialize(out, child, "idx_ones");
542                 written_bytes += l2_bits.serialize(out, child, "l2_bits");
543                 written_bytes += l2_ones.serialize(out, child, "l2_ones");
544 
545                 structure_tree::add_size(child, written_bytes);
546 
547                 return written_bytes;
548         }
549 
550 private:
551         uint32_t find_block_nth(
552                 const_iterator_type const begin, const_iterator_type const end,
553                 uint32_t target_accum) const
554         {
555                 const_iterator_type gaps = begin;
556                 uint32_t accum = -1;
557 
558                 for (; std::distance(gaps, end) > 0; gaps++) {
559                         word w(*gaps);
560                         auto const [_case, leading_1s, len] = w.metadata();
561 
562                         if (leading_1s and (accum += leading_1s) >= target_accum)
563                                 return 1;
564 
565                         for (size_t i = 0; i < len; i++) {
566                                 uint32_t wi = w.access_fast(i, _case);
567                                 if (wi == 0) break; /* Word was not full */
568 
569                                 accum += wi;
570                                 if (accum == target_accum) return 1;
571                                 if (accum > target_accum) return 0;
572                         }
573                 }
574 
575                 return 0;
576         }
577 
578         inline const_iterator_type pack_word(
579                 const_iterator_type const begin, const_iterator_type const end)
580         {
581                 /* Early return */
582                 if (std::distance(begin, end) <= 0)
583                         return begin;
584 
585                 /* Pointer to gaps */
586                 const_iterator_type gaps = begin;
587 
588                 /* Create and pack word */
589                 word w = word();
590                 while (std::distance(gaps, end) > 0 and w.add_if_enough_space(*gaps)) gaps++;
591                 s18_seq[s18_seq_size++] = w.pack();
592 
593                 /* Return iterator to the very next compressable gap */
594                 return gaps;
595         }
596 };
597 
```

```
598
599 template<uint16_t b_s, class vector_type>
600 class access_support
601 {
602 private:
603         vector<b_s, vector_type> const &bv;
604 public:
605         access_support(void)=delete;
606         access_support(vector<b_s, vector_type> &cv)
607                 : bv(cv)
608         {}
609         uint32_t operator()(size_t const key) const { return bv[key]; }
610
611 };
612
613 template<uint8_t q, uint16_t b_s, class vector_type>
614 class rank_support
615 {
616 static_assert(q < 2, "rank_support: bit pattern must be 0 or 1");
617 private:
618         vector<b_s, vector_type> const &bv;
619
620         typedef typename vector_type::iterator       iterator_type;
621         typedef typename vector_type::const_iterator const_iterator_type;
622
623 private:
624         size_t rank0(size_t const key) const
625         {
626                 return key - rank1(key);
627         }
628
629         size_t rank1(size_t const key) const
630         {
631                 uint32_t pos = static_cast<uint32_t>(bv.l2_bits[key / bv.l2_bits_div] - 1);
632                 return bv.idx_ones[pos] + find_block_nth(
633                         bv.s18_seq.begin() + pos * b_s,
634                         bv.s18_seq.end(),
635                         static_cast<uint32_t>(key) - static_cast<uint32_t>(bv.idx_bits[pos])
636                 );
637         }
638
639         size_t find_block_nth(
640                 const_iterator_type const begin, const_iterator_type const end,
641                 uint32_t target_accum) const
642         {
643                 const_iterator_type gaps = begin;
644                 int_fast64_t accum = -1;
645                 int_fast64_t one_cnt = 0;
646
647                 for (; std::distance(gaps, end) > 0; gaps++) {
648                         word w(*gaps);
649                         auto const [_case, leading_1s, len] = w.metadata();
650
651                         if (accum + leading_1s >= target_accum)
652                                 return one_cnt + target_accum - accum - 1;
653
654                         accum += leading_1s;
655                         one_cnt += leading_1s;
656
657                         for (size_t i = 0; i < len; i++, one_cnt++) {
658                                 int_fast32_t wi = w.access_fast(i, _case);
659                                 if (wi == 0) break; /* Word was not full */
660
661                                 accum += wi;
662                                 if (accum >= target_accum) return one_cnt;
```

```
663                         }
664                 }
665
666                 return one_cnt;
667         }
668 public:
669         rank_support(void)=delete;
670         rank_support(vector<b_s, vector_type> &cv)
671                 : bv(cv)
672         {}
673         size_t operator()(size_t const key) const
674         {
675                 return q ? rank1(key) : rank0(key);
676         }
677 };
678
679 template<uint8_t q, uint16_t b_s, class vector_type>
680 class select_support
681 {
682 static_assert(q < 2, "select_support: bit pattern must be 0 or 1");
683 private:
684         vector<b_s, vector_type> const &bv;
685
686         typedef typename vector_type::iterator       iterator_type;
687         typedef typename vector_type::const_iterator const_iterator_type;
688
689 private:
690         size_t select0(size_t const key) const
691         {
692                 return key;
693         }
694
695         size_t select1(size_t const key) const
696         {
697                 uint32_t pos = static_cast<uint32_t>(bv.l2_ones[key / bv.l2_ones_div] - 1);
698                 return bv.idx_bits[pos] + partial_sum(
699                         bv.s18_seq.begin() + pos * b_s,
700                         bv.s18_seq.end(),
701                         static_cast<uint32_t>(key) - static_cast<uint32_t>(bv.idx_ones[pos])
702                 );
703         }
704
705         size_t partial_sum(
706                 const_iterator_type const begin, const_iterator_type const end,
707                 uint32_t counter) const
708         {
709                 const_iterator_type gaps = begin;
710                 size_t accum = 0;
711
712                 for (; std::distance(gaps, end) > 0 and counter; gaps++) {
713                         word w(*gaps);
714                         auto const [_case, leading_1s, len] = w.metadata();
715
716                         accum += std::min(counter, leading_1s);
717                         counter -= std::min(counter, leading_1s);
718
719                         for (size_t i = 0; i < len and counter; i++, counter--) {
720                                 uint32_t wi = w.access_fast(i, _case);
721                                 if (wi == 0) break; /* Word was not full */
722                                 accum += wi;
723                         }
724                 }
725
726 #if DEBUG
```

```
727                 if (counter)
728                         throw std::runtime_error("select_support_s18::partial_sum: "
729                                 "vector consumed before target counter");
730 #endif
731                 return accum;
732         }
733 public:
734         select_support(void)=delete;
735         select_support(vector<b_s, vector_type> &cv)
736                 : bv(cv)
737         {}
738         size_t operator()(size_t const key) const
739         {
740                 return q ? select1(key) : select0(key);
741         }
742 };
743
744 } /* namespace s18 */
745 } /* namespace sdsl */
746
747 #endif
```

Apéndice B

Datos de Pruebas de Rendimiento

B.1. Access

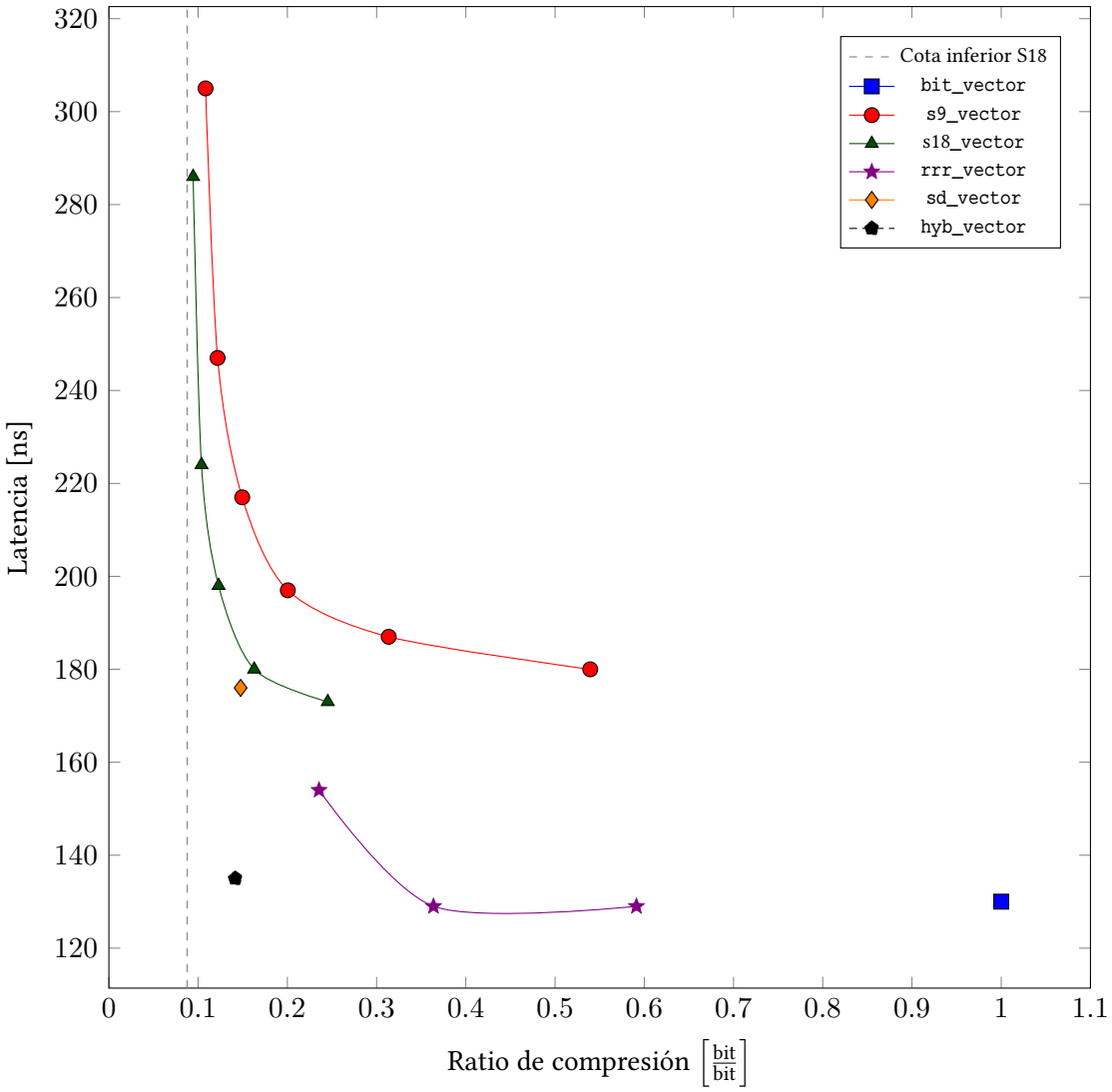


Figura B.1: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.01$. Los resutados completos se encuentran tabulados en la Tabla B.1.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	130	0.1512
s9_vector	8	180	0.0815
s9_vector	16	187	0.0474
s9_vector	32	197	0.0303
s9_vector	64	217	0.0226
s9_vector	128	247	0.0184
s9_vector	256	305	0.0164
s9_vector	512	403	0.0154
s18_vector	1	173	0.0371
s18_vector	2	180	0.0246
s18_vector	4	198	0.0186
s18_vector	8	224	0.0157
s18_vector	16	286	0.0143
s18_vector	32	442	0.0136
s18_vector	64	716	0.0133
rrr_vector	7	129	0.0894
rrr_vector	15	129	0.0550
rrr_vector	31	154	0.0356
rrr_vector	63	171	0.0249
rrr_vector	127	197	0.0196
rrr_vector	255	262	0.0175
sd_vector	-	176	0.0223
hyb_vector	-	135	0.0214

Tabla B.1: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.01$.

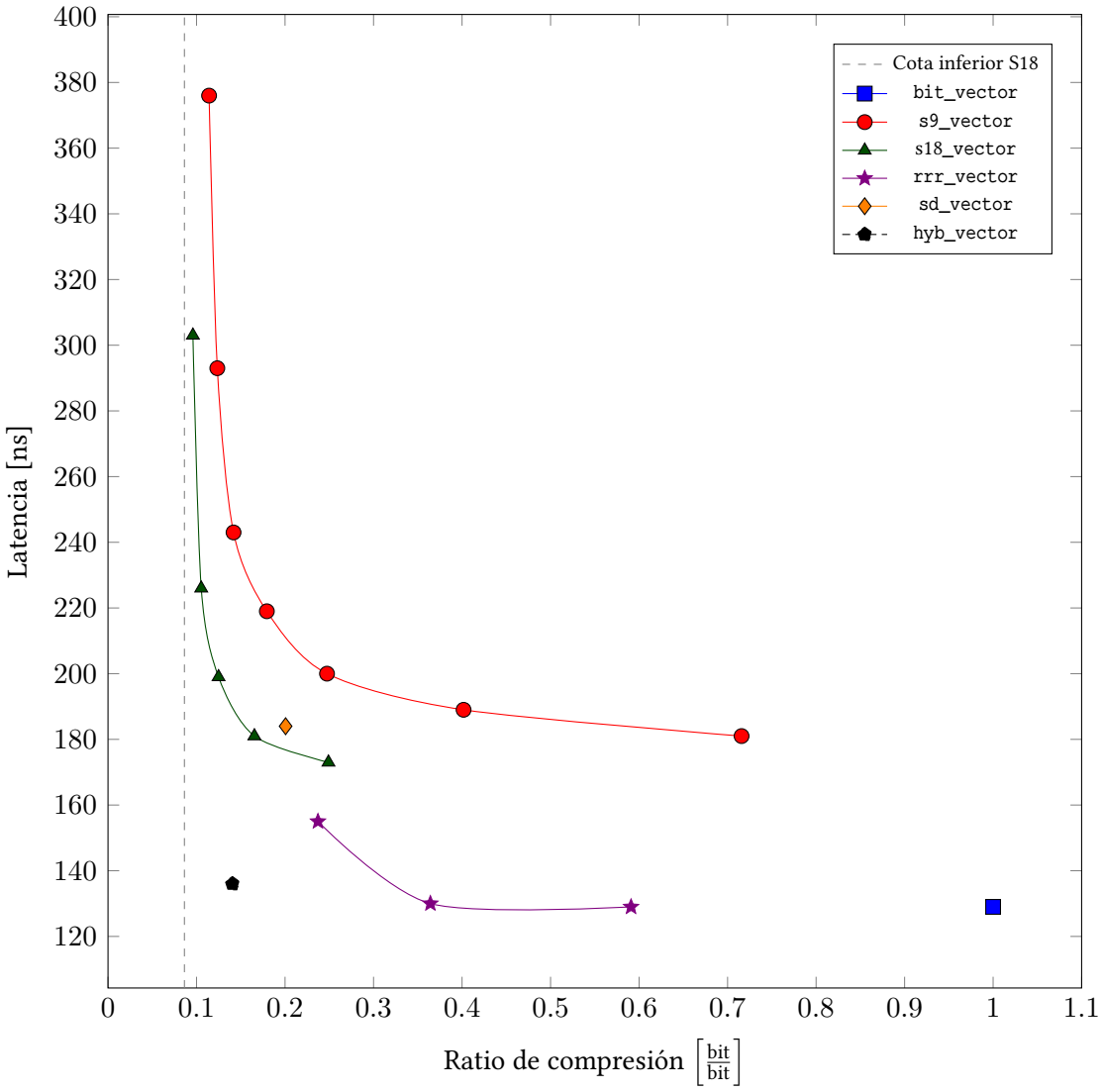


Figura B.2: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.02$. Los resutados completos se encuentran tabulados en la Tabla B.2.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	129	0.1532
s9_vector	8	181	0.1096
s9_vector	16	189	0.0615
s9_vector	32	200	0.0379
s9_vector	64	219	0.0275
s9_vector	128	243	0.0217
s9_vector	256	293	0.0189
s9_vector	512	376	0.0175
s18_vector	1	173	0.0382
s18_vector	2	181	0.0253
s18_vector	4	199	0.0191
s18_vector	8	226	0.0161
s18_vector	16	303	0.0147
s18_vector	32	434	0.0140
s18_vector	64	739	0.0137
rrr_vector	7	129	0.0905
rrr_vector	15	130	0.0558
rrr_vector	31	155	0.0363
rrr_vector	63	172	0.0260
rrr_vector	127	197	0.0218
rrr_vector	255	266	0.0205
sd_vector	-	184	0.0307
hyb_vector	-	136	0.0215

Tabla B.2: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.02$.

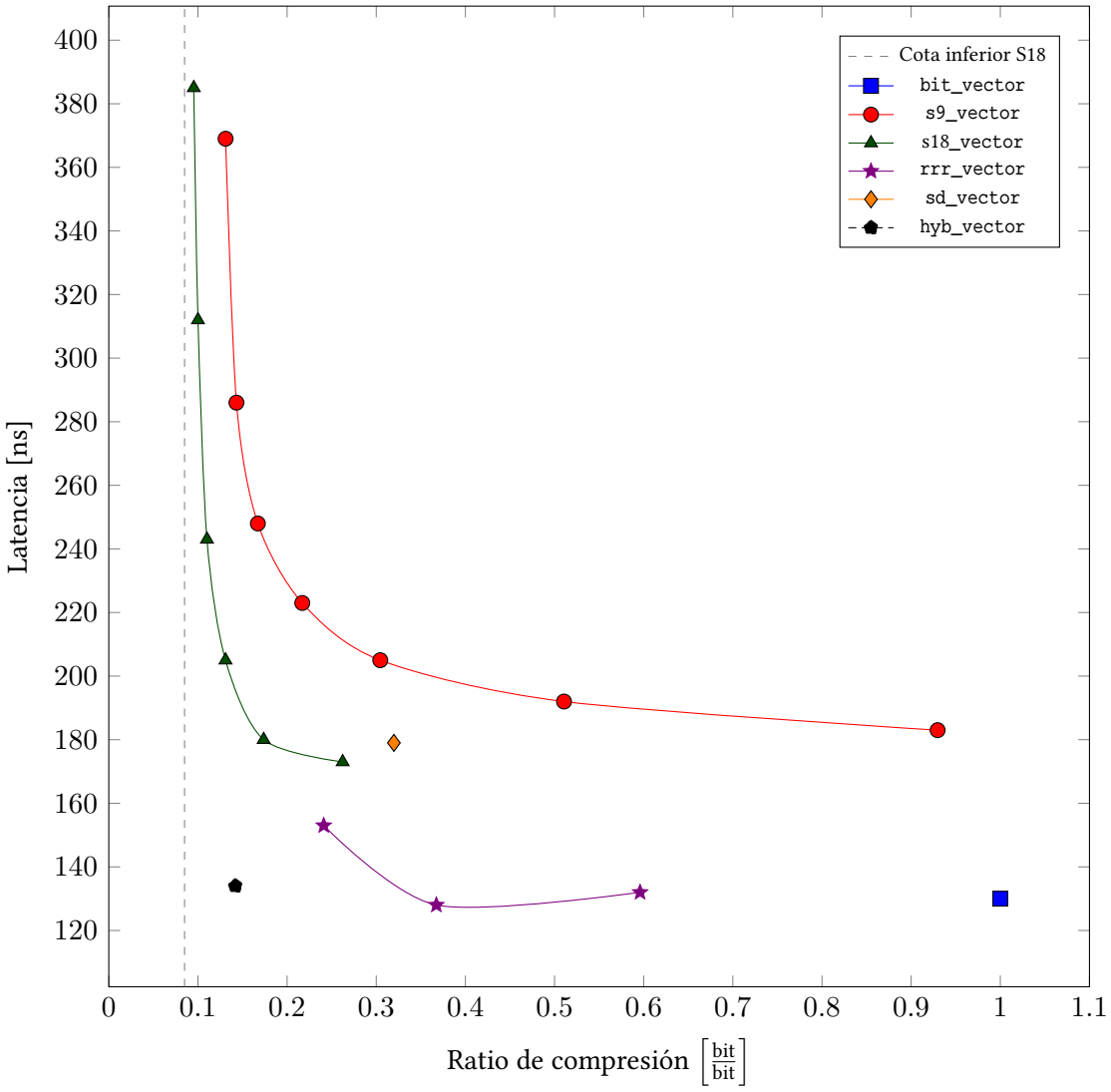


Figura B.3: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.03$. Los resutados completos se encuentran tabulados en la Tabla B.3.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	130	0.1506
s9_vector	8	183	0.1400
s9_vector	16	192	0.0769
s9_vector	32	205	0.0459
s9_vector	64	223	0.0327
s9_vector	128	248	0.0252
s9_vector	256	286	0.0216
s9_vector	512	369	0.0197
s18_vector	1	173	0.0395
s18_vector	2	180	0.0262
s18_vector	4	205	0.0197
s18_vector	8	243	0.0166
s18_vector	16	312	0.0151
s18_vector	32	385	0.0144
s18_vector	64	841	0.0140
rrr_vector	7	132	0.0897
rrr_vector	15	128	0.0553
rrr_vector	31	153	0.0363
rrr_vector	63	171	0.0265
rrr_vector	127	195	0.0232
rrr_vector	255	261	0.0233
sd_vector	-	179	0.0482
hyb_vector	-	134	0.0214

Tabla B.3: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.03$.

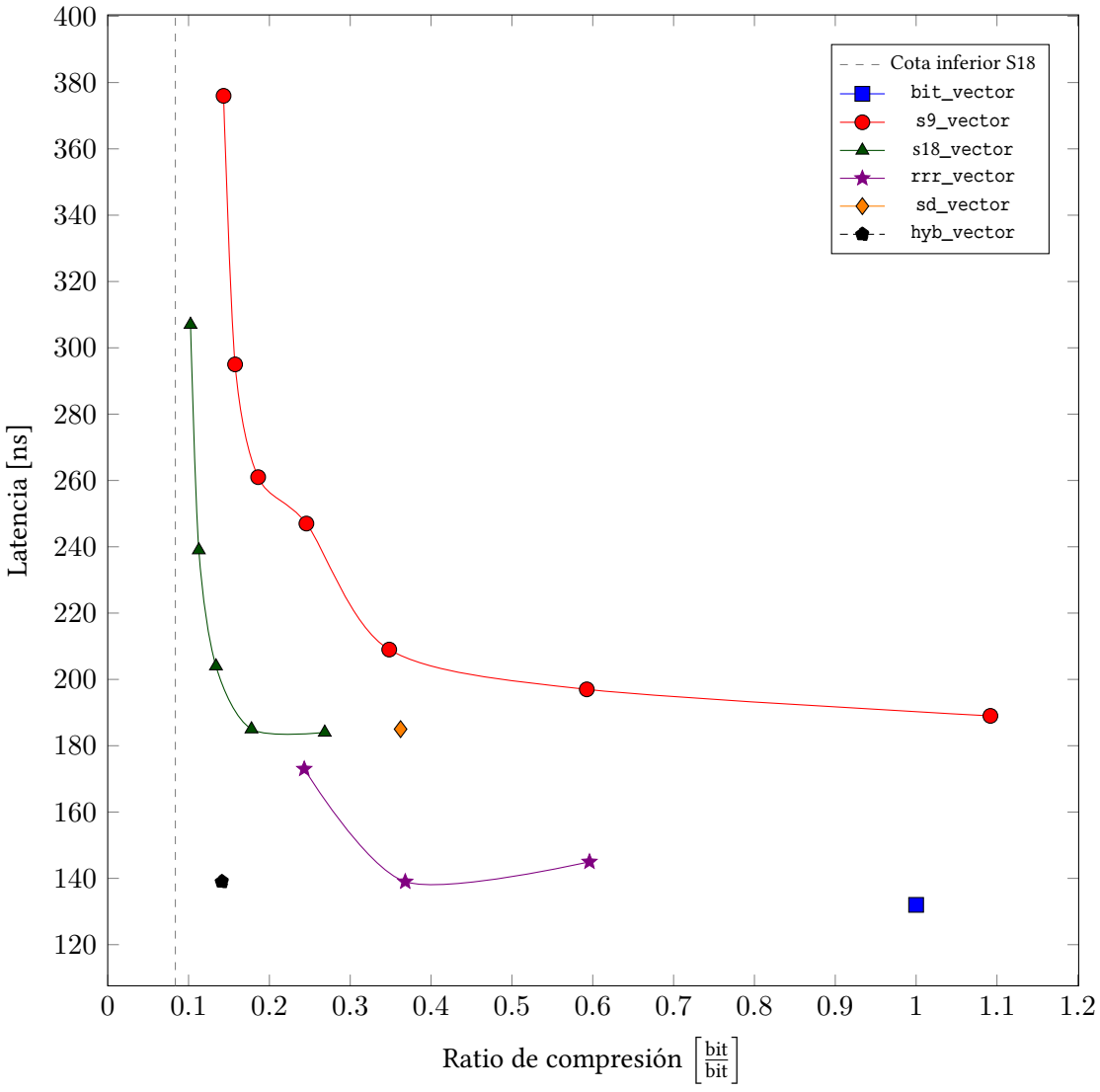


Figura B.4: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.04$. Los resutados completos se encuentran tabulados en la Tabla B.4.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	132	0.1509
s9_vector	8	189	0.1648
s9_vector	16	197	0.0894
s9_vector	32	209	0.0526
s9_vector	64	247	0.0371
s9_vector	128	261	0.0281
s9_vector	256	295	0.0238
s9_vector	512	376	0.0216
s18_vector	1	184	0.0405
s18_vector	2	185	0.0268
s18_vector	4	204	0.0202
s18_vector	8	239	0.0170
s18_vector	16	307	0.0155
s18_vector	32	432	0.0147
s18_vector	64	708	0.0144
rrr_vector	7	145	0.0900
rrr_vector	15	139	0.0556
rrr_vector	31	173	0.0367
rrr_vector	63	190	0.0273
rrr_vector	127	212	0.0247
rrr_vector	255	272	0.0257
sd_vector	-	185	0.0547
hyb_vector	-	139	0.0213

Tabla B.4: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.04$.

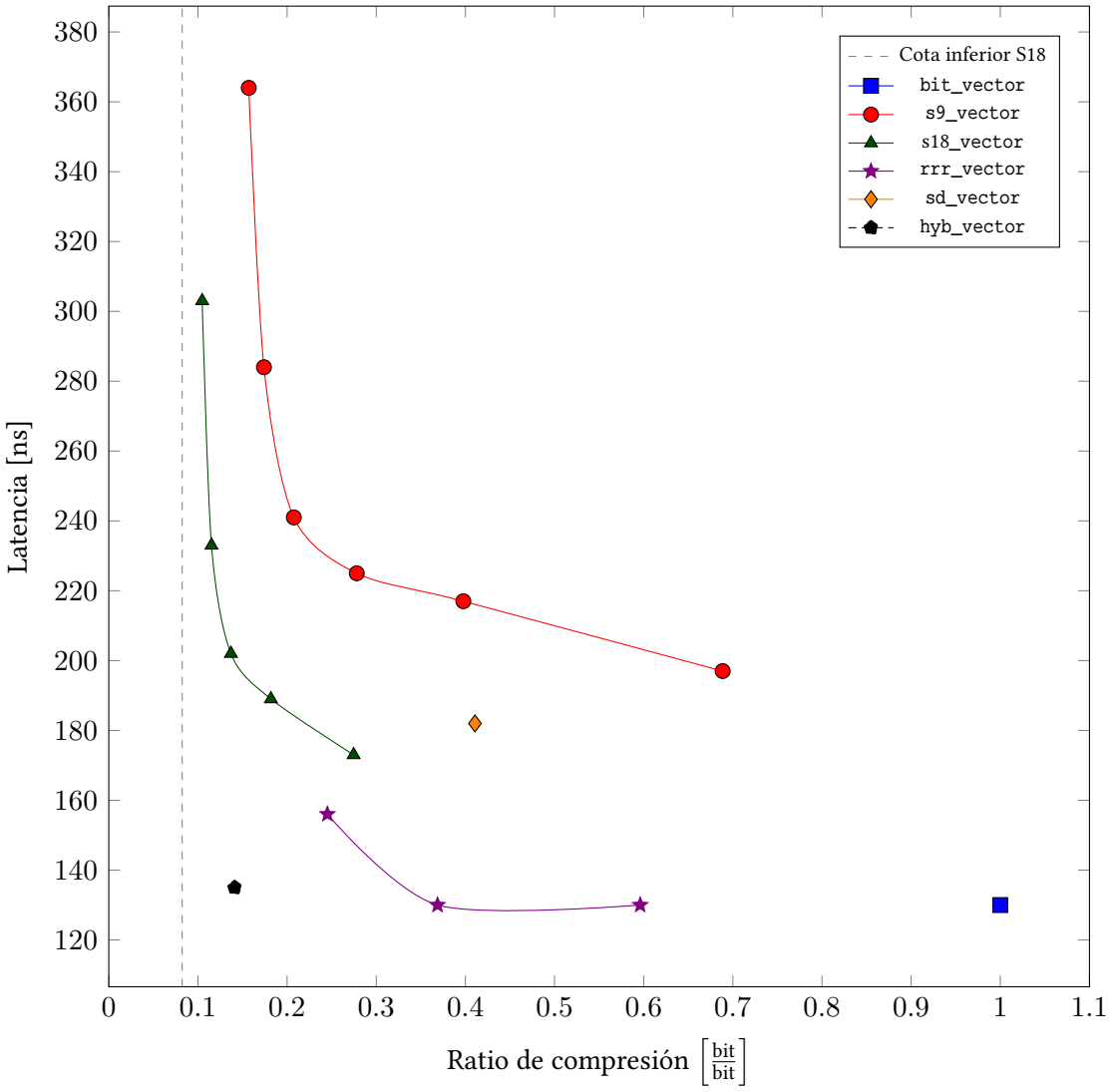


Figura B.5: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.05$. Los resutados completos se encuentran tabulados en la Tabla B.5.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	130	0.1514
s9_vector	8	188	0.1939
s9_vector	16	197	0.1042
s9_vector	32	217	0.0602
s9_vector	64	225	0.0421
s9_vector	128	241	0.0314
s9_vector	256	284	0.0263
s9_vector	512	364	0.0238
s18_vector	1	173	0.0416
s18_vector	2	189	0.0275
s18_vector	4	202	0.0207
s18_vector	8	233	0.0174
s18_vector	16	303	0.0159
s18_vector	32	446	0.0151
s18_vector	64	764	0.0147
rrr_vector	7	130	0.0902
rrr_vector	15	130	0.0558
rrr_vector	31	156	0.0371
rrr_vector	63	173	0.0281
rrr_vector	127	197	0.0267
rrr_vector	255	259	0.0286
sd_vector	-	182	0.0622
hyb_vector	-	135	0.0214

Tabla B.5: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.05$.

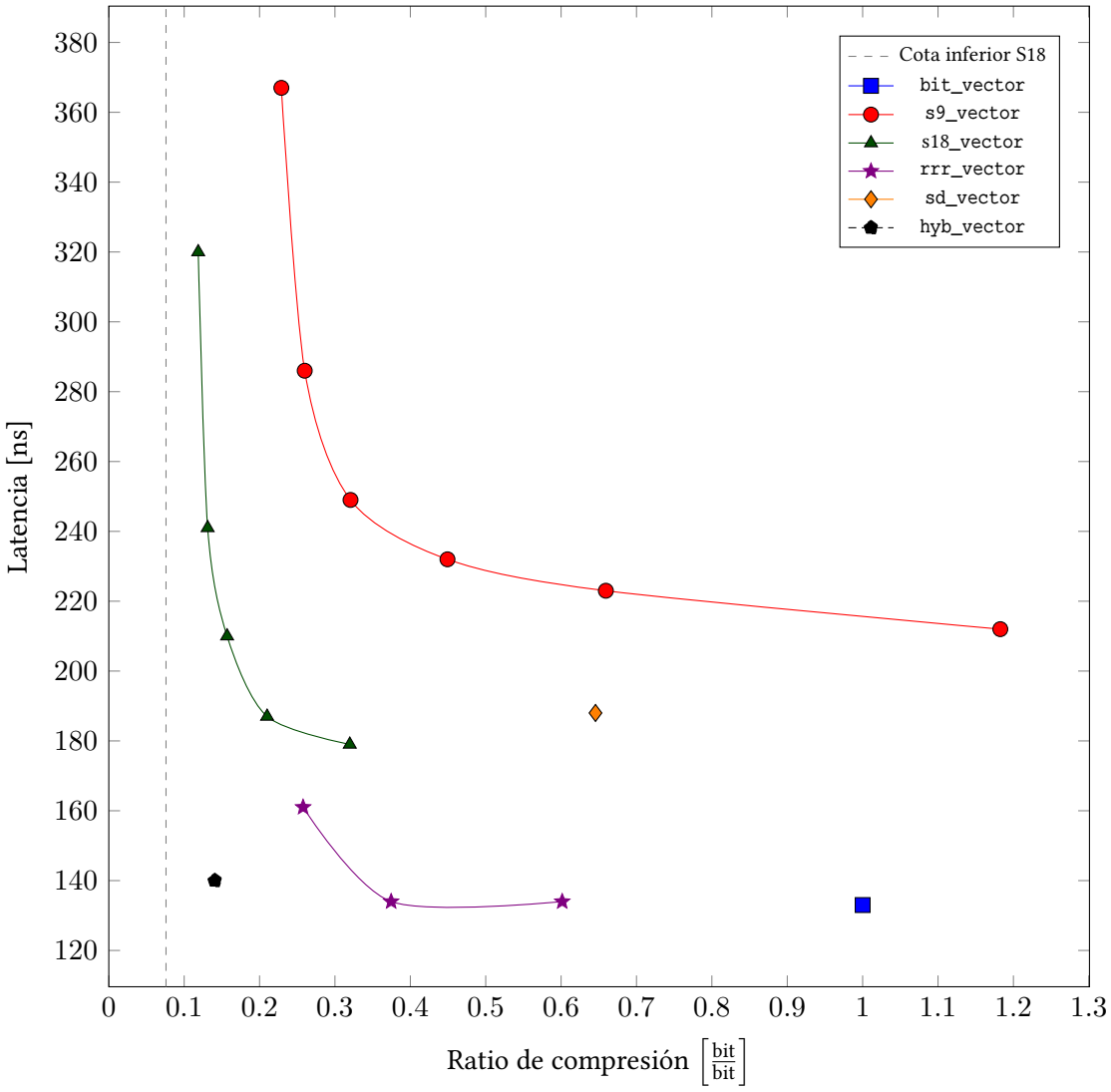


Figura B.6: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.1$. Los resutados completos se encuentran tabulados en la Tabla B.6.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	133	0.1495
s9_vector	8	208	0.3375
s9_vector	16	212	0.1768
s9_vector	32	223	0.0986
s9_vector	64	232	0.0672
s9_vector	128	249	0.0479
s9_vector	256	286	0.0389
s9_vector	512	367	0.0342
s18_vector	1	179	0.0478
s18_vector	2	187	0.0314
s18_vector	4	210	0.0235
s18_vector	8	241	0.0196
s18_vector	16	320	0.0177
s18_vector	32	404	0.0168
s18_vector	64	750	0.0164
rrr_vector	7	134	0.0899
rrr_vector	15	134	0.0560
rrr_vector	31	161	0.0385
rrr_vector	63	178	0.0316
rrr_vector	127	203	0.0345
rrr_vector	255	266	0.0412
sd_vector	-	188	0.0965
hyb_vector	-	140	0.0210

Tabla B.6: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.1$.

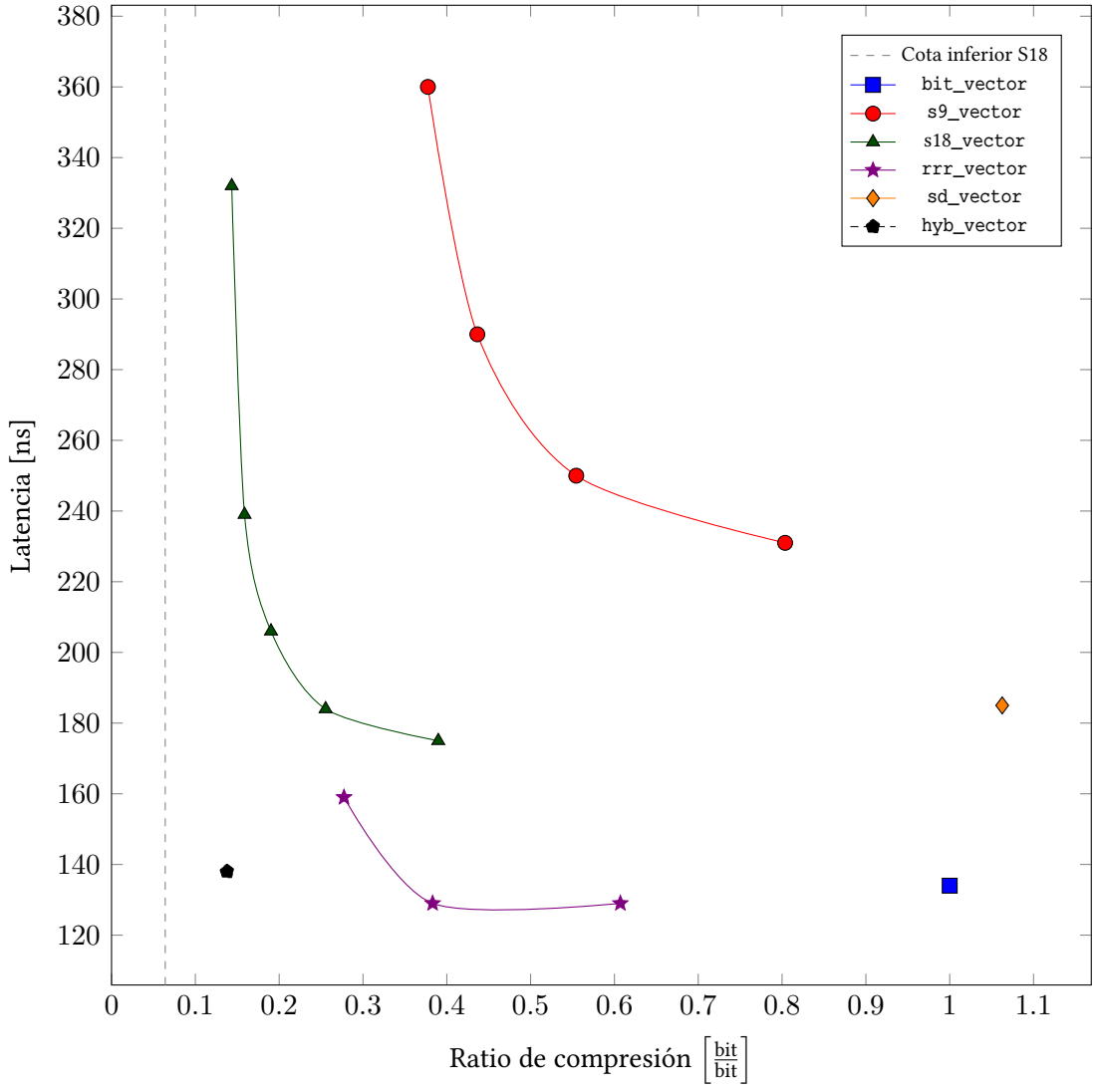


Figura B.7: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.2$. Los resutados completos se encuentran tabulados en la Tabla B.7.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	134	0.1454
s9_vector	8	218	0.6279
s9_vector	16	221	0.3231
s9_vector	32	224	0.1748
s9_vector	64	231	0.1169
s9_vector	128	250	0.0806
s9_vector	256	290	0.0635
s9_vector	512	360	0.0549
s18_vector	1	175	0.0567
s18_vector	2	184	0.0371
s18_vector	4	206	0.0277
s18_vector	8	239	0.0231
s18_vector	16	332	0.0209
s18_vector	32	434	0.0198
s18_vector	64	779	0.0193
rrr_vector	7	129	0.0883
rrr_vector	15	129	0.0557
rrr_vector	31	159	0.0403
rrr_vector	63	175	0.0373
rrr_vector	127	198	0.0474
rrr_vector	255	255	0.0626
sd_vector	-	185	0.1545
hyb_vector	-	138	0.0200

Tabla B.7: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.2$.

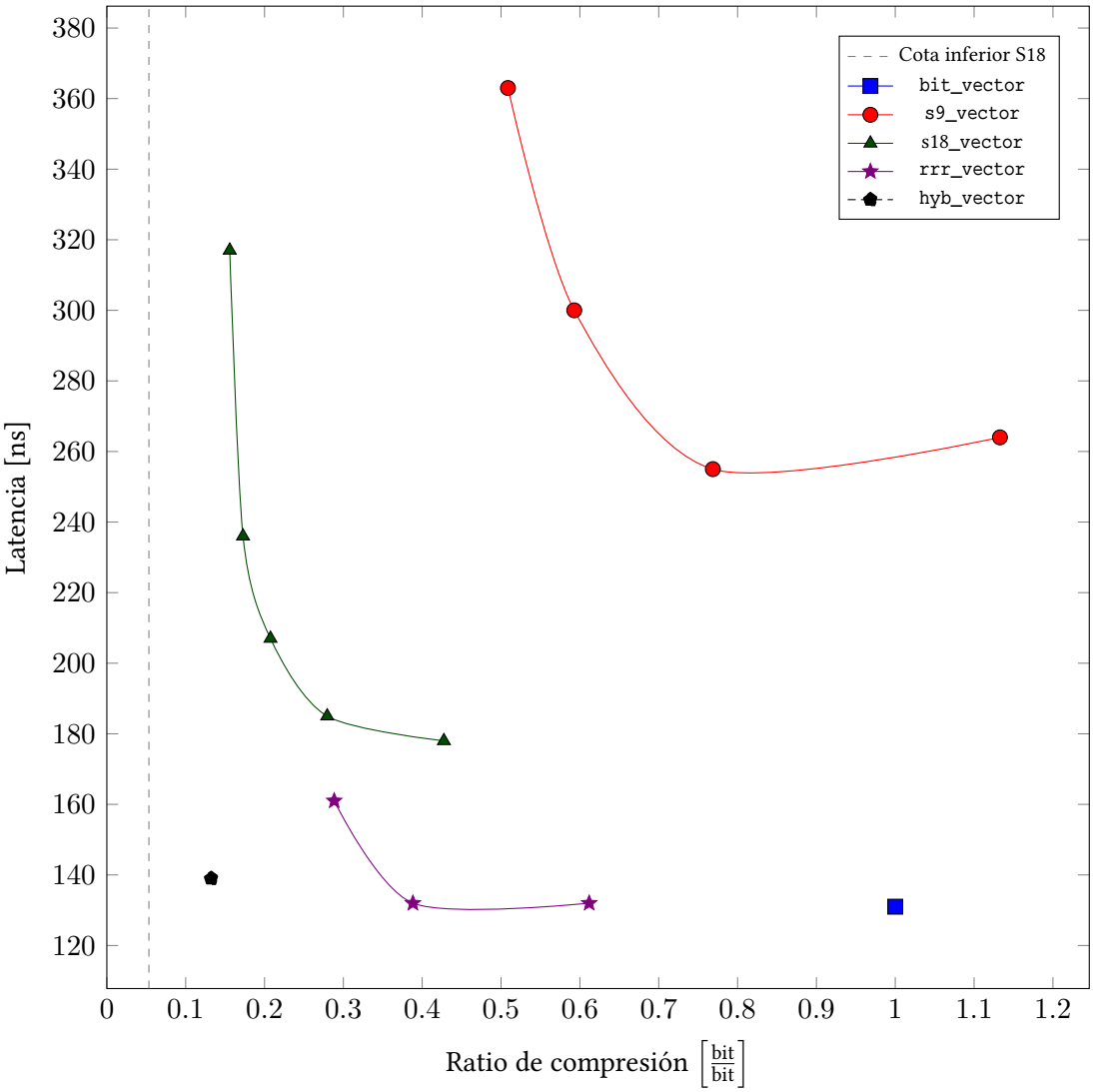


Figura B.8: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.3$. Los resutados completos se encuentran tabulados en la Tabla B.8.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	131	0.1424
s9_vector	8	227	0.8921
s9_vector	16	252	0.4563
s9_vector	32	246	0.2431
s9_vector	64	264	0.1614
s9_vector	128	255	0.1095
s9_vector	256	300	0.0845
s9_vector	512	363	0.0725
s18_vector	1	178	0.0609
s18_vector	2	185	0.0398
s18_vector	4	207	0.0296
s18_vector	8	236	0.0246
s18_vector	16	317	0.0222
s18_vector	32	471	0.0210
s18_vector	64	739	0.0205
rrr_vector	7	132	0.0871
rrr_vector	15	132	0.0553
rrr_vector	31	161	0.0411
rrr_vector	63	191	0.0406
rrr_vector	127	203	0.0540
rrr_vector	255	247	0.0752
sd_vector	-	187	0.2040
hyb_vector	-	139	0.0189

Tabla B.8: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.3$.

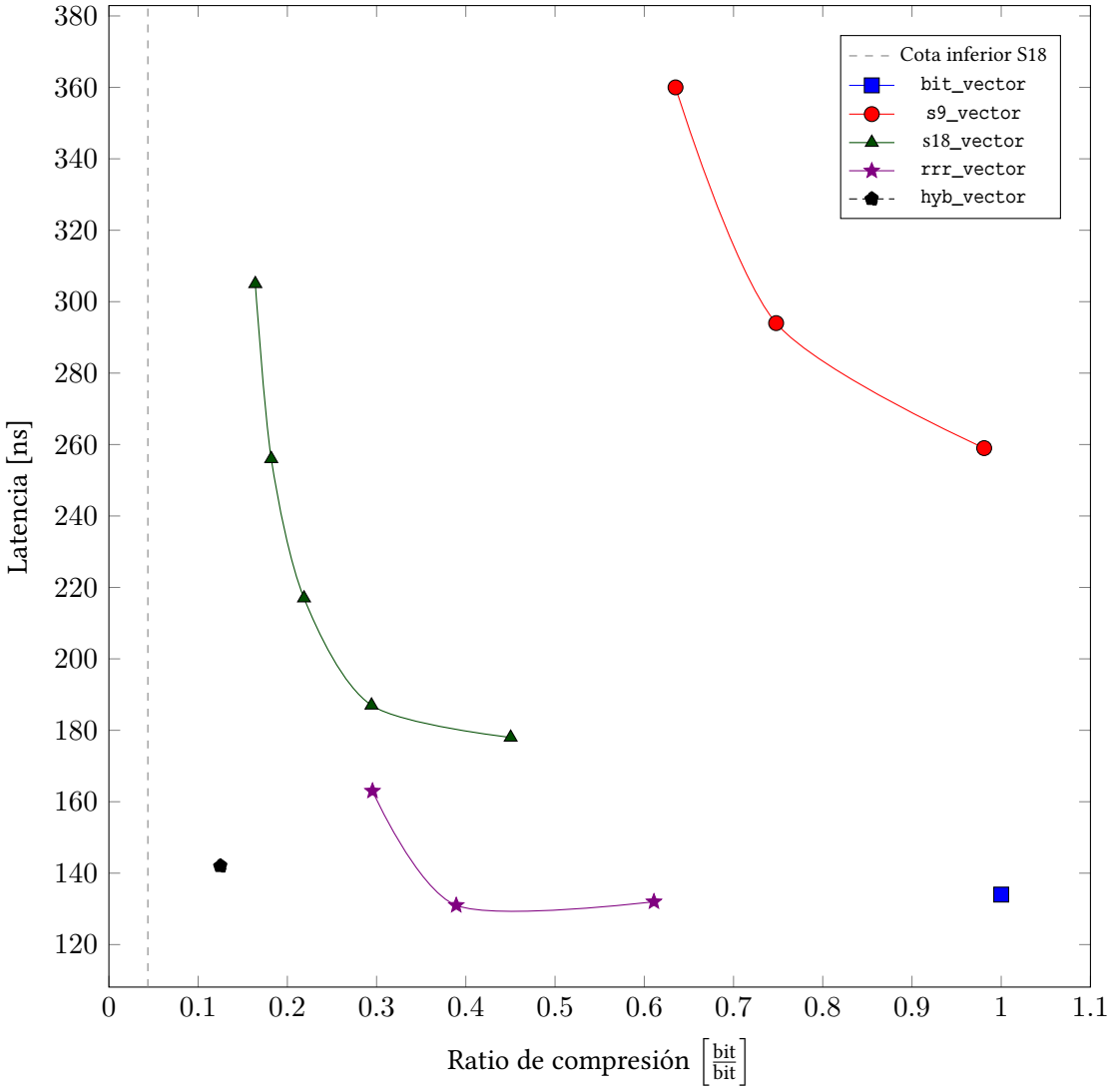


Figura B.9: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.4$. Los resutados completos se encuentran tabulados en la Tabla B.9.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	134	0.1409
s9_vector	8	239	1.1595
s9_vector	16	237	0.5902
s9_vector	32	244	0.3110
s9_vector	64	254	0.2060
s9_vector	128	259	0.1382
s9_vector	256	294	0.1054
s9_vector	512	360	0.0895
s18_vector	1	178	0.0634
s18_vector	2	187	0.0415
s18_vector	4	217	0.0308
s18_vector	8	256	0.0256
s18_vector	16	305	0.0231
s18_vector	32	401	0.0219
s18_vector	64	690	0.0213
rrr_vector	7	132	0.0861
rrr_vector	15	131	0.0549
rrr_vector	31	163	0.0416
rrr_vector	63	176	0.0423
rrr_vector	127	197	0.0593
rrr_vector	255	236	0.0844
sd_vector	-	188	0.2401
hyb_vector	-	142	0.0176

Tabla B.9: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.4$.

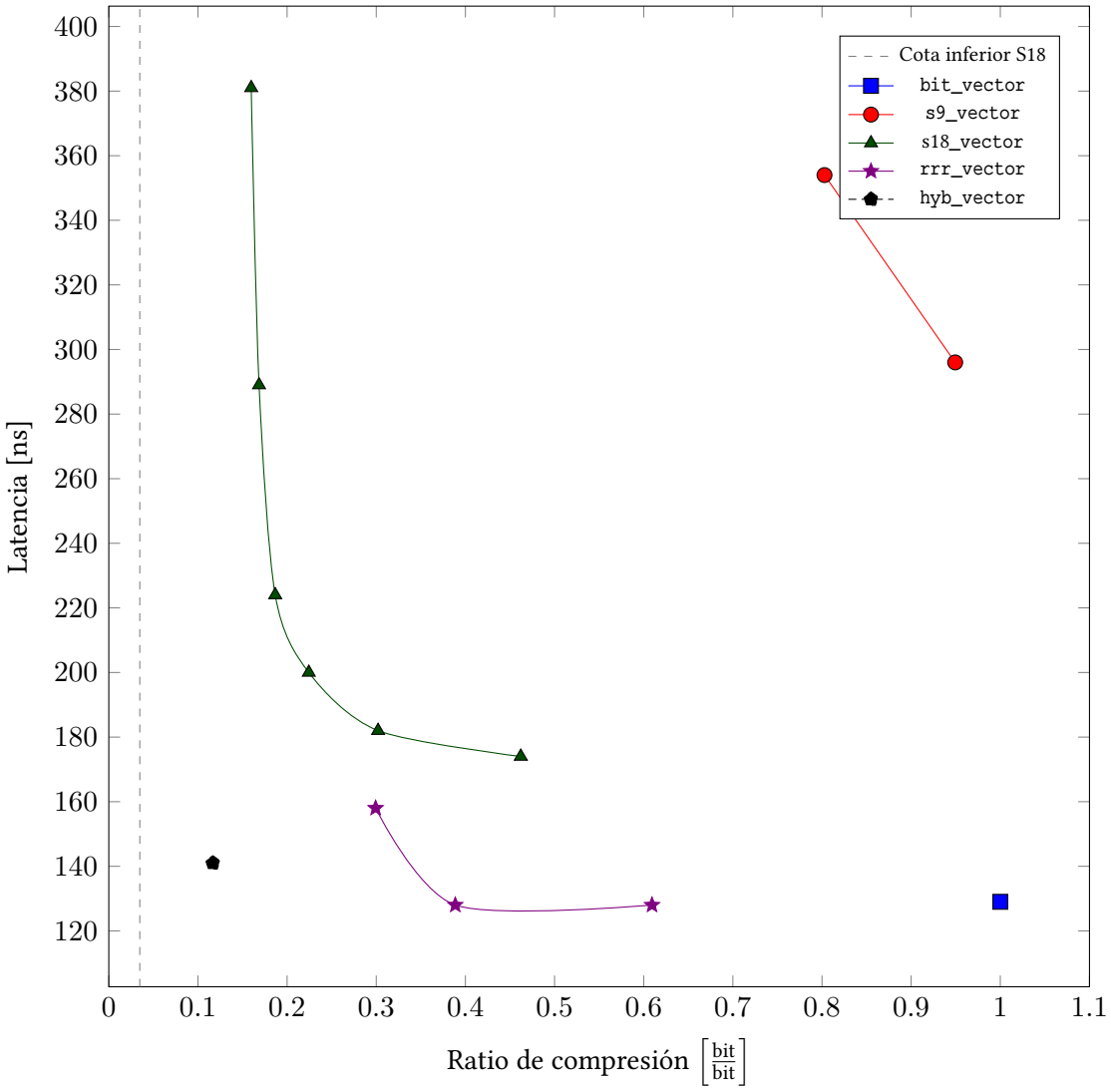


Figura B.10: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.5$. Los resutados completos se encuentran tabulados en la Tabla B.10.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	129	0.1354
s9_vector	8	239	1.4722
s9_vector	16	240	0.7466
s9_vector	32	244	0.3894
s9_vector	64	248	0.2578
s9_vector	128	263	0.1705
s9_vector	256	296	0.1286
s9_vector	512	354	0.1087
s18_vector	1	174	0.0626
s18_vector	2	182	0.0409
s18_vector	4	200	0.0304
s18_vector	8	224	0.0253
s18_vector	16	289	0.0228
s18_vector	32	381	0.0216
s18_vector	64	590	0.0210
rrr_vector	7	128	0.0825
rrr_vector	15	128	0.0526
rrr_vector	31	158	0.0405
rrr_vector	63	171	0.0420
rrr_vector	127	191	0.0593
rrr_vector	255	223	0.0880
sd_vector	-	187	0.2821
hyb_vector	-	141	0.0158

Tabla B.10: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.5$.

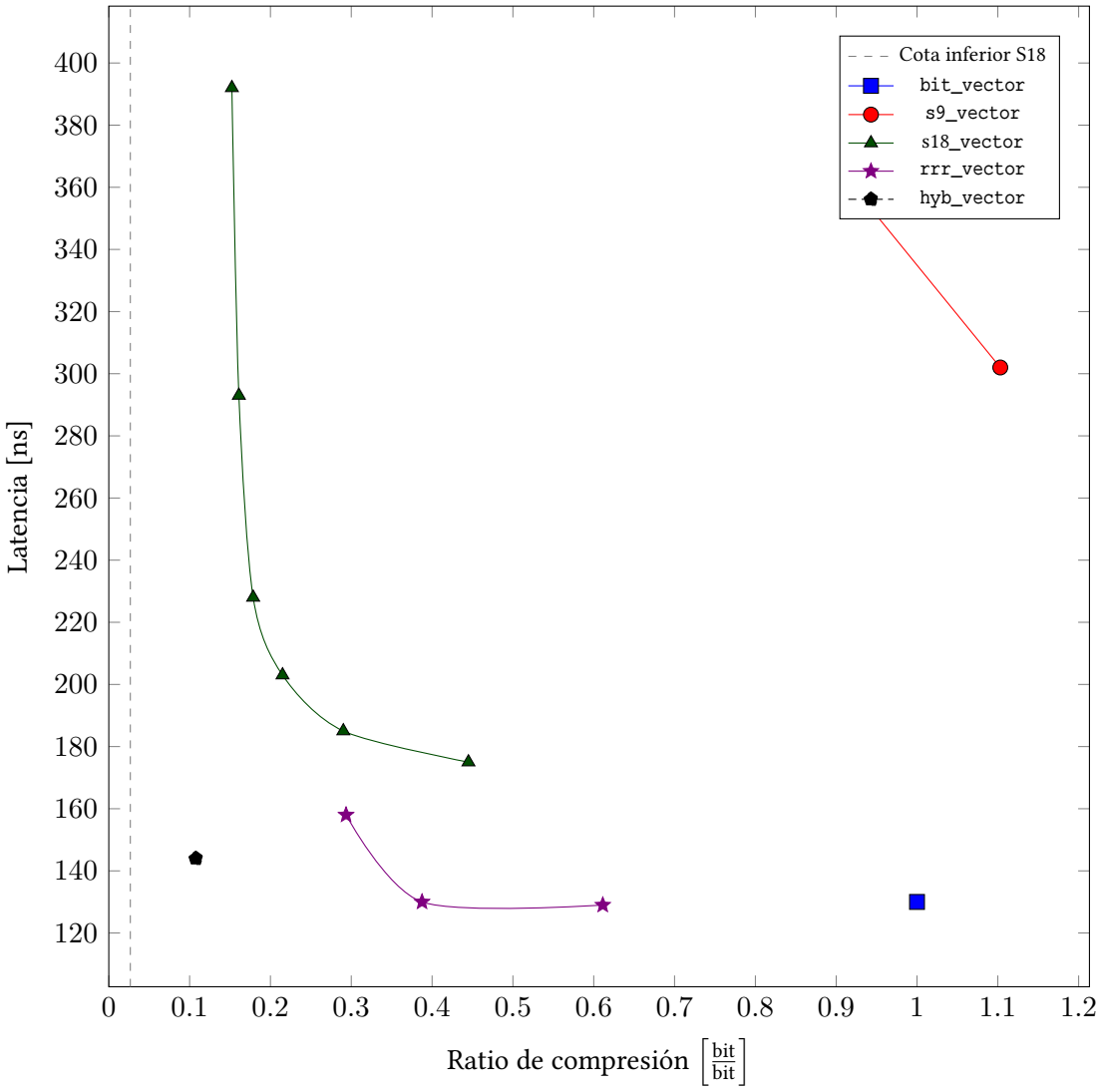


Figura B.11: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.6$. Los resutados completos se encuentran tabulados en la Tabla B.11.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	130	0.1333
s9_vector	8	246	1.7407
s9_vector	16	248	0.8799
s9_vector	32	253	0.4550
s9_vector	64	257	0.3014
s9_vector	128	270	0.1970
s9_vector	256	302	0.1470
s9_vector	512	357	0.1241
s18_vector	1	175	0.0593
s18_vector	2	185	0.0387
s18_vector	4	203	0.0287
s18_vector	8	228	0.0238
s18_vector	16	293	0.0215
s18_vector	32	392	0.0203
s18_vector	64	605	0.0197
rrr_vector	7	129	0.0815
rrr_vector	15	130	0.0517
rrr_vector	31	158	0.0391
rrr_vector	63	169	0.0406
rrr_vector	127	188	0.0563
rrr_vector	255	213	0.0837
sd_vector	-	186	0.3243
hyb_vector	-	144	0.0144

Tabla B.11: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.6$.

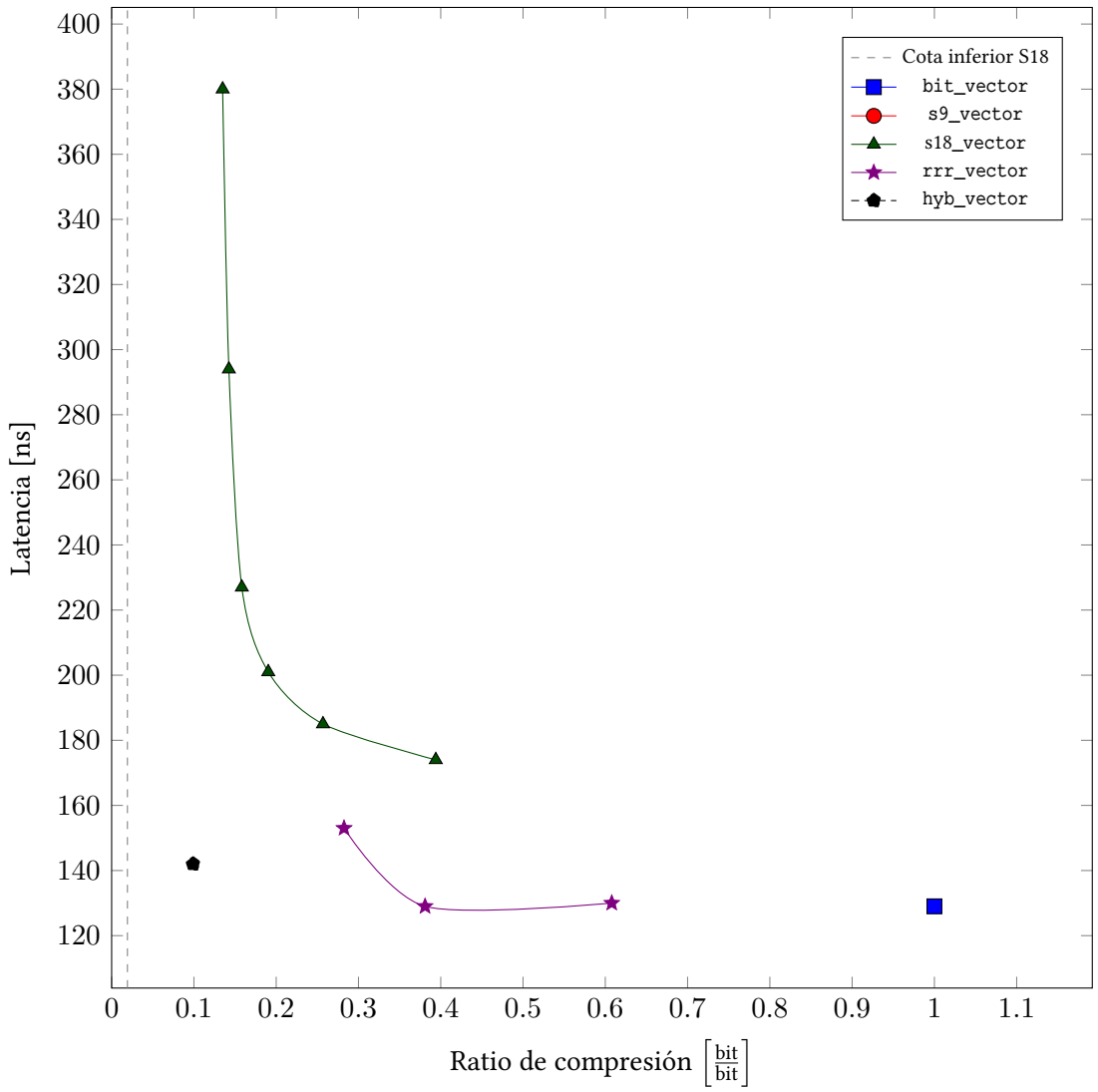


Figura B.12: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.7$. Los resutados completos se encuentran tabulados en la Tabla B.12.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	129	0.1288
s9_vector	8	251	2.0358
s9_vector	16	251	1.0258
s9_vector	32	262	0.5256
s9_vector	64	263	0.3485
s9_vector	128	275	0.2253
s9_vector	256	310	0.1653
s9_vector	512	363	0.1396
s18_vector	1	174	0.0508
s18_vector	2	185	0.0331
s18_vector	4	201	0.0245
s18_vector	8	227	0.0204
s18_vector	16	294	0.0184
s18_vector	32	380	0.0174
s18_vector	64	624	0.0169
rrr_vector	7	130	0.0783
rrr_vector	15	129	0.0491
rrr_vector	31	153	0.0364
rrr_vector	63	164	0.0364
rrr_vector	127	182	0.0499
rrr_vector	255	200	0.0729
sd_vector	-	186	0.3518
hyb_vector	-	142	0.0128

Tabla B.12: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.7$.

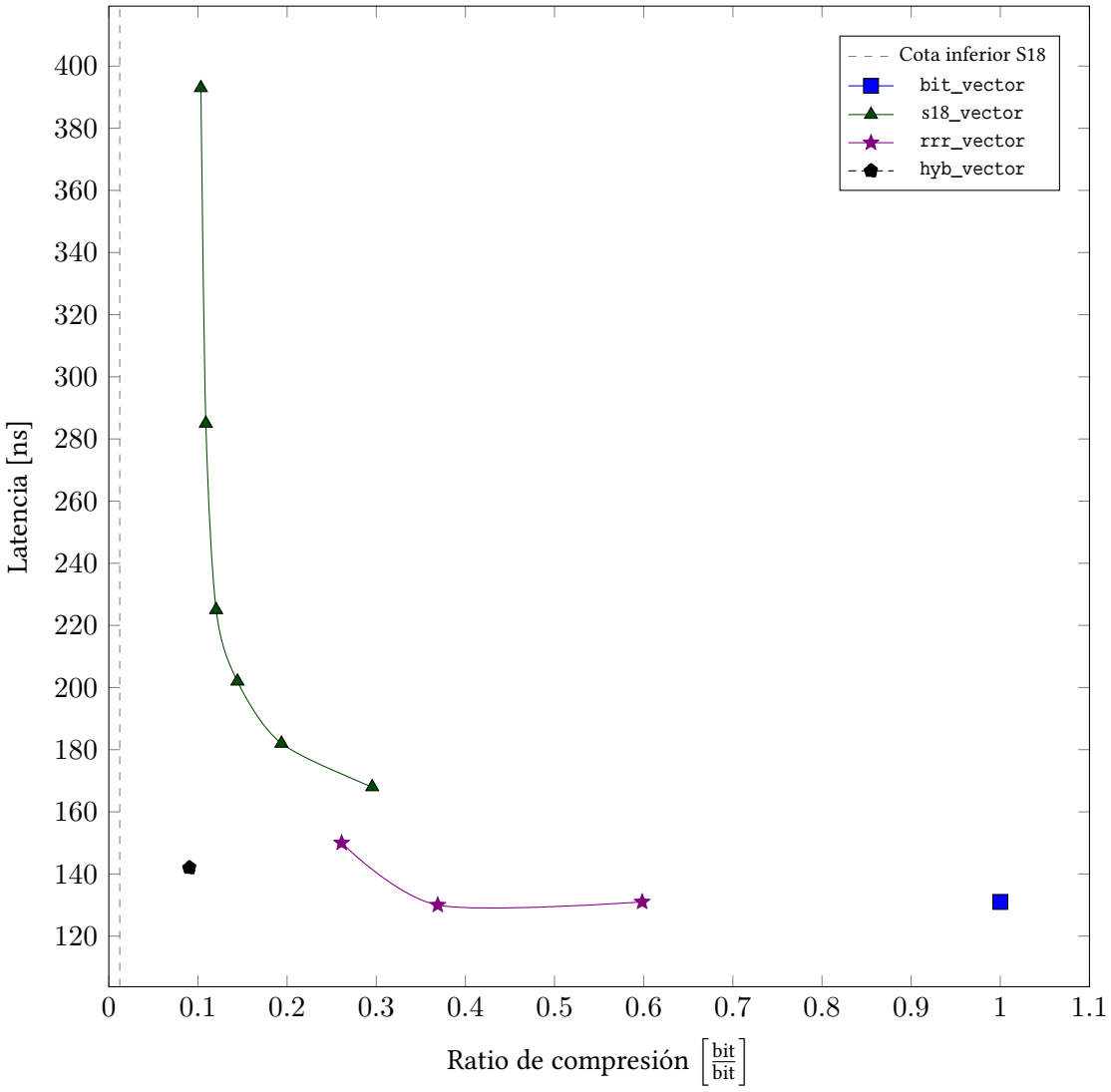


Figura B.13: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.8$. Los resutados completos se encuentran tabulados en la Tabla B.13.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	131	0.1252
s9_vector	8	260	2.3285
s9_vector	16	259	1.1701
s9_vector	32	269	0.5944
s9_vector	64	272	0.3945
s9_vector	128	282	0.2521
s9_vector	256	312	0.1821
s9_vector	512	357	0.1538
s18_vector	1	168	0.0370
s18_vector	2	182	0.0243
s18_vector	4	202	0.0181
s18_vector	8	225	0.0151
s18_vector	16	285	0.0136
s18_vector	32	393	0.0129
s18_vector	64	629	0.0126
rrr_vector	7	131	0.0749
rrr_vector	15	130	0.0462
rrr_vector	31	150	0.0327
rrr_vector	63	159	0.0303
rrr_vector	127	170	0.0385
rrr_vector	255	195	0.0561
sd_vector	-	188	0.3790
hyb_vector	-	142	0.0113

Tabla B.13: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.8$.

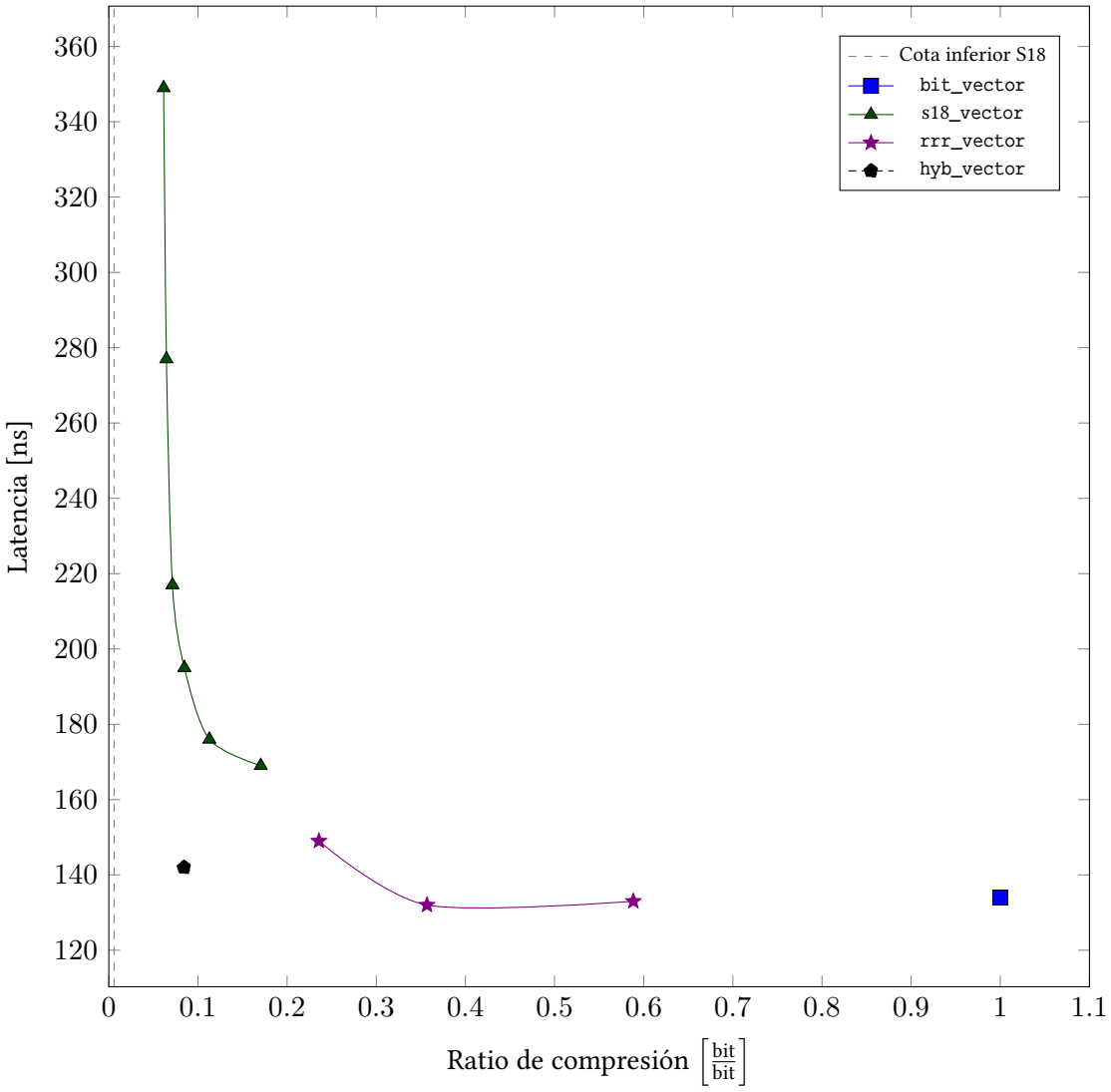


Figura B.14: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.9$. Los resutados completos se encuentran tabulados en la Tabla B.14.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	134	0.1227
s9_vector	8	268	2.5763
s9_vector	16	266	1.2914
s9_vector	32	279	0.6509
s9_vector	64	280	0.4328
s9_vector	128	290	0.2735
s9_vector	256	314	0.1948
s9_vector	512	358	0.1652
s18_vector	1	169	0.0209
s18_vector	2	176	0.0138
s18_vector	4	195	0.0104
s18_vector	8	217	0.0088
s18_vector	16	277	0.0079
s18_vector	32	349	0.0075
s18_vector	64	538	0.0074
rrr_vector	7	133	0.0722
rrr_vector	15	132	0.0438
rrr_vector	31	149	0.0289
rrr_vector	63	153	0.0231
rrr_vector	127	161	0.0256
rrr_vector	255	173	0.0344
sd_vector	-	191	0.3254
hyb_vector	-	142	0.0103

Tabla B.14: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.9$.

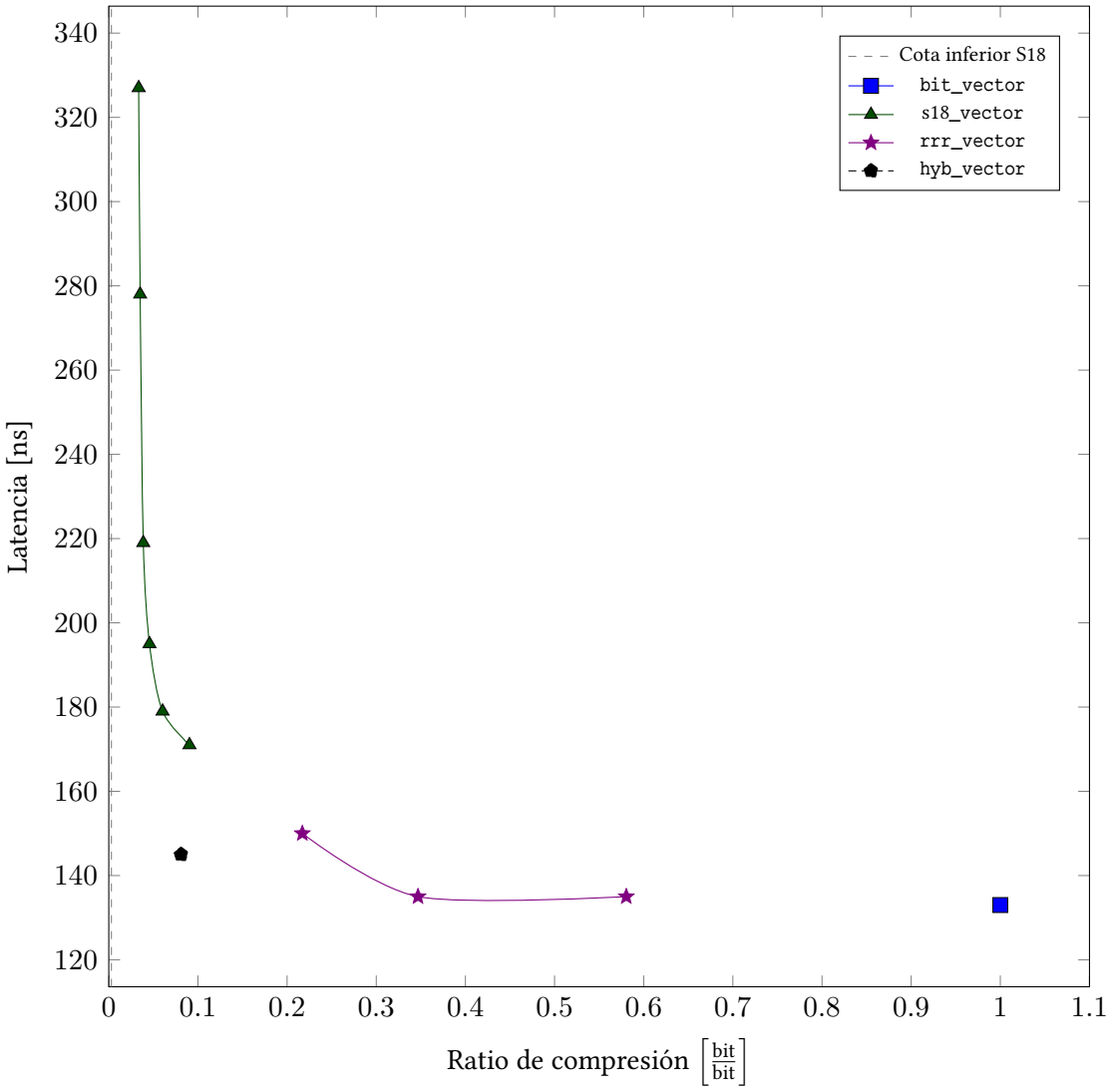


Figura B.15: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.95$. Los resutados completos se encuentran tabulados en la Tabla B.15.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	133	0.1208
s9_vector	8	277	2.7266
s9_vector	16	277	1.3651
s9_vector	32	287	0.6854
s9_vector	64	291	0.4562
s9_vector	128	301	0.2867
s9_vector	256	324	0.2023
s9_vector	512	368	0.1723
s18_vector	1	171	0.0109
s18_vector	2	179	0.0073
s18_vector	4	195	0.0055
s18_vector	8	219	0.0047
s18_vector	16	278	0.0042
s18_vector	32	327	0.0040
s18_vector	64	608	0.0040
rrr_vector	7	135	0.0701
rrr_vector	15	135	0.0419
rrr_vector	31	150	0.0262
rrr_vector	63	151	0.0187
rrr_vector	127	156	0.0171
rrr_vector	255	160	0.0203
sd_vector	-	194	0.3394
hyb_vector	-	145	0.0098

Tabla B.15: Rendimiento de la operación Access. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.95$.

B.2. Rank

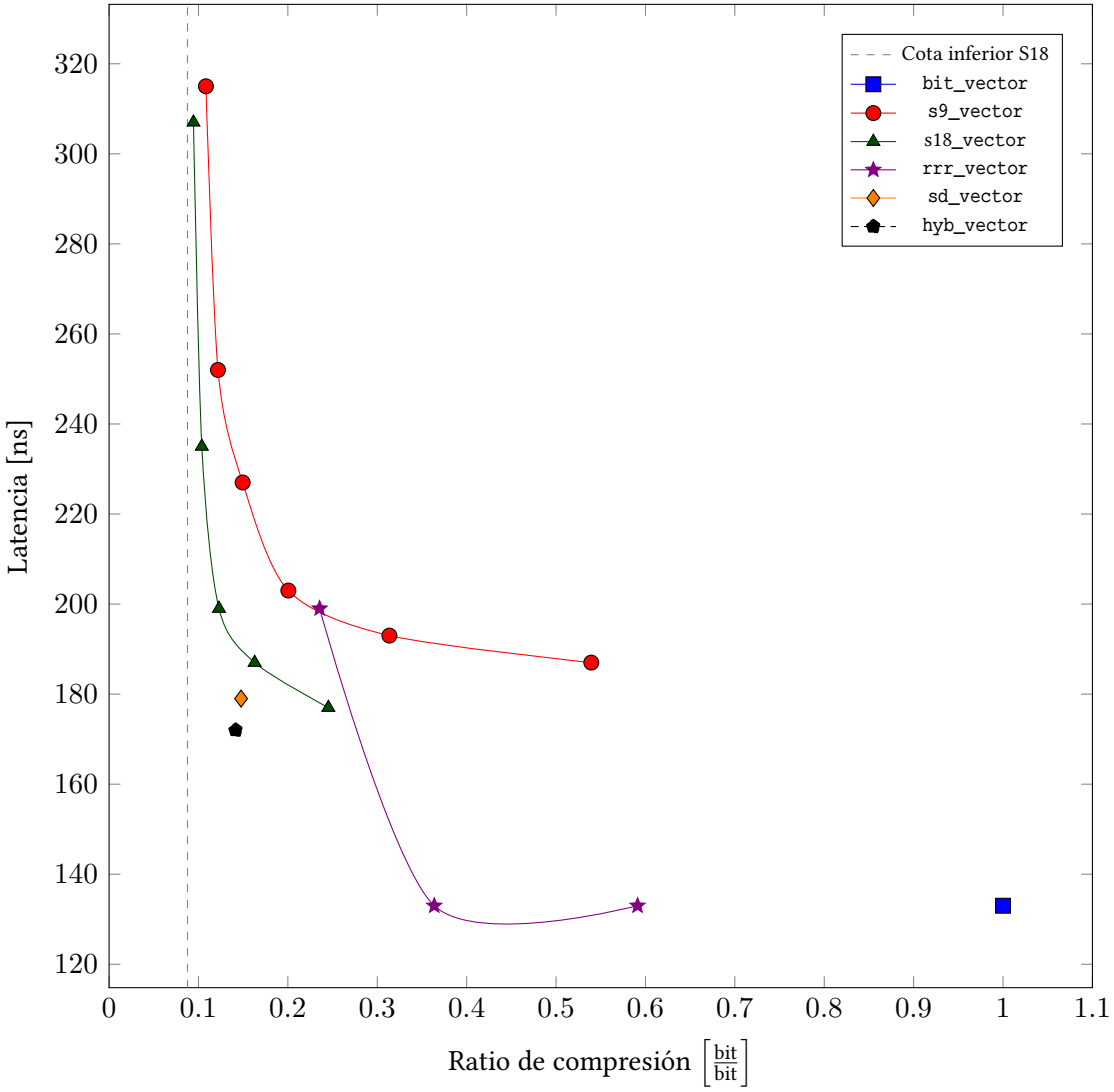


Figura B.16: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.01$. Los resutados completos se encuentran tabulados en la Tabla B.16.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	133	0.1512
s9_vector	8	187	0.0815
s9_vector	16	193	0.0474
s9_vector	32	203	0.0303
s9_vector	64	227	0.0226
s9_vector	128	252	0.0184
s9_vector	256	315	0.0164
s9_vector	512	401	0.0154
s18_vector	1	177	0.0371
s18_vector	2	187	0.0246
s18_vector	4	199	0.0186
s18_vector	8	235	0.0157
s18_vector	16	307	0.0143
s18_vector	32	445	0.0136
s18_vector	64	719	0.0133
rrr_vector	7	133	0.0894
rrr_vector	15	133	0.0550
rrr_vector	31	199	0.0356
rrr_vector	63	203	0.0249
rrr_vector	127	222	0.0196
rrr_vector	255	323	0.0175
sd_vector	-	179	0.0223
hyb_vector	-	172	0.0214

Tabla B.16: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.01$.

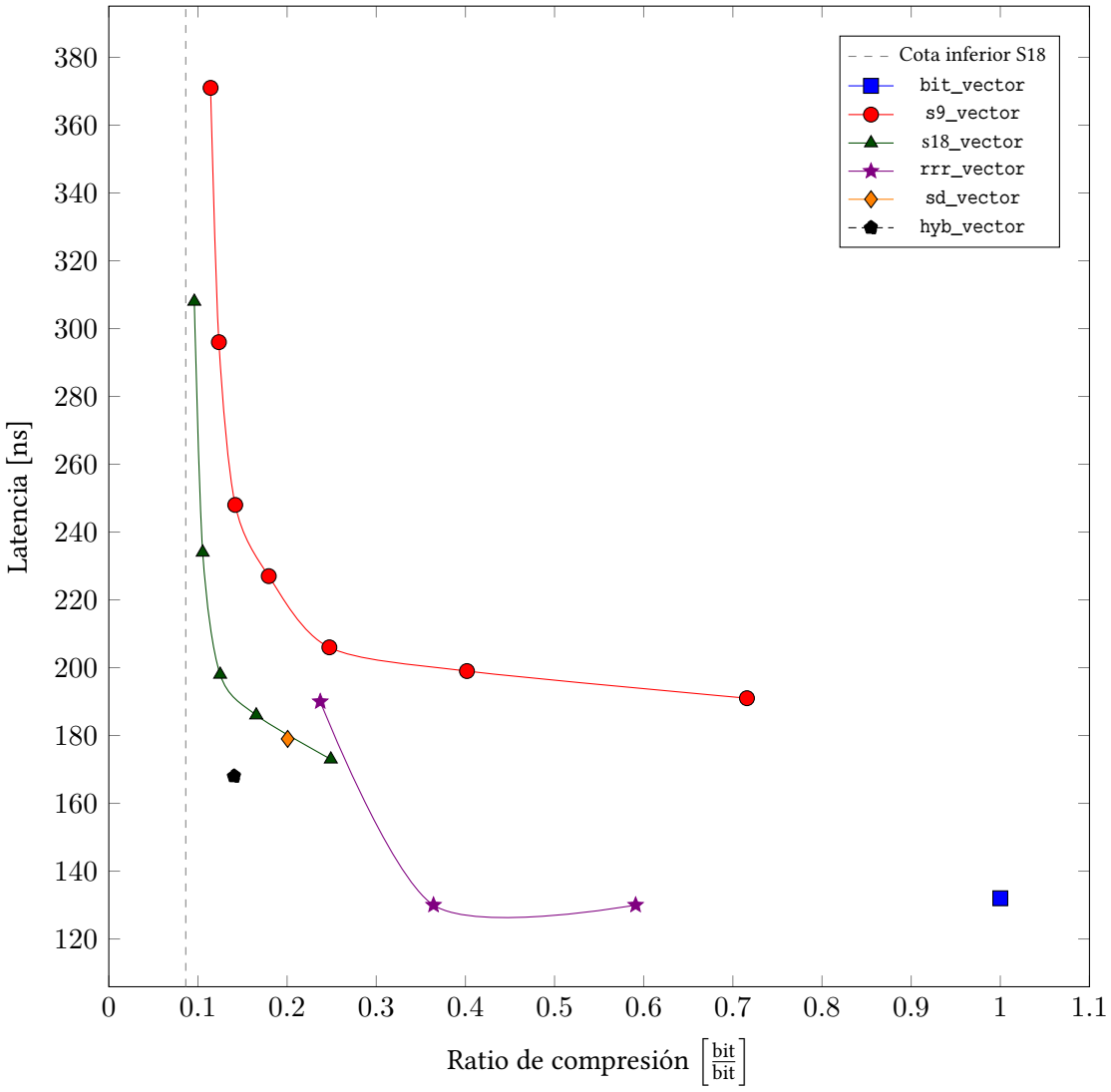


Figura B.17: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.02$. Los resutados completos se encuentran tabulados en la Tabla B.17.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	132	0.1532
s9_vector	8	191	0.1096
s9_vector	16	199	0.0615
s9_vector	32	206	0.0379
s9_vector	64	227	0.0275
s9_vector	128	248	0.0217
s9_vector	256	296	0.0189
s9_vector	512	371	0.0175
s18_vector	1	173	0.0382
s18_vector	2	186	0.0253
s18_vector	4	198	0.0191
s18_vector	8	234	0.0161
s18_vector	16	308	0.0147
s18_vector	32	433	0.0140
s18_vector	64	617	0.0137
rrr_vector	7	130	0.0905
rrr_vector	15	130	0.0558
rrr_vector	31	190	0.0363
rrr_vector	63	195	0.0260
rrr_vector	127	214	0.0218
rrr_vector	255	334	0.0205
sd_vector	-	179	0.0307
hyb_vector	-	168	0.0215

Tabla B.17: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.02$.

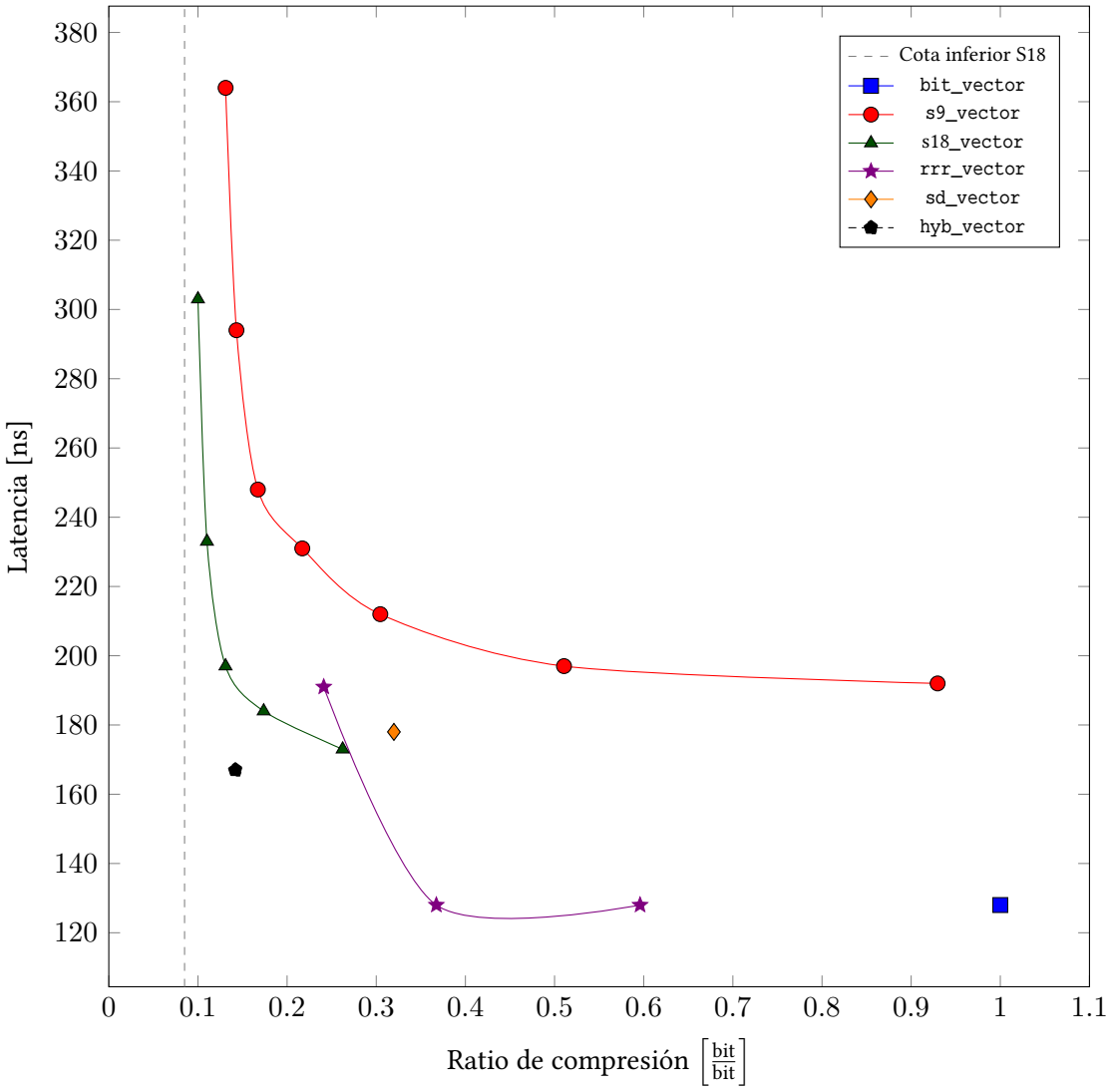


Figura B.18: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.03$. Los resutados completos se encuentran tabulados en la Tabla B.18.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	128	0.1506
s9_vector	8	192	0.1400
s9_vector	16	197	0.0769
s9_vector	32	212	0.0459
s9_vector	64	231	0.0327
s9_vector	128	248	0.0252
s9_vector	256	294	0.0216
s9_vector	512	364	0.0197
s18_vector	1	173	0.0395
s18_vector	2	184	0.0262
s18_vector	4	197	0.0197
s18_vector	8	233	0.0166
s18_vector	16	303	0.0151
s18_vector	32	420	0.0144
s18_vector	64	739	0.0140
rrr_vector	7	128	0.0897
rrr_vector	15	128	0.0553
rrr_vector	31	191	0.0363
rrr_vector	63	193	0.0265
rrr_vector	127	211	0.0232
rrr_vector	255	341	0.0233
sd_vector	-	178	0.0482
hyb_vector	-	167	0.0214

Tabla B.18: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.03$.

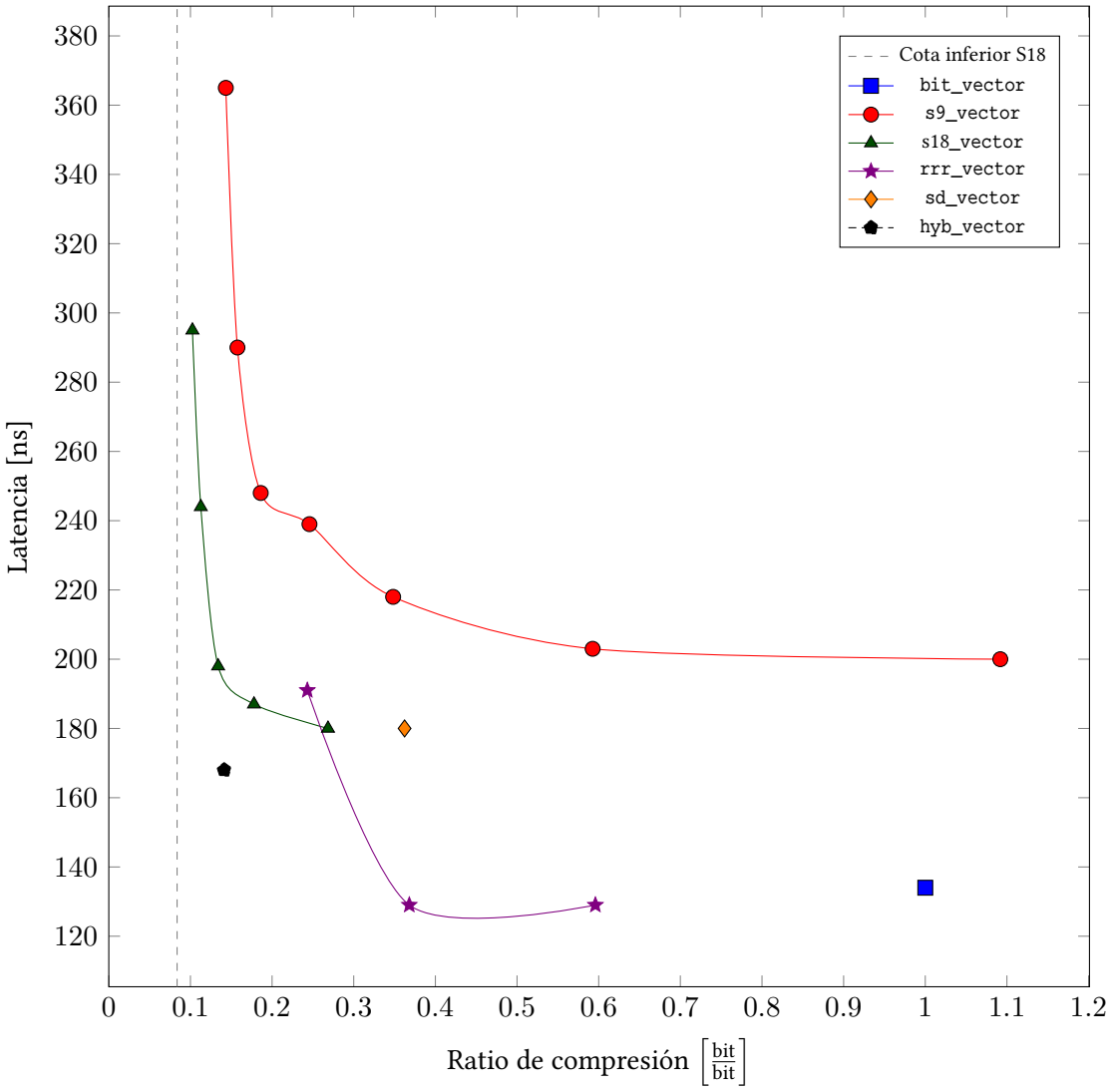


Figura B.19: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.04$. Los resutados completos se encuentran tabulados en la Tabla B.19.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	134	0.1509
s9_vector	8	200	0.1648
s9_vector	16	203	0.0894
s9_vector	32	218	0.0526
s9_vector	64	239	0.0371
s9_vector	128	248	0.0281
s9_vector	256	290	0.0238
s9_vector	512	365	0.0216
s18_vector	1	180	0.0405
s18_vector	2	187	0.0268
s18_vector	4	198	0.0202
s18_vector	8	244	0.0170
s18_vector	16	295	0.0155
s18_vector	32	454	0.0147
s18_vector	64	636	0.0144
rrr_vector	7	129	0.0900
rrr_vector	15	129	0.0556
rrr_vector	31	191	0.0367
rrr_vector	63	194	0.0273
rrr_vector	127	211	0.0247
rrr_vector	255	355	0.0257
sd_vector	-	180	0.0547
hyb_vector	-	168	0.0213

Tabla B.19: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.04$.

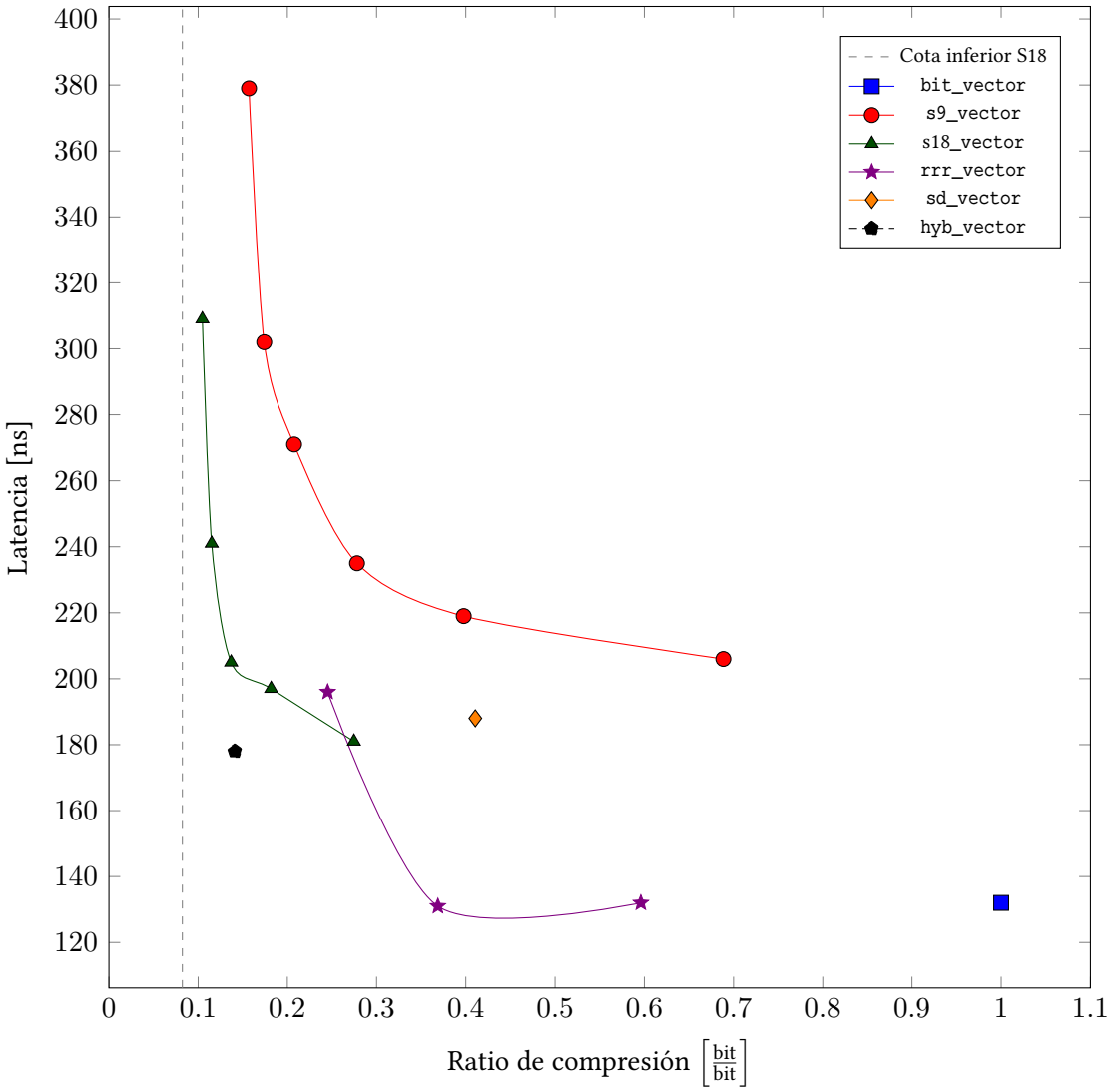


Figura B.20: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.05$. Los resutados completos se encuentran tabulados en la Tabla B.20.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	132	0.1514
s9_vector	8	196	0.1939
s9_vector	16	206	0.1042
s9_vector	32	219	0.0602
s9_vector	64	235	0.0421
s9_vector	128	271	0.0314
s9_vector	256	302	0.0263
s9_vector	512	379	0.0238
s18_vector	1	181	0.0416
s18_vector	2	197	0.0275
s18_vector	4	205	0.0207
s18_vector	8	241	0.0174
s18_vector	16	309	0.0159
s18_vector	32	479	0.0151
s18_vector	64	756	0.0147
rrr_vector	7	132	0.0902
rrr_vector	15	131	0.0558
rrr_vector	31	196	0.0371
rrr_vector	63	199	0.0281
rrr_vector	127	214	0.0267
rrr_vector	255	394	0.0286
sd_vector	-	188	0.0622
hyb_vector	-	178	0.0214

Tabla B.20: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.05$.

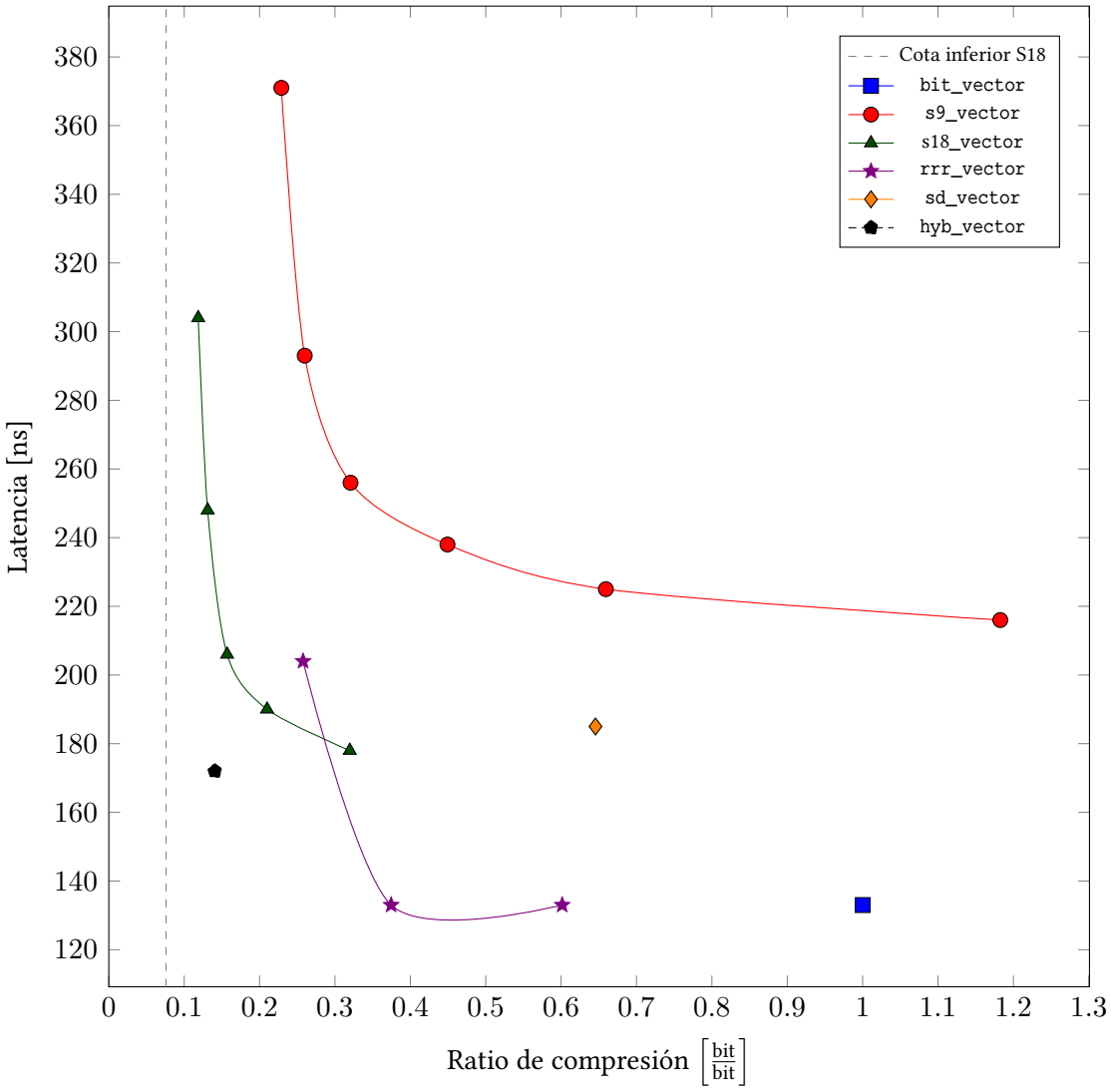


Figura B.21: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.1$. Los resutados completos se encuentran tabulados en la Tabla B.21.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	133	0.1495
s9_vector	8	208	0.3375
s9_vector	16	216	0.1768
s9_vector	32	225	0.0986
s9_vector	64	238	0.0672
s9_vector	128	256	0.0479
s9_vector	256	293	0.0389
s9_vector	512	371	0.0342
s18_vector	1	178	0.0478
s18_vector	2	190	0.0314
s18_vector	4	206	0.0235
s18_vector	8	248	0.0196
s18_vector	16	304	0.0177
s18_vector	32	468	0.0168
s18_vector	64	769	0.0164
rrr_vector	7	133	0.0899
rrr_vector	15	133	0.0560
rrr_vector	31	204	0.0385
rrr_vector	63	204	0.0316
rrr_vector	127	225	0.0345
rrr_vector	255	447	0.0412
sd_vector	-	185	0.0965
hyb_vector	-	172	0.0210

Tabla B.21: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.1$.

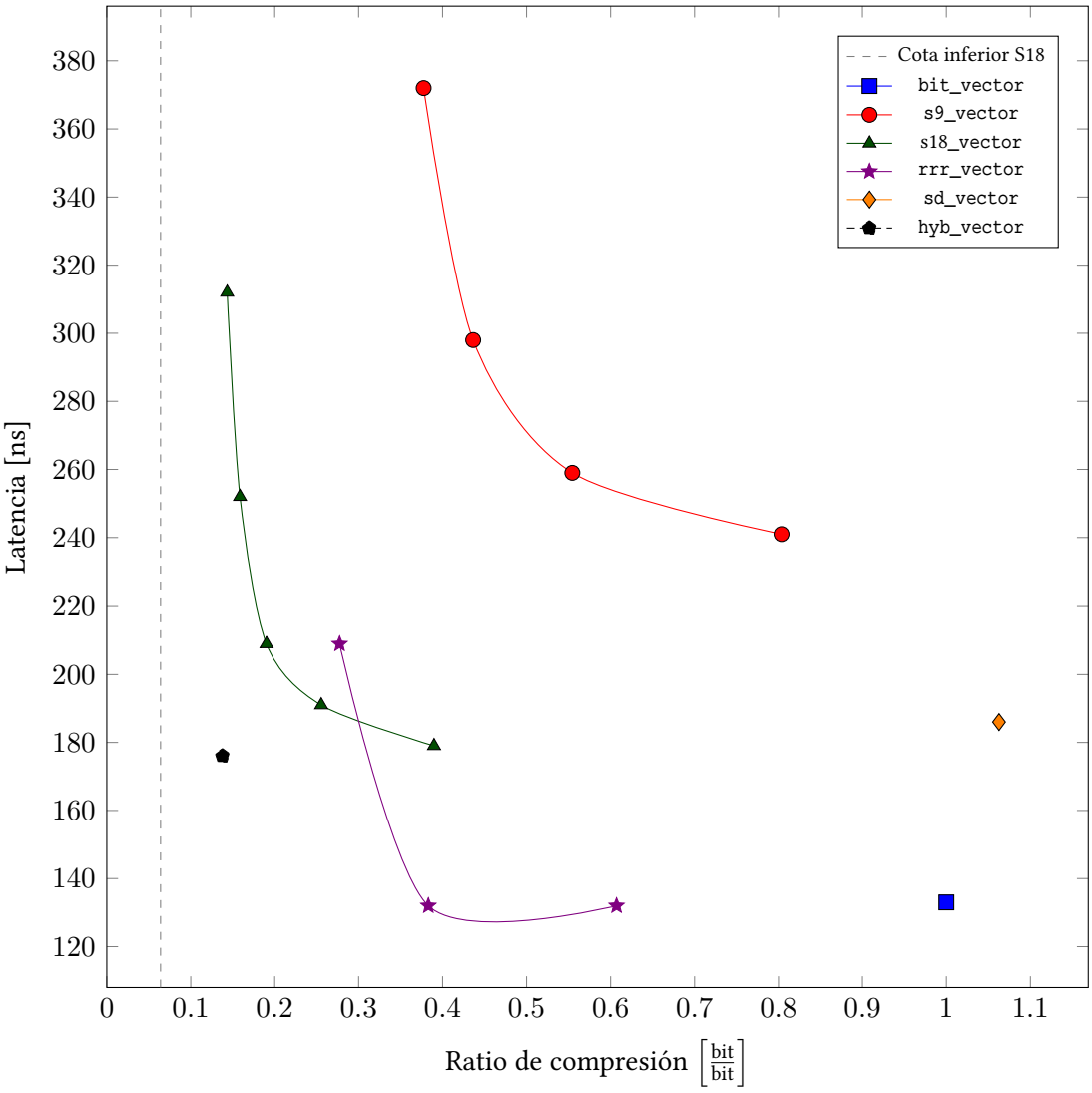


Figura B.22: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.2$. Los resutados completos se encuentran tabulados en la Tabla B.22.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	133	0.1454
s9_vector	8	227	0.6279
s9_vector	16	232	0.3231
s9_vector	32	231	0.1748
s9_vector	64	241	0.1169
s9_vector	128	259	0.0806
s9_vector	256	298	0.0635
s9_vector	512	372	0.0549
s18_vector	1	179	0.0567
s18_vector	2	191	0.0371
s18_vector	4	209	0.0277
s18_vector	8	252	0.0231
s18_vector	16	312	0.0209
s18_vector	32	476	0.0198
s18_vector	64	692	0.0193
rrr_vector	7	132	0.0883
rrr_vector	15	132	0.0557
rrr_vector	31	209	0.0403
rrr_vector	63	207	0.0373
rrr_vector	127	228	0.0474
rrr_vector	255	592	0.0626
sd_vector	-	186	0.1545
hyb_vector	-	176	0.0200

Tabla B.22: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.2$.

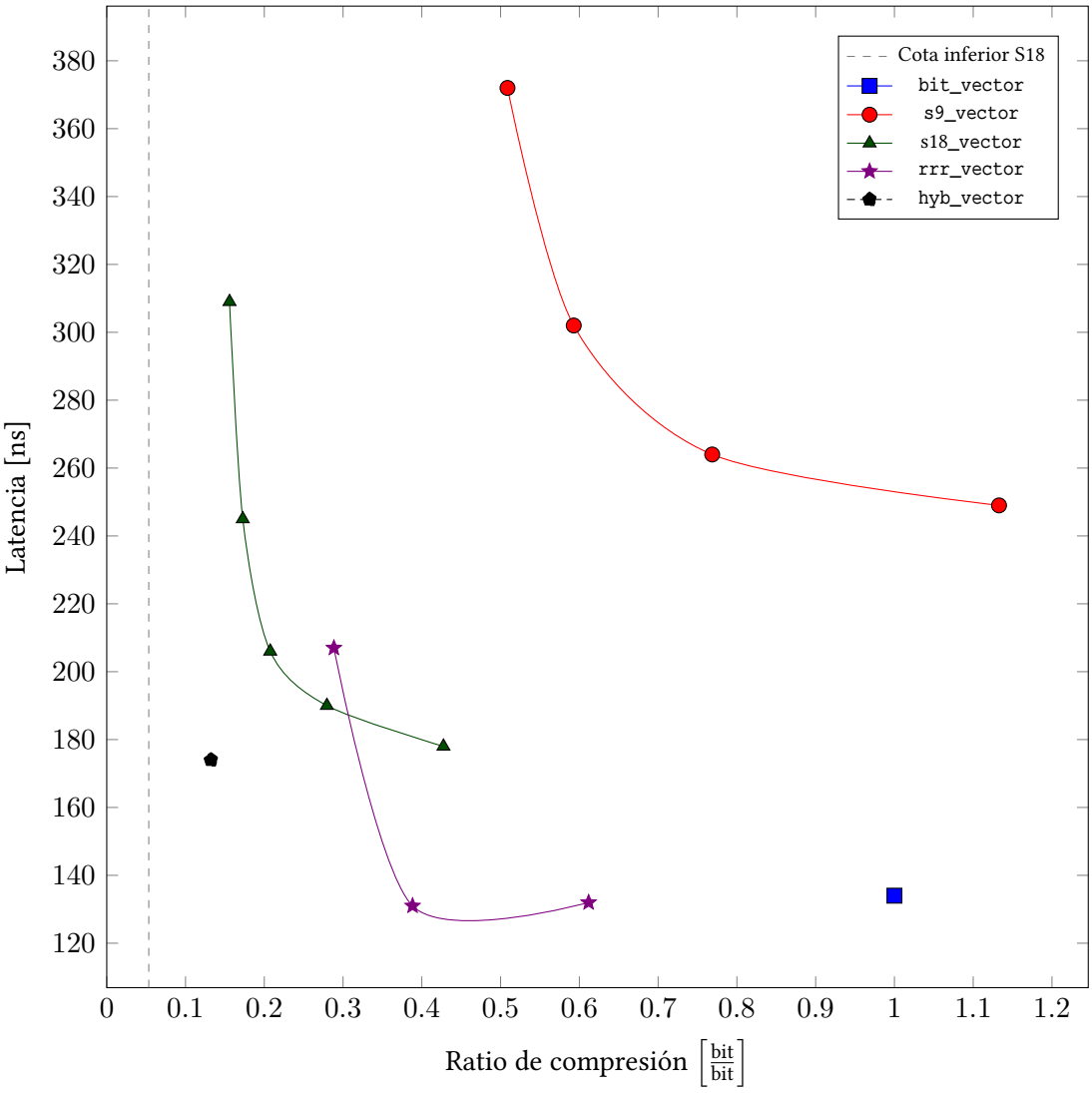


Figura B.23: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.3$. Los resutados completos se encuentran tabulados en la Tabla B.23.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	134	0.1424
s9_vector	8	238	0.8921
s9_vector	16	239	0.4563
s9_vector	32	241	0.2431
s9_vector	64	249	0.1614
s9_vector	128	264	0.1095
s9_vector	256	302	0.0845
s9_vector	512	372	0.0725
s18_vector	1	178	0.0609
s18_vector	2	190	0.0398
s18_vector	4	206	0.0296
s18_vector	8	245	0.0246
s18_vector	16	309	0.0222
s18_vector	32	443	0.0210
s18_vector	64	660	0.0205
rrr_vector	7	132	0.0871
rrr_vector	15	131	0.0553
rrr_vector	31	207	0.0411
rrr_vector	63	209	0.0406
rrr_vector	127	222	0.0540
rrr_vector	255	687	0.0752
sd_vector	-	185	0.2040
hyb_vector	-	174	0.0189

Tabla B.23: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.3$.

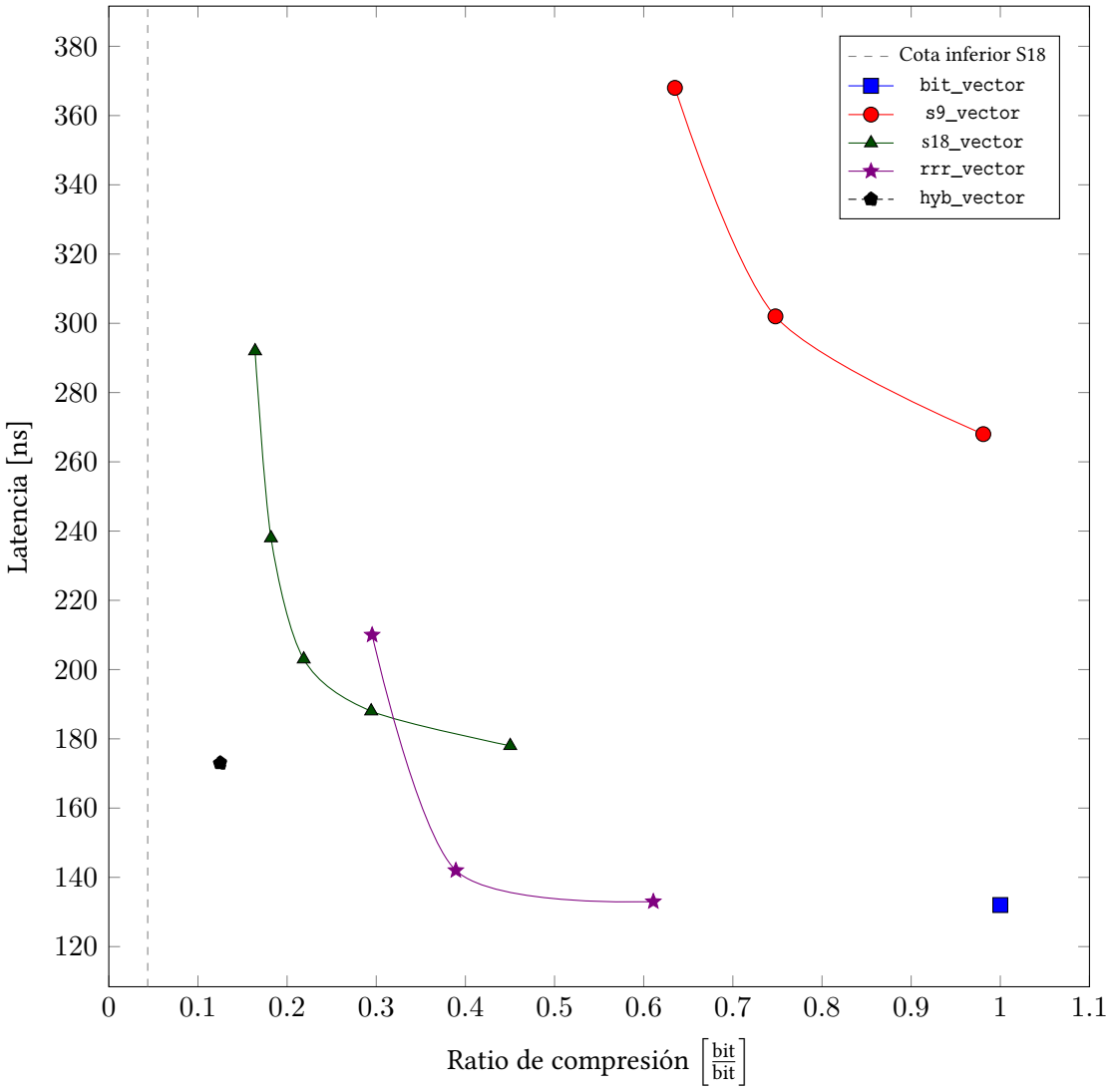


Figura B.24: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.4$. Los resutados completos se encuentran tabulados en la Tabla B.24.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	132	0.1409
s9_vector	8	247	1.1595
s9_vector	16	249	0.5902
s9_vector	32	251	0.3110
s9_vector	64	256	0.2060
s9_vector	128	268	0.1382
s9_vector	256	302	0.1054
s9_vector	512	368	0.0895
s18_vector	1	178	0.0634
s18_vector	2	188	0.0415
s18_vector	4	203	0.0308
s18_vector	8	238	0.0256
s18_vector	16	292	0.0231
s18_vector	32	423	0.0219
s18_vector	64	639	0.0213
rrr_vector	7	133	0.0861
rrr_vector	15	142	0.0549
rrr_vector	31	210	0.0416
rrr_vector	63	211	0.0423
rrr_vector	127	223	0.0593
rrr_vector	255	788	0.0844
sd_vector	-	187	0.2401
hyb_vector	-	173	0.0176

Tabla B.24: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.4$.

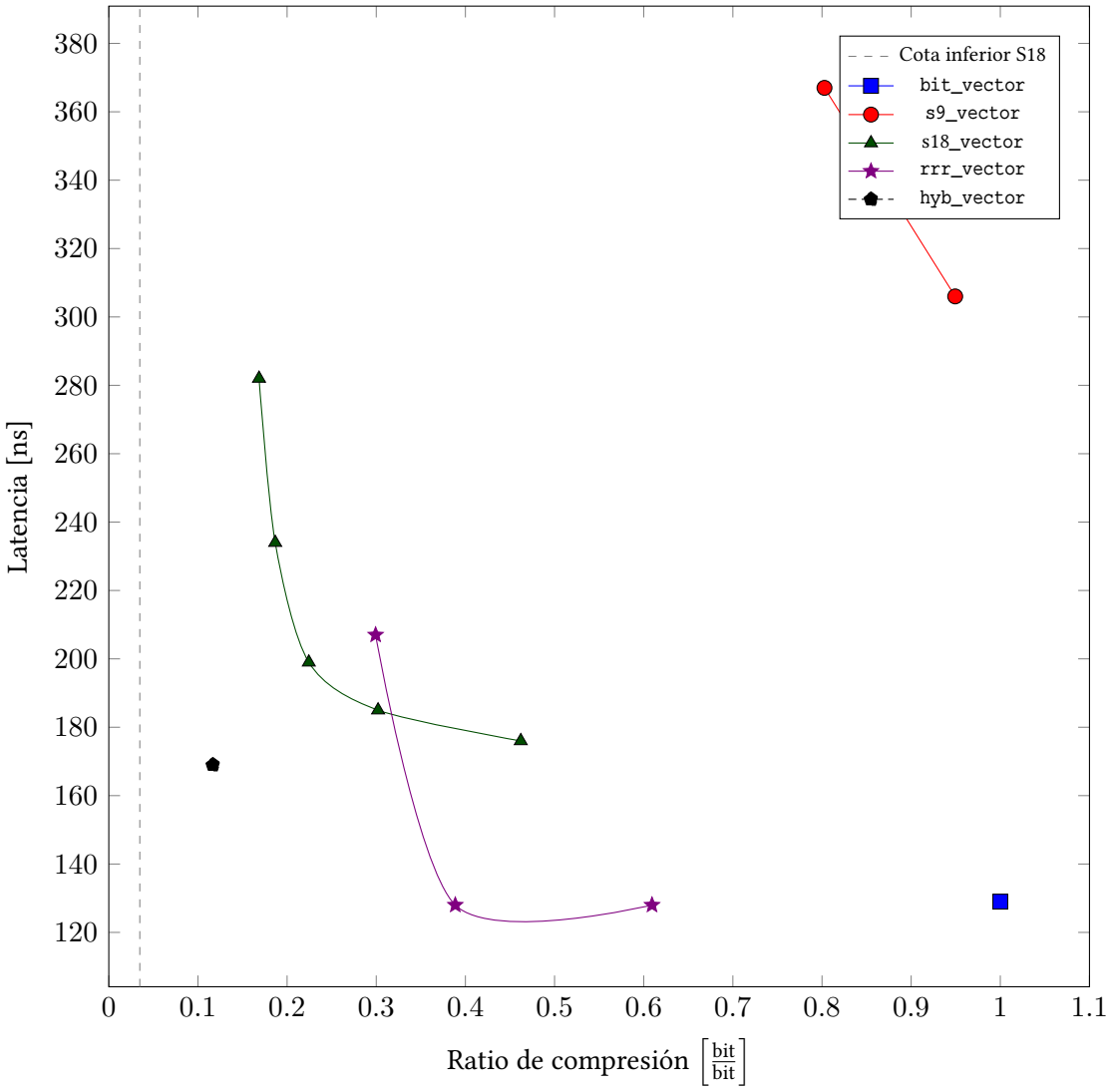


Figura B.25: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.5$. Los resutados completos se encuentran tabulados en la Tabla B.25.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	129	0.1354
s9_vector	8	253	1.4722
s9_vector	16	257	0.7466
s9_vector	32	255	0.3894
s9_vector	64	262	0.2578
s9_vector	128	270	0.1705
s9_vector	256	306	0.1286
s9_vector	512	367	0.1087
s18_vector	1	176	0.0626
s18_vector	2	185	0.0409
s18_vector	4	199	0.0304
s18_vector	8	234	0.0253
s18_vector	16	282	0.0228
s18_vector	32	406	0.0216
s18_vector	64	642	0.0210
rrr_vector	7	128	0.0825
rrr_vector	15	128	0.0526
rrr_vector	31	207	0.0405
rrr_vector	63	217	0.0420
rrr_vector	127	223	0.0593
rrr_vector	255	849	0.0880
sd_vector	-	184	0.2821
hyb_vector	-	169	0.0158

Tabla B.25: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.5$.

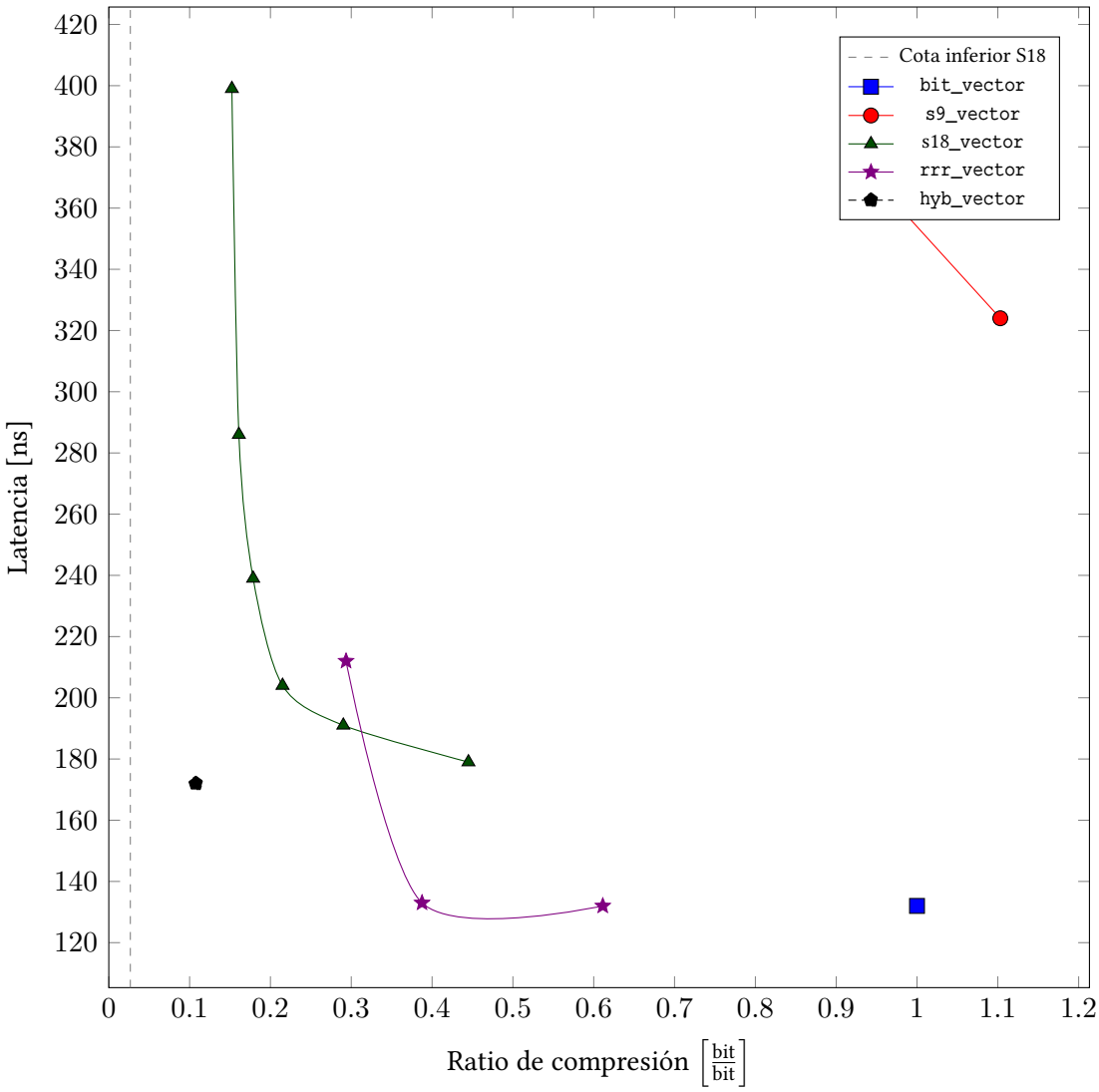


Figura B.26: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.6$. Los resutados completos se encuentran tabulados en la Tabla B.26.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	132	0.1333
s9_vector	8	266	1.7407
s9_vector	16	266	0.8799
s9_vector	32	269	0.4550
s9_vector	64	278	0.3014
s9_vector	128	283	0.1970
s9_vector	256	324	0.1470
s9_vector	512	374	0.1241
s18_vector	1	179	0.0593
s18_vector	2	191	0.0387
s18_vector	4	204	0.0287
s18_vector	8	239	0.0238
s18_vector	16	286	0.0215
s18_vector	32	399	0.0203
s18_vector	64	591	0.0197
rrr_vector	7	132	0.0815
rrr_vector	15	133	0.0517
rrr_vector	31	212	0.0391
rrr_vector	63	212	0.0406
rrr_vector	127	226	0.0563
rrr_vector	255	868	0.0837
sd_vector	-	189	0.3243
hyb_vector	-	172	0.0144

Tabla B.26: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.6$.

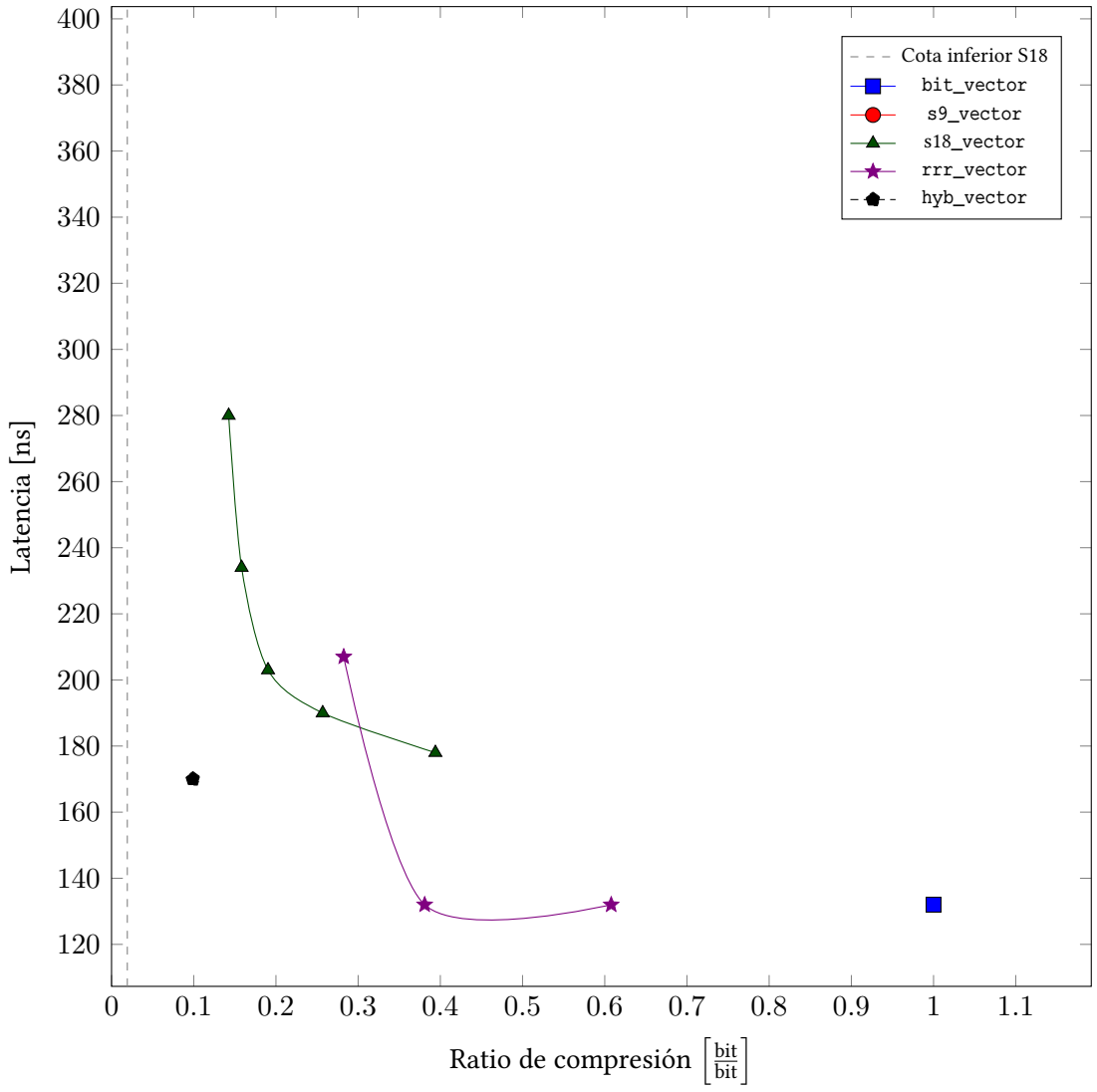


Figura B.27: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.7$. Los resutados completos se encuentran tabulados en la Tabla B.27.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	132	0.1288
s9_vector	8	270	2.0358
s9_vector	16	274	1.0258
s9_vector	32	278	0.5256
s9_vector	64	291	0.3485
s9_vector	128	296	0.2253
s9_vector	256	330	0.1653
s9_vector	512	379	0.1396
s18_vector	1	178	0.0508
s18_vector	2	190	0.0331
s18_vector	4	203	0.0245
s18_vector	8	234	0.0204
s18_vector	16	280	0.0184
s18_vector	32	416	0.0174
s18_vector	64	606	0.0169
rrr_vector	7	132	0.0783
rrr_vector	15	132	0.0491
rrr_vector	31	207	0.0364
rrr_vector	63	210	0.0364
rrr_vector	127	223	0.0499
rrr_vector	255	833	0.0729
sd_vector	-	191	0.3518
hyb_vector	-	170	0.0128

Tabla B.27: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.7$.

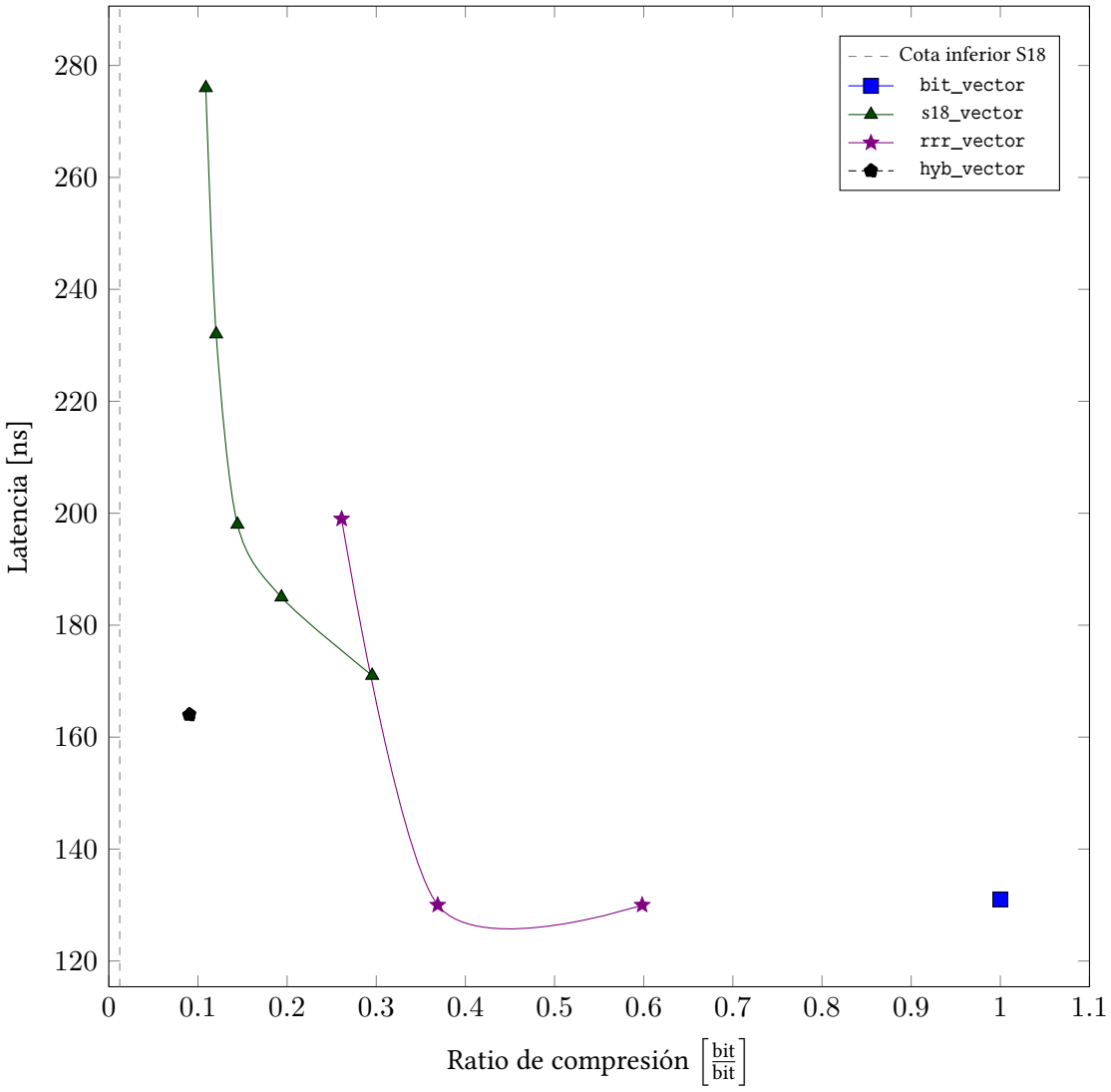


Figura B.28: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.8$. Los resutados completos se encuentran tabulados en la Tabla B.28.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	131	0.1252
s9_vector	8	284	2.3285
s9_vector	16	285	1.1701
s9_vector	32	287	0.5944
s9_vector	64	296	0.3945
s9_vector	128	300	0.2521
s9_vector	256	336	0.1821
s9_vector	512	379	0.1538
s18_vector	1	171	0.0370
s18_vector	2	185	0.0243
s18_vector	4	198	0.0181
s18_vector	8	232	0.0151
s18_vector	16	276	0.0136
s18_vector	32	409	0.0129
s18_vector	64	594	0.0126
rrr_vector	7	130	0.0749
rrr_vector	15	130	0.0462
rrr_vector	31	199	0.0327
rrr_vector	63	205	0.0303
rrr_vector	127	216	0.0385
rrr_vector	255	705	0.0561
sd_vector	-	190	0.3790
hyb_vector	-	164	0.0113

Tabla B.28: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.8$.

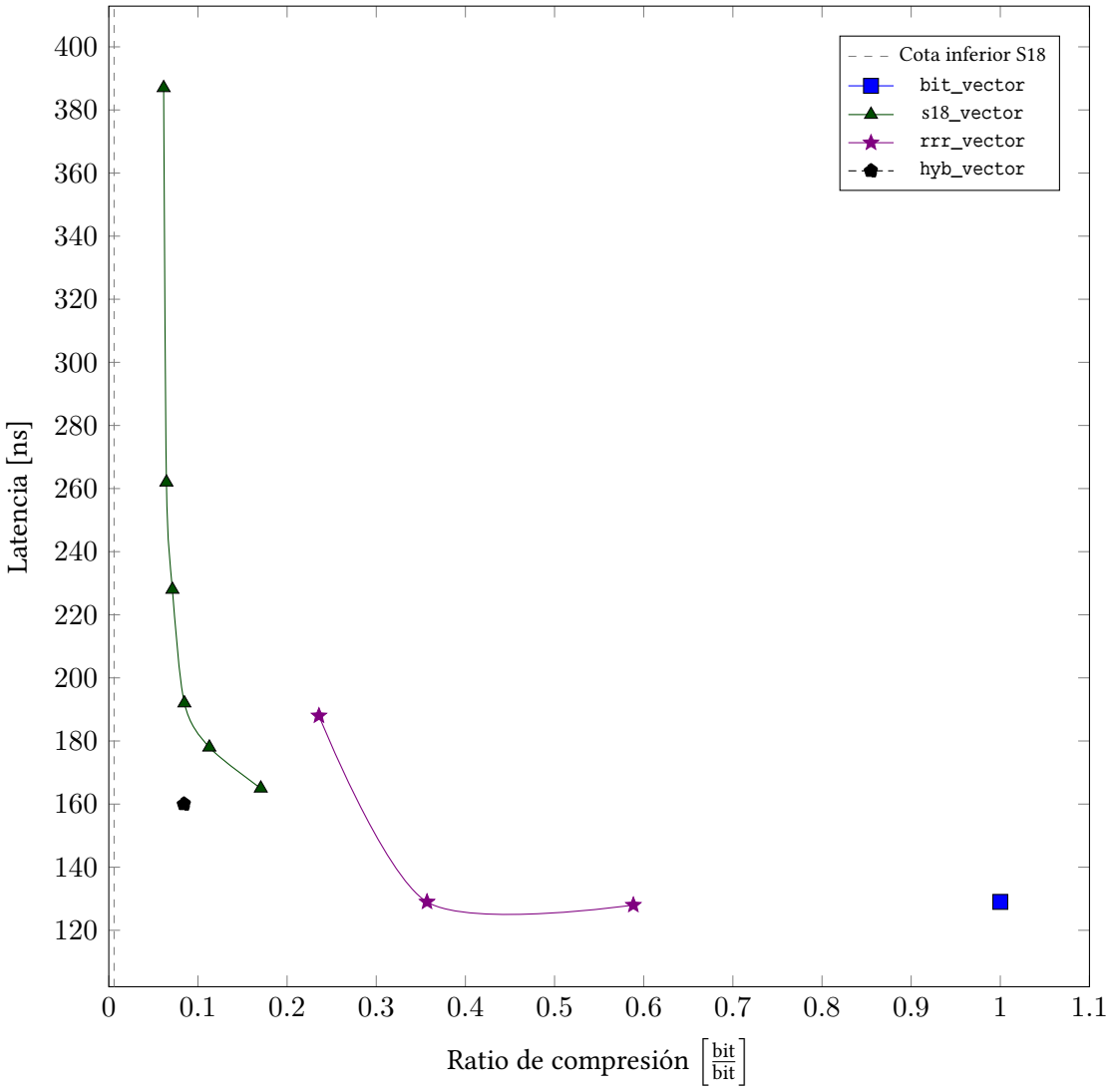


Figura B.29: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.9$. Los resutados completos se encuentran tabulados en la Tabla B.29.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	129	0.1227
s9_vector	8	289	2.5763
s9_vector	16	292	1.2914
s9_vector	32	294	0.6509
s9_vector	64	308	0.4328
s9_vector	128	308	0.2735
s9_vector	256	342	0.1948
s9_vector	512	379	0.1652
s18_vector	1	165	0.0209
s18_vector	2	178	0.0138
s18_vector	4	192	0.0104
s18_vector	8	228	0.0088
s18_vector	16	262	0.0079
s18_vector	32	387	0.0075
s18_vector	64	561	0.0074
rrr_vector	7	128	0.0722
rrr_vector	15	129	0.0438
rrr_vector	31	188	0.0289
rrr_vector	63	197	0.0231
rrr_vector	127	207	0.0256
rrr_vector	255	514	0.0344
sd_vector	-	185	0.3254
hyb_vector	-	160	0.0103

Tabla B.29: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.9$.

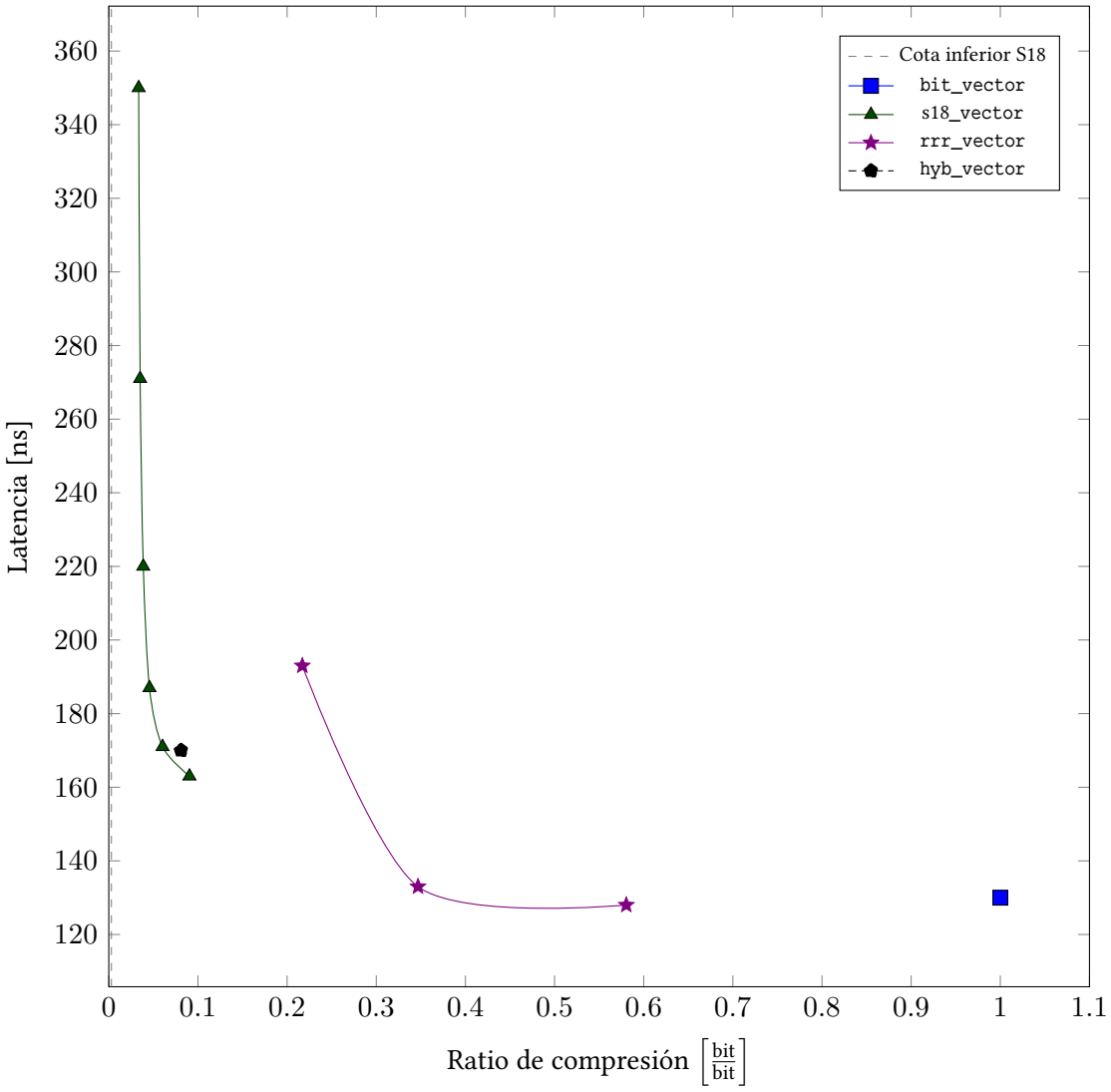


Figura B.30: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.95$. Los resutados completos se encuentran tabulados en la Tabla B.30.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	130	0.1208
s9_vector	8	293	2.7266
s9_vector	16	297	1.3651
s9_vector	32	300	0.6854
s9_vector	64	315	0.4562
s9_vector	128	315	0.2867
s9_vector	256	346	0.2023
s9_vector	512	383	0.1723
s18_vector	1	163	0.0109
s18_vector	2	171	0.0073
s18_vector	4	187	0.0055
s18_vector	8	220	0.0047
s18_vector	16	271	0.0042
s18_vector	32	350	0.0040
s18_vector	64	590	0.0040
rrr_vector	7	128	0.0701
rrr_vector	15	133	0.0419
rrr_vector	31	193	0.0262
rrr_vector	63	190	0.0187
rrr_vector	127	212	0.0171
rrr_vector	255	428	0.0203
sd_vector	-	199	0.3394
hyb_vector	-	170	0.0098

Tabla B.30: Rendimiento de la operación RANK. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.95$.

B.3. Select

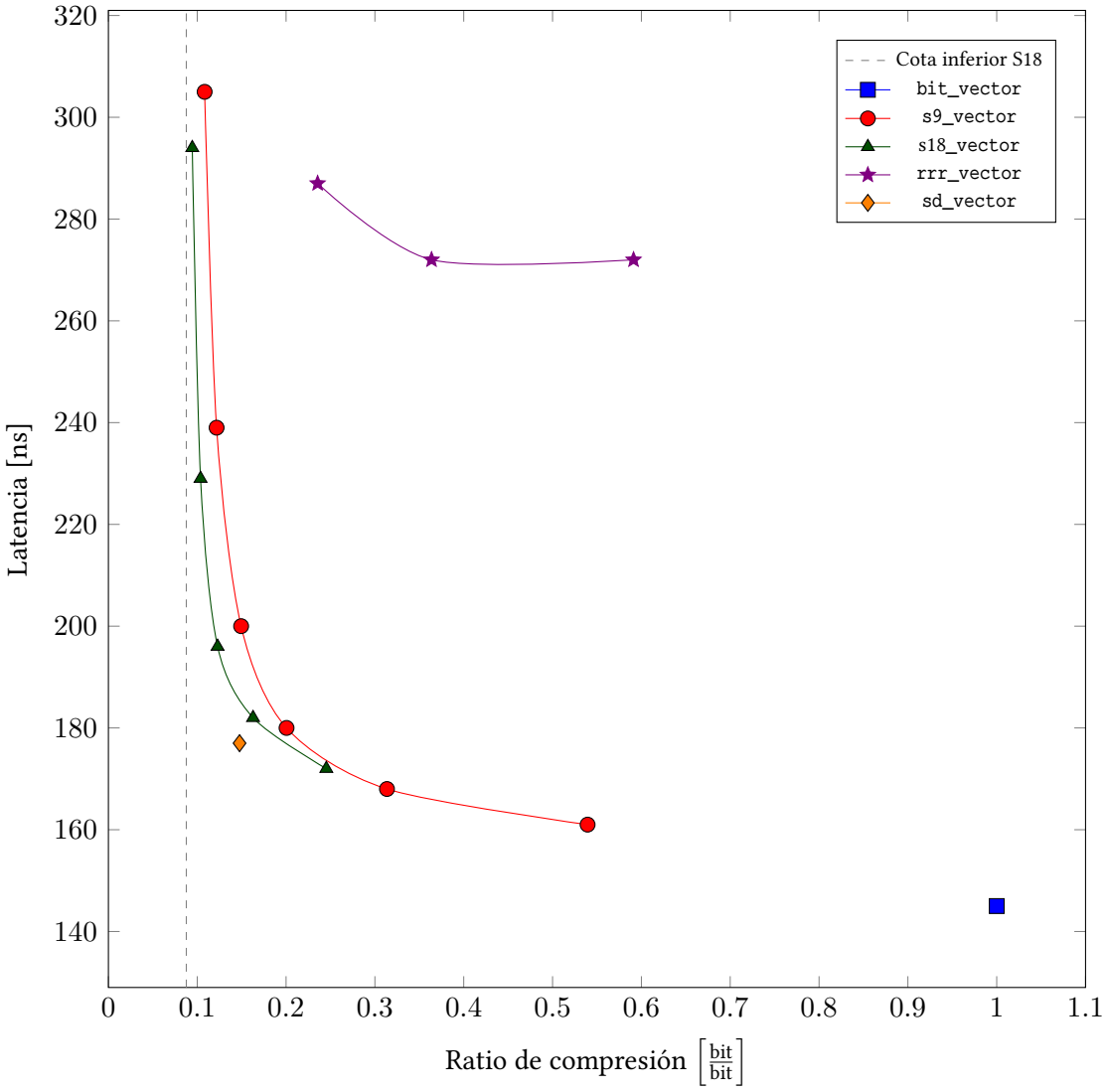


Figura B.31: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.01$. Los resutados completos se encuentran tabulados en la Tabla B.31.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	145	0.1512
s9_vector	8	161	0.0815
s9_vector	16	168	0.0474
s9_vector	32	180	0.0303
s9_vector	64	200	0.0226
s9_vector	128	239	0.0184
s9_vector	256	305	0.0164
s9_vector	512	443	0.0154
s18_vector	1	172	0.0371
s18_vector	2	182	0.0246
s18_vector	4	196	0.0186
s18_vector	8	229	0.0157
s18_vector	16	294	0.0143
s18_vector	32	402	0.0136
s18_vector	64	691	0.0133
rrr_vector	7	272	0.0894
rrr_vector	15	272	0.0550
rrr_vector	31	287	0.0356
rrr_vector	63	281	0.0249
rrr_vector	127	300	0.0196
rrr_vector	255	409	0.0175
sd_vector	-	177	0.0223

Tabla B.31: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.01$.

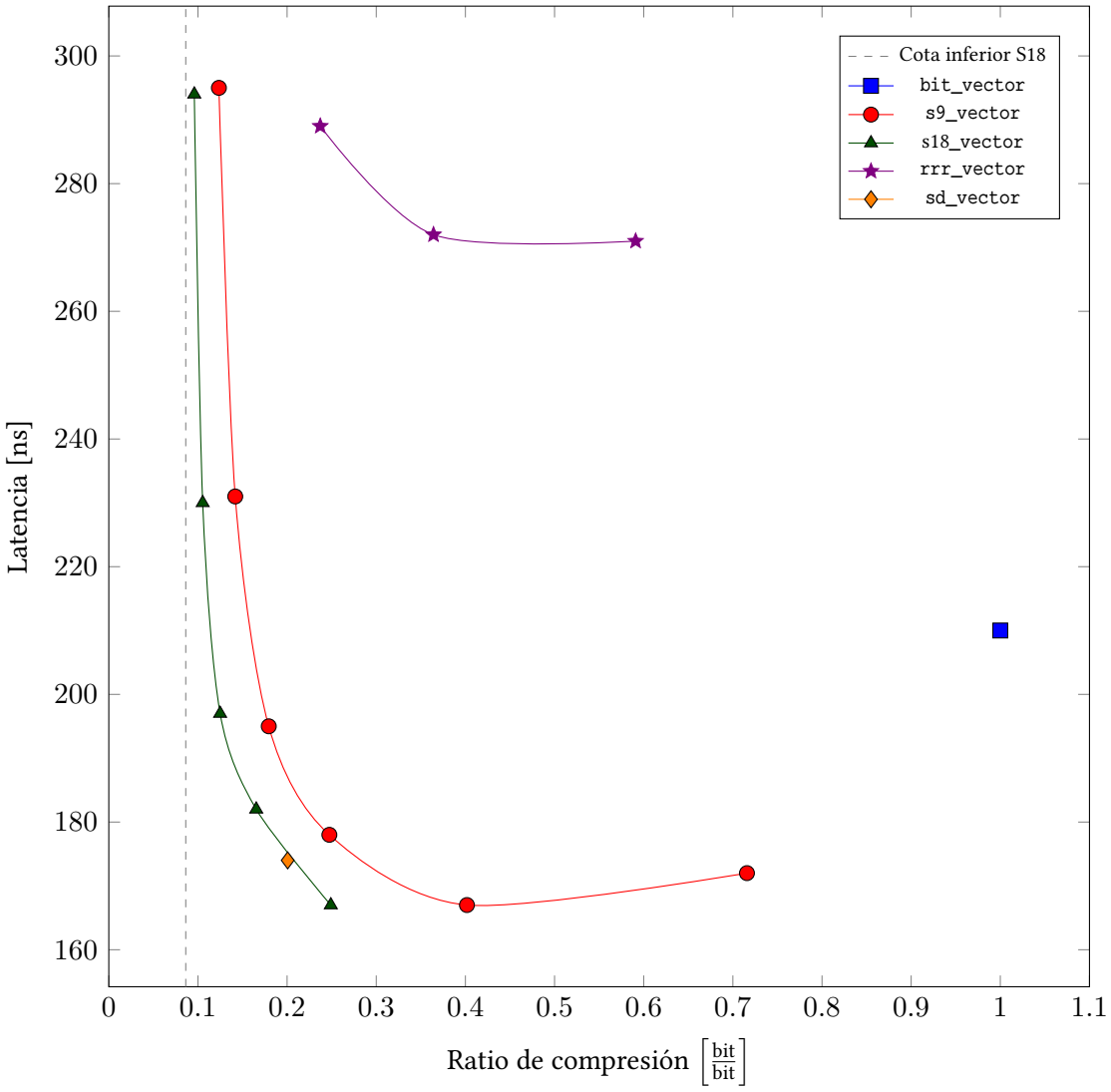


Figura B.32: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.02$. Los resutados completos se encuentran tabulados en la Tabla B.32.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	210	0.1532
s9_vector	8	172	0.1096
s9_vector	16	167	0.0615
s9_vector	32	178	0.0379
s9_vector	64	195	0.0275
s9_vector	128	231	0.0217
s9_vector	256	295	0.0189
s9_vector	512	421	0.0175
s18_vector	1	167	0.0382
s18_vector	2	182	0.0253
s18_vector	4	197	0.0191
s18_vector	8	230	0.0161
s18_vector	16	294	0.0147
s18_vector	32	420	0.0140
s18_vector	64	706	0.0137
rrr_vector	7	271	0.0905
rrr_vector	15	272	0.0558
rrr_vector	31	289	0.0363
rrr_vector	63	284	0.0260
rrr_vector	127	300	0.0218
rrr_vector	255	418	0.0205
sd_vector	-	174	0.0307

Tabla B.32: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.02$.

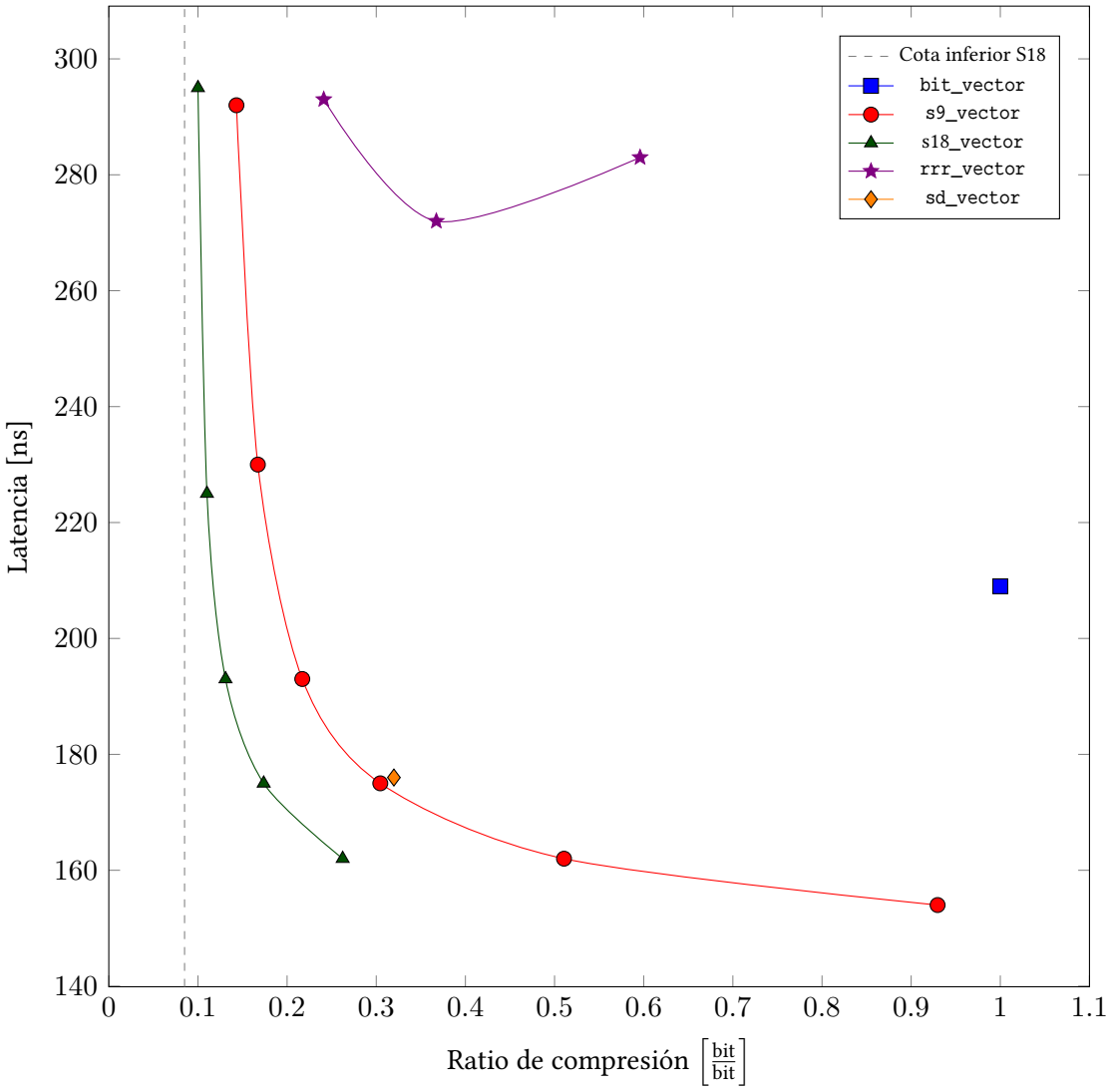


Figura B.33: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.03$. Los resutados completos se encuentran tabulados en la Tabla B.33.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	209	0.1506
s9_vector	8	154	0.1400
s9_vector	16	162	0.0769
s9_vector	32	175	0.0459
s9_vector	64	193	0.0327
s9_vector	128	230	0.0252
s9_vector	256	292	0.0216
s9_vector	512	408	0.0197
s18_vector	1	162	0.0395
s18_vector	2	175	0.0262
s18_vector	4	193	0.0197
s18_vector	8	225	0.0166
s18_vector	16	295	0.0151
s18_vector	32	454	0.0144
s18_vector	64	697	0.0140
rrr_vector	7	283	0.0897
rrr_vector	15	272	0.0553
rrr_vector	31	293	0.0363
rrr_vector	63	287	0.0265
rrr_vector	127	297	0.0232
rrr_vector	255	421	0.0233
sd_vector	-	176	0.0482

Tabla B.33: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.03$.

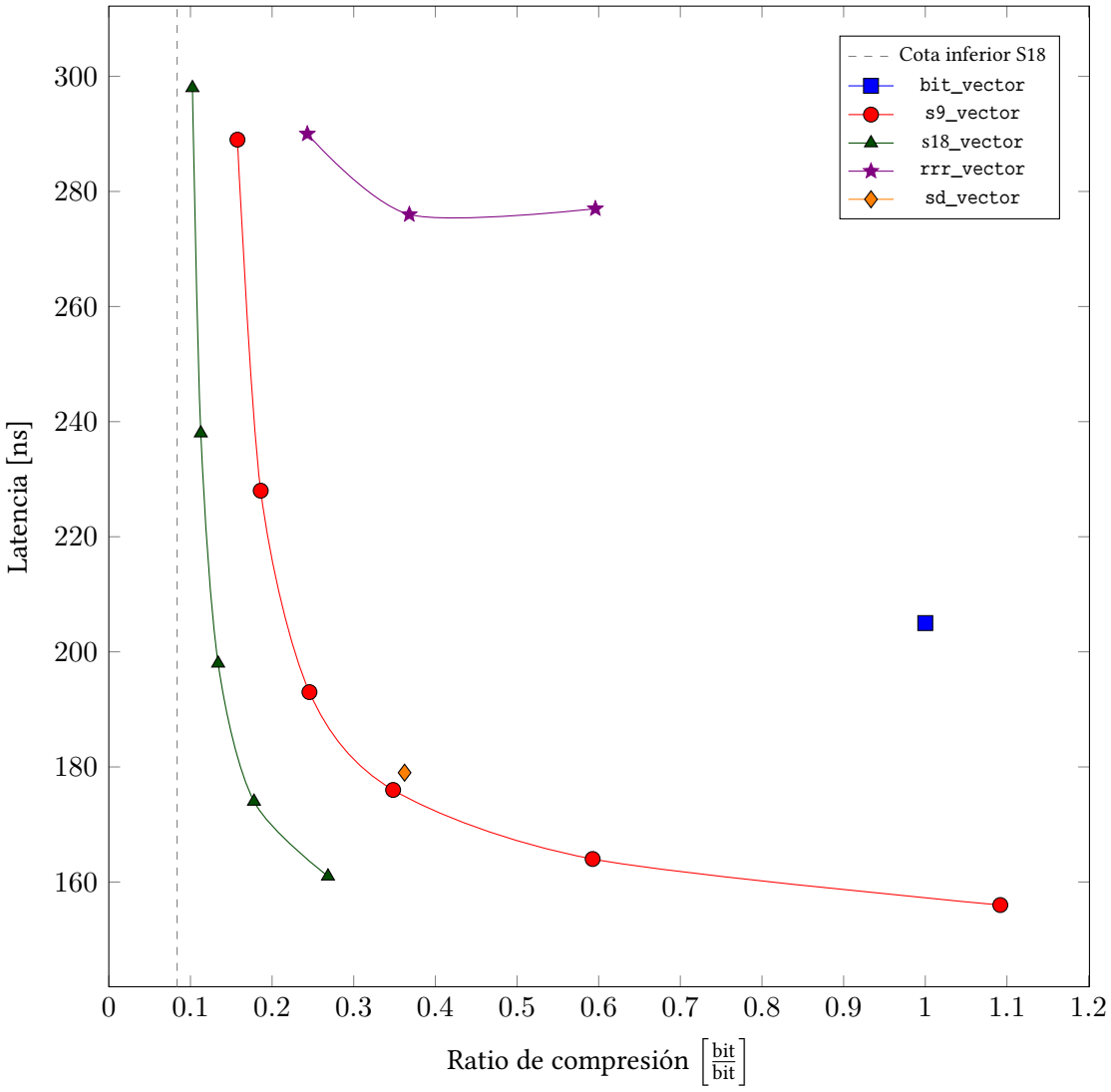


Figura B.34: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.04$. Los resutados completos se encuentran tabulados en la Tabla B.34.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	205	0.1509
s9_vector	8	156	0.1648
s9_vector	16	164	0.0894
s9_vector	32	176	0.0526
s9_vector	64	193	0.0371
s9_vector	128	228	0.0281
s9_vector	256	289	0.0238
s9_vector	512	408	0.0216
s18_vector	1	161	0.0405
s18_vector	2	174	0.0268
s18_vector	4	198	0.0202
s18_vector	8	238	0.0170
s18_vector	16	298	0.0155
s18_vector	32	441	0.0147
s18_vector	64	655	0.0144
rrr_vector	7	277	0.0900
rrr_vector	15	276	0.0556
rrr_vector	31	290	0.0367
rrr_vector	63	289	0.0273
rrr_vector	127	304	0.0247
rrr_vector	255	437	0.0257
sd_vector	-	179	0.0547

Tabla B.34: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.04$.

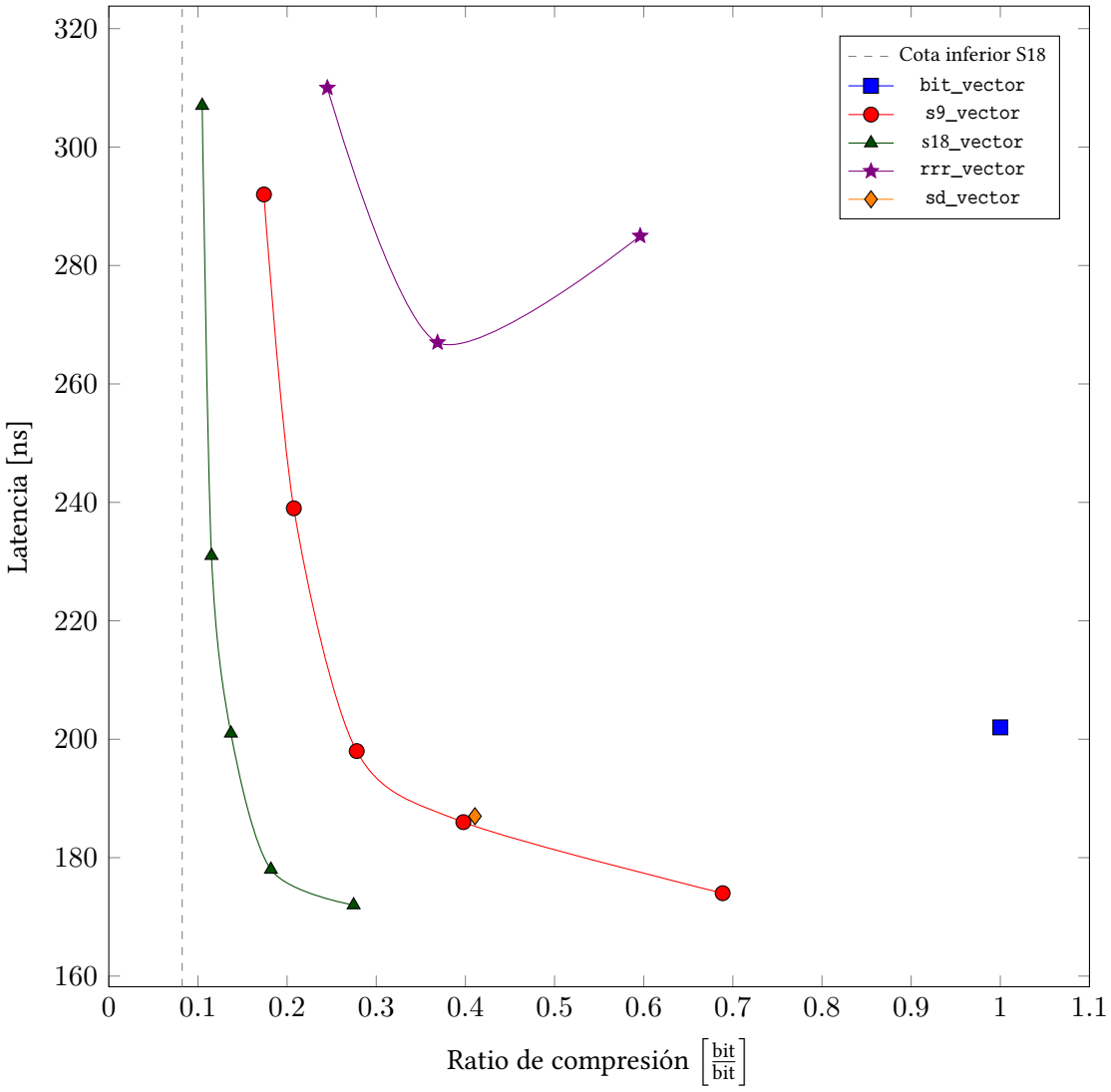


Figura B.35: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.05$. Los resutados completos se encuentran tabulados en la Tabla B.35.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	202	0.1514
s9_vector	8	161	0.1939
s9_vector	16	174	0.1042
s9_vector	32	186	0.0602
s9_vector	64	198	0.0421
s9_vector	128	239	0.0314
s9_vector	256	292	0.0263
s9_vector	512	406	0.0238
s18_vector	1	172	0.0416
s18_vector	2	178	0.0275
s18_vector	4	201	0.0207
s18_vector	8	231	0.0174
s18_vector	16	307	0.0159
s18_vector	32	430	0.0151
s18_vector	64	760	0.0147
rrr_vector	7	285	0.0902
rrr_vector	15	267	0.0558
rrr_vector	31	310	0.0371
rrr_vector	63	323	0.0281
rrr_vector	127	325	0.0267
rrr_vector	255	502	0.0286
sd_vector	-	187	0.0622

Tabla B.35: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.05$.

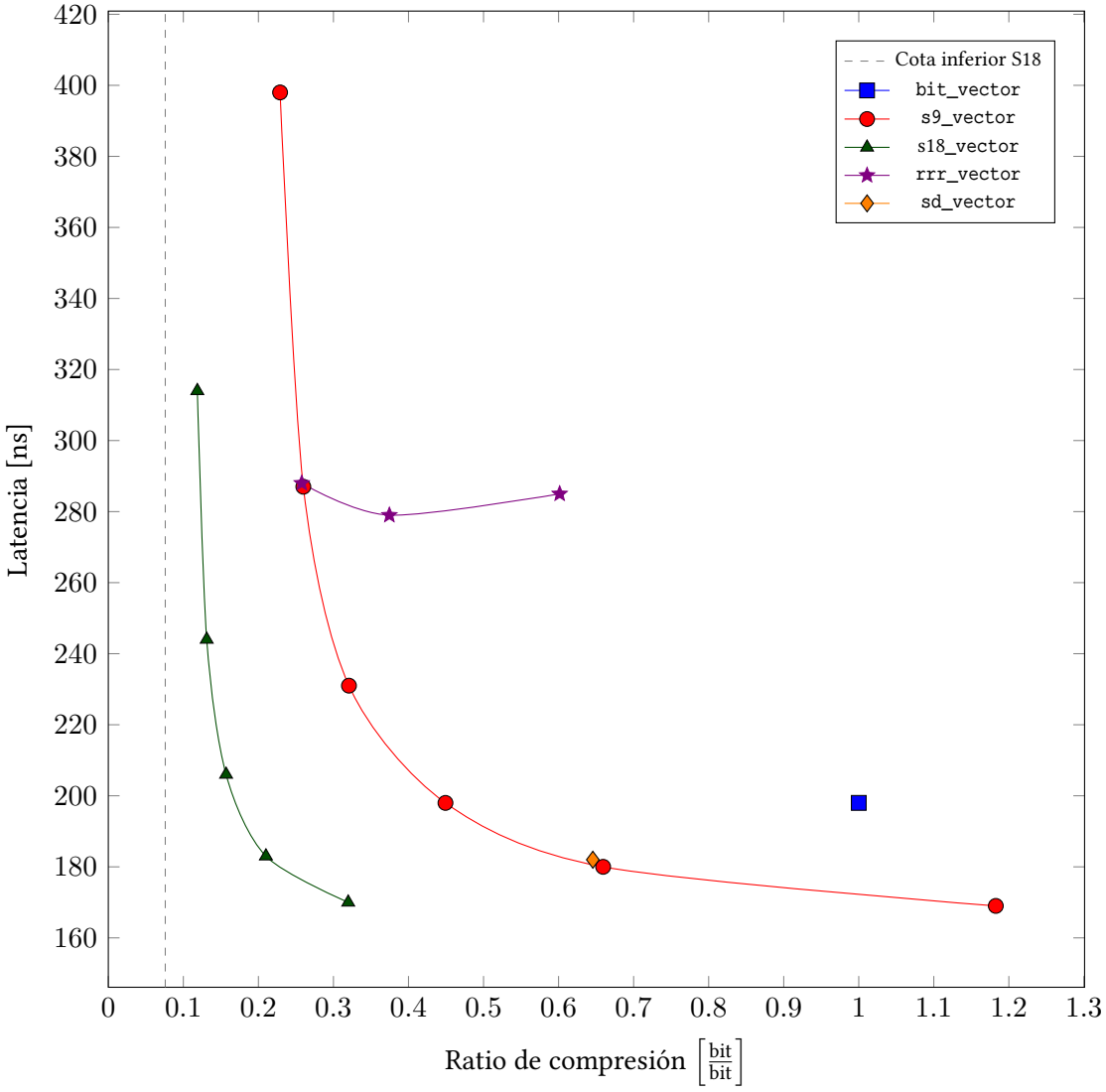


Figura B.36: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.1$. Los resutados completos se encuentran tabulados en la Tabla B.36.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	198	0.1495
s9_vector	8	165	0.3375
s9_vector	16	169	0.1768
s9_vector	32	180	0.0986
s9_vector	64	198	0.0672
s9_vector	128	231	0.0479
s9_vector	256	287	0.0389
s9_vector	512	398	0.0342
s18_vector	1	170	0.0478
s18_vector	2	183	0.0314
s18_vector	4	206	0.0235
s18_vector	8	244	0.0196
s18_vector	16	314	0.0177
s18_vector	32	431	0.0168
s18_vector	64	663	0.0164
rrr_vector	7	285	0.0899
rrr_vector	15	279	0.0560
rrr_vector	31	288	0.0385
rrr_vector	63	292	0.0316
rrr_vector	127	317	0.0345
rrr_vector	255	424	0.0412
sd_vector	-	182	0.0965

Tabla B.36: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.1$.

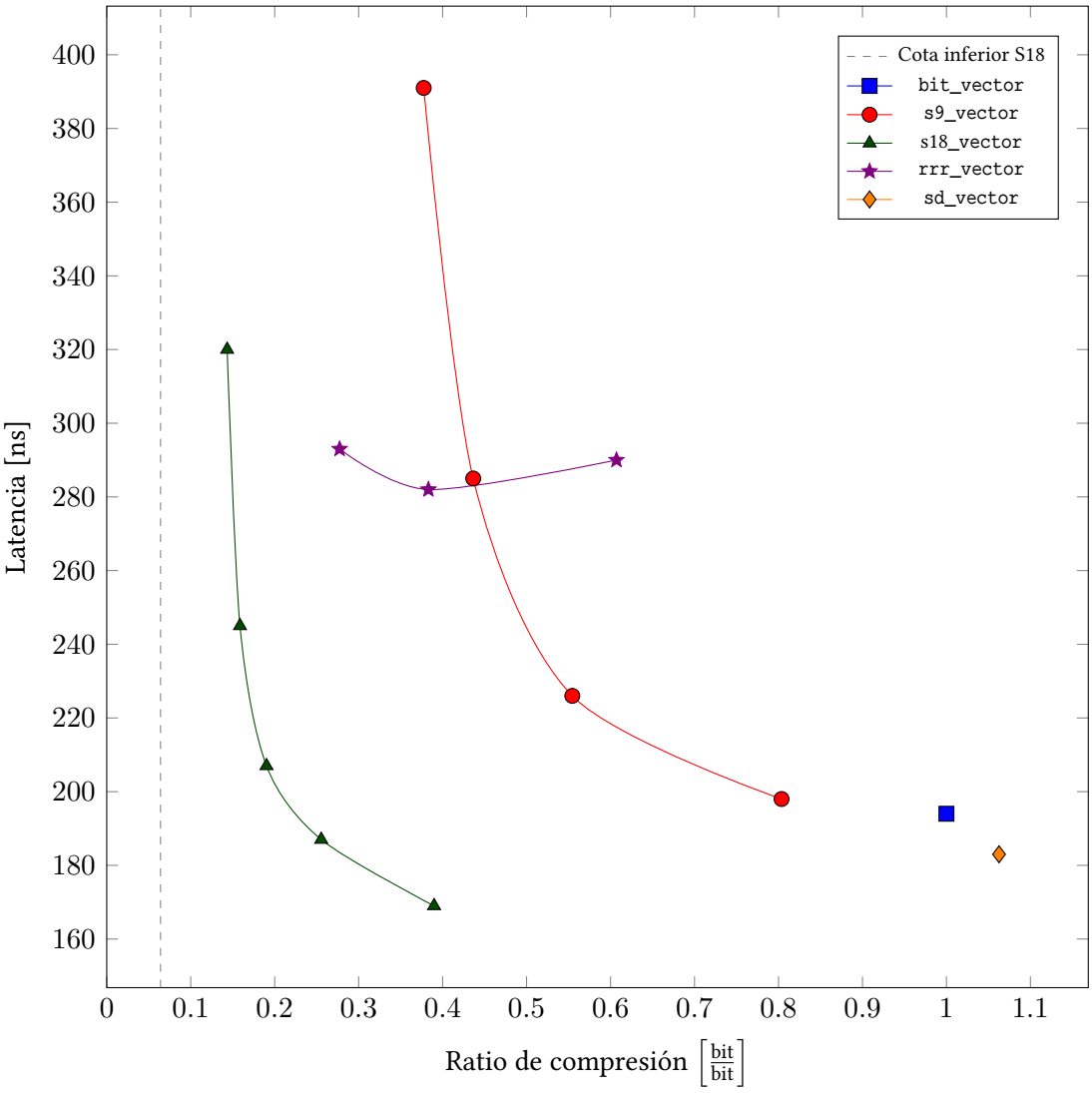


Figura B.37: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.2$. Los resutados completos se encuentran tabulados en la Tabla B.37.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	194	0.1454
s9_vector	8	168	0.6279
s9_vector	16	171	0.3231
s9_vector	32	183	0.1748
s9_vector	64	198	0.1169
s9_vector	128	226	0.0806
s9_vector	256	285	0.0635
s9_vector	512	391	0.0549
s18_vector	1	169	0.0567
s18_vector	2	187	0.0371
s18_vector	4	207	0.0277
s18_vector	8	245	0.0231
s18_vector	16	320	0.0209
s18_vector	32	464	0.0198
s18_vector	64	737	0.0193
rrr_vector	7	290	0.0883
rrr_vector	15	282	0.0557
rrr_vector	31	293	0.0403
rrr_vector	63	283	0.0373
rrr_vector	127	298	0.0474
rrr_vector	255	402	0.0626
sd_vector	-	183	0.1545

Tabla B.37: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.2$.

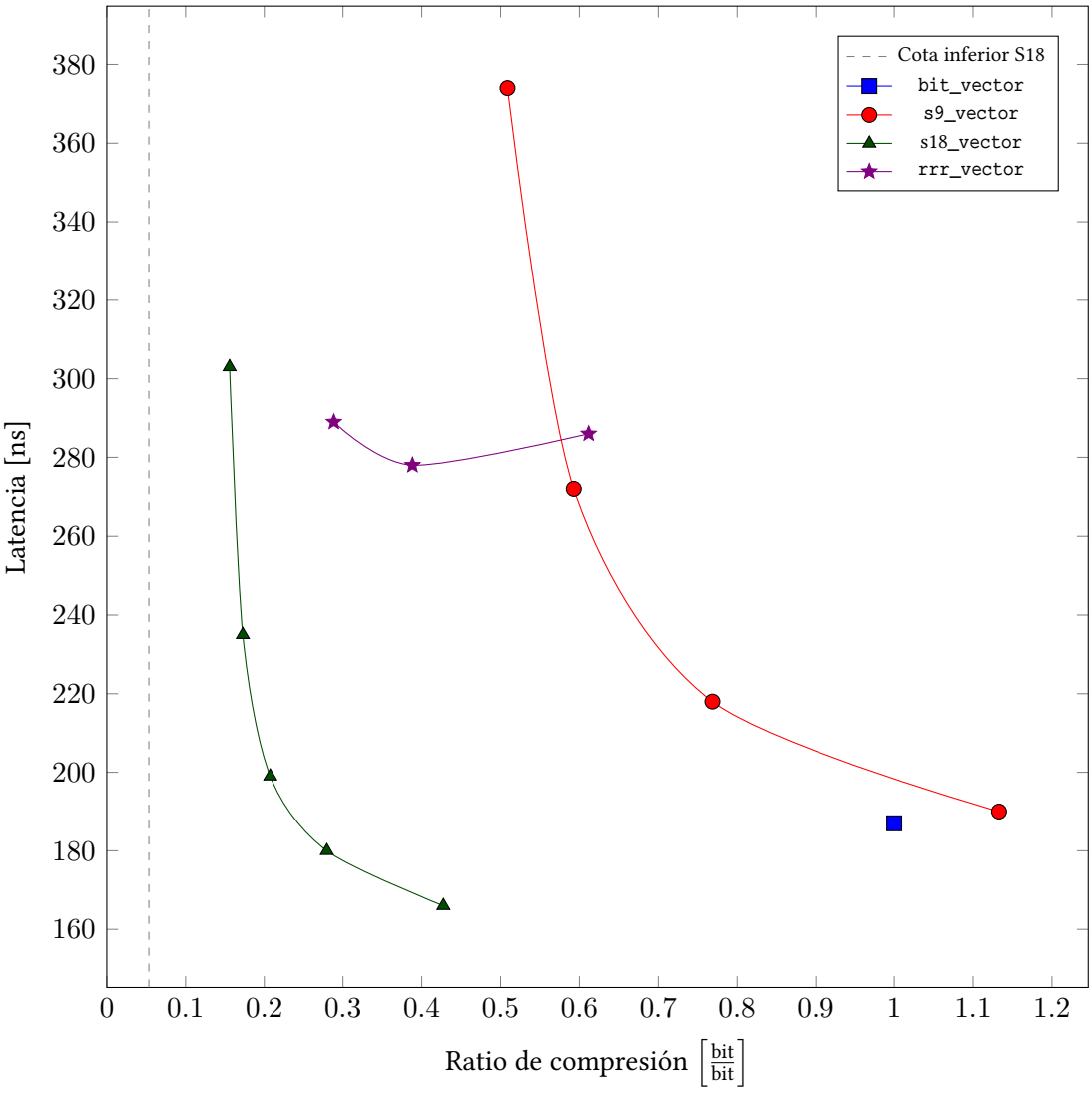


Figura B.38: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.3$. Los resutados completos se encuentran tabulados en la Tabla B.38.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	187	0.1424
s9_vector	8	164	0.8921
s9_vector	16	169	0.4563
s9_vector	32	176	0.2431
s9_vector	64	190	0.1614
s9_vector	128	218	0.1095
s9_vector	256	272	0.0845
s9_vector	512	374	0.0725
s18_vector	1	166	0.0609
s18_vector	2	180	0.0398
s18_vector	4	199	0.0296
s18_vector	8	235	0.0246
s18_vector	16	303	0.0222
s18_vector	32	433	0.0210
s18_vector	64	693	0.0205
rrr_vector	7	286	0.0871
rrr_vector	15	278	0.0553
rrr_vector	31	289	0.0411
rrr_vector	63	278	0.0406
rrr_vector	127	289	0.0540
rrr_vector	255	381	0.0752
sd_vector	-	180	0.2040

Tabla B.38: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.3$.

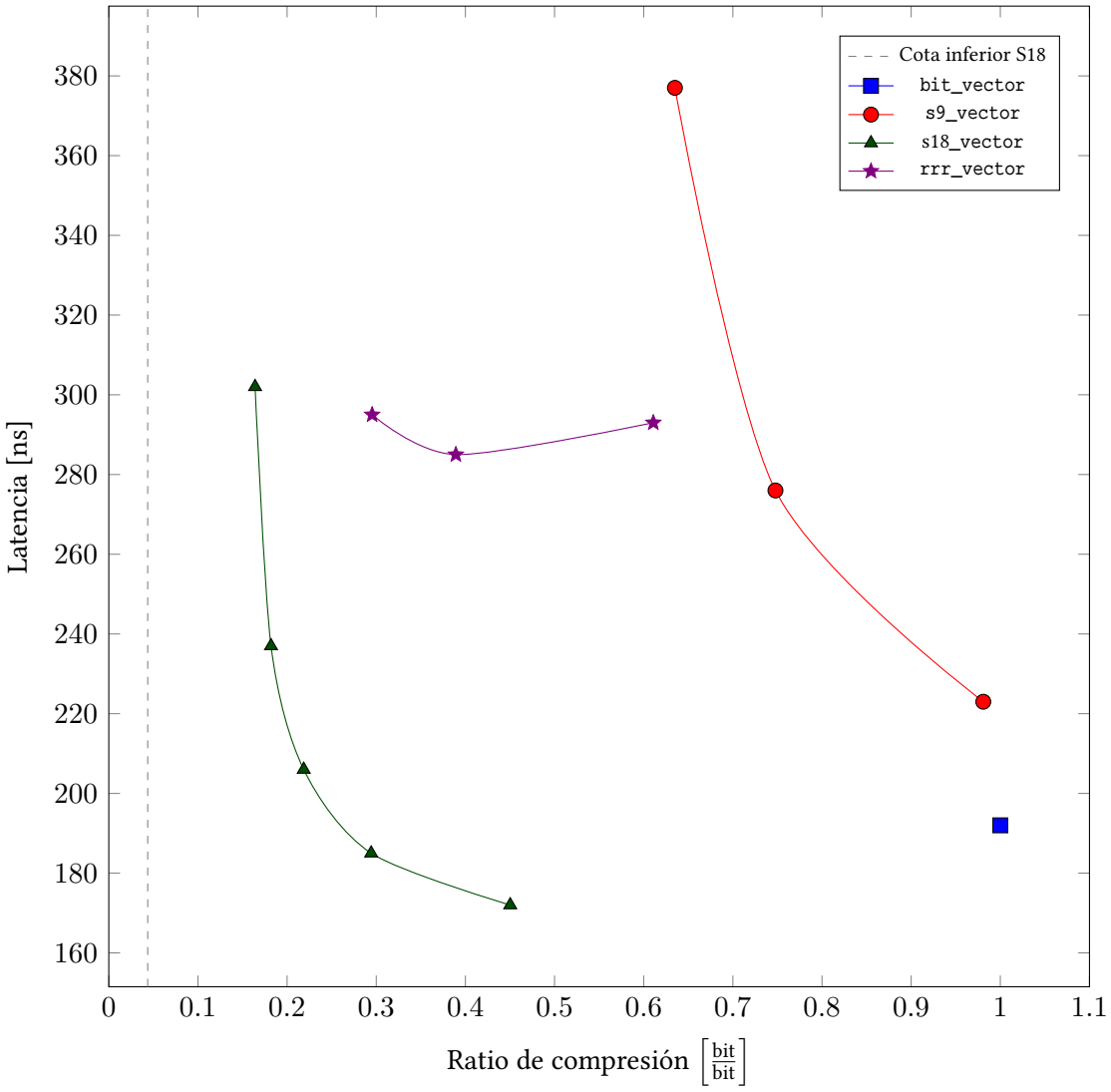


Figura B.39: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.4$. Los resutados completos se encuentran tabulados en la Tabla B.39.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	192	0.1409
s9_vector	8	174	1.1595
s9_vector	16	176	0.5902
s9_vector	32	187	0.3110
s9_vector	64	197	0.2060
s9_vector	128	223	0.1382
s9_vector	256	276	0.1054
s9_vector	512	377	0.0895
s18_vector	1	172	0.0634
s18_vector	2	185	0.0415
s18_vector	4	206	0.0308
s18_vector	8	237	0.0256
s18_vector	16	302	0.0231
s18_vector	32	431	0.0219
s18_vector	64	664	0.0213
rrr_vector	7	293	0.0861
rrr_vector	15	285	0.0549
rrr_vector	31	295	0.0416
rrr_vector	63	286	0.0423
rrr_vector	127	291	0.0593
rrr_vector	255	368	0.0844
sd_vector	-	185	0.2401

Tabla B.39: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.4$.

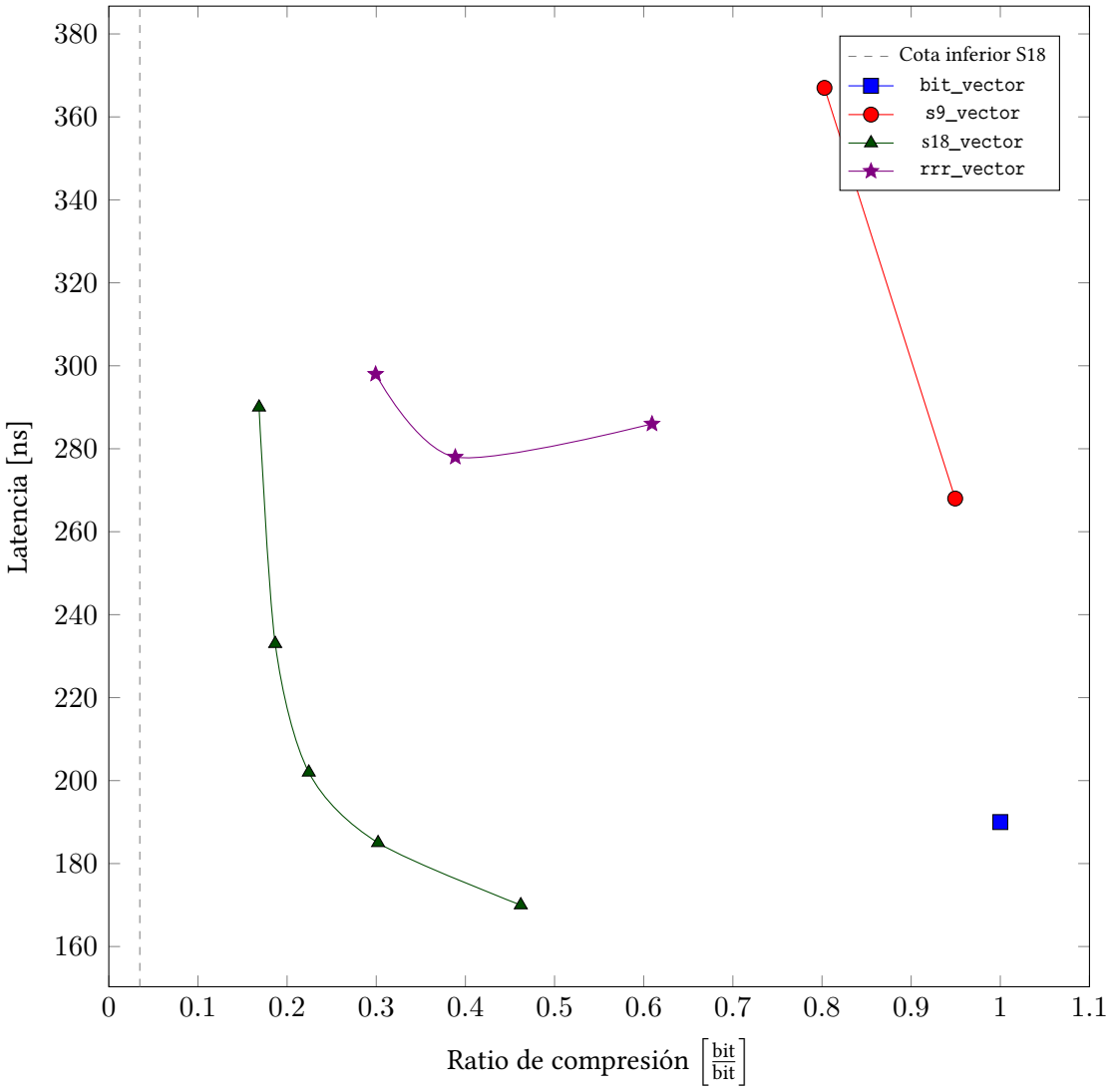


Figura B.40: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.5$. Los resutados completos se encuentran tabulados en la Tabla B.40.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	190	0.1354
s9_vector	8	170	1.4722
s9_vector	16	177	0.7466
s9_vector	32	185	0.3894
s9_vector	64	196	0.2578
s9_vector	128	222	0.1705
s9_vector	256	268	0.1286
s9_vector	512	367	0.1087
s18_vector	1	170	0.0626
s18_vector	2	185	0.0409
s18_vector	4	202	0.0304
s18_vector	8	233	0.0253
s18_vector	16	290	0.0228
s18_vector	32	409	0.0216
s18_vector	64	591	0.0210
rrr_vector	7	286	0.0825
rrr_vector	15	278	0.0526
rrr_vector	31	298	0.0405
rrr_vector	63	282	0.0420
rrr_vector	127	285	0.0593
rrr_vector	255	352	0.0880
sd_vector	-	186	0.2821

Tabla B.40: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.5$.

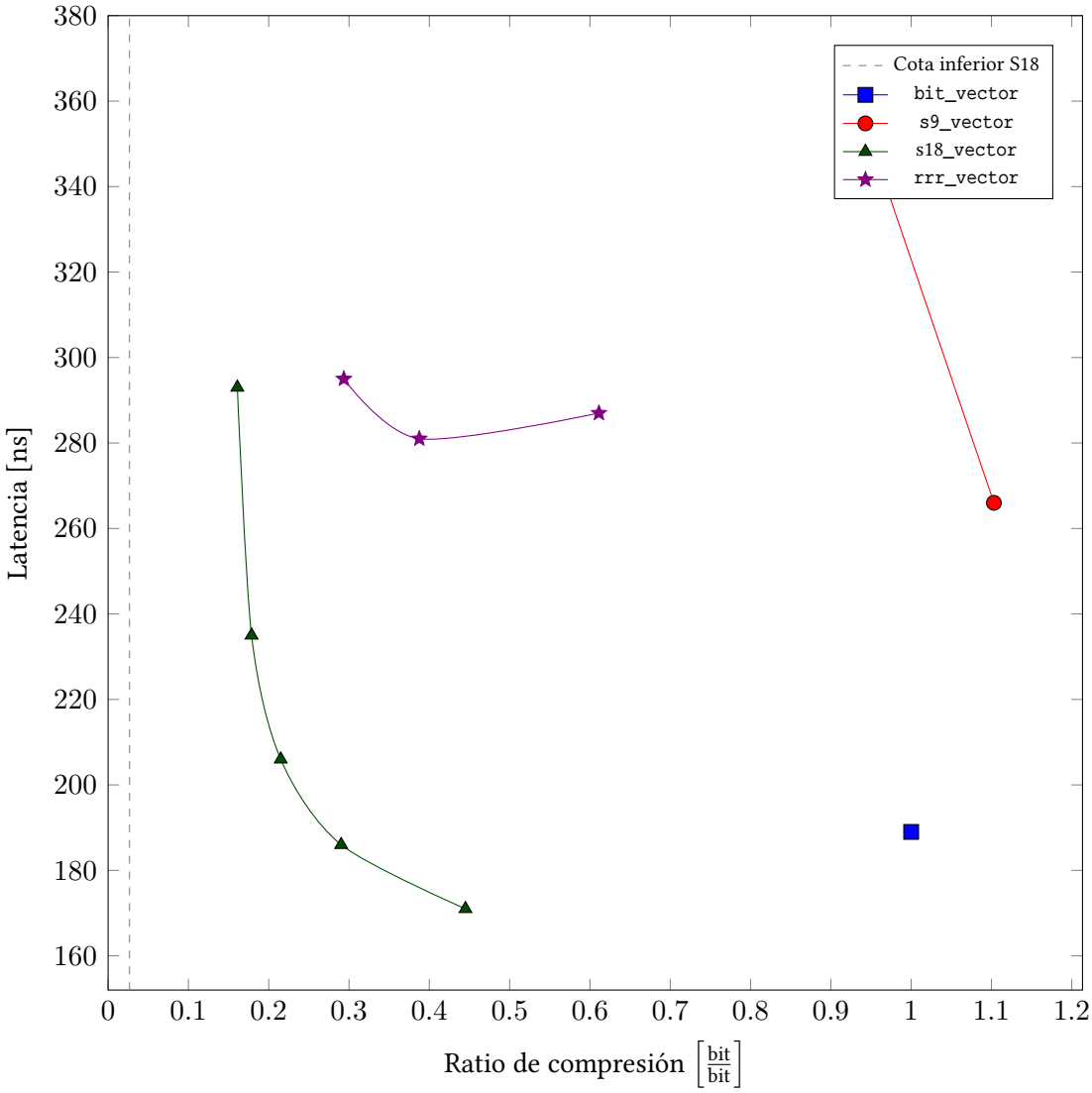


Figura B.41: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.6$. Los resutados completos se encuentran tabulados en la Tabla B.41.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	189	0.1333
s9_vector	8	172	1.7407
s9_vector	16	179	0.8799
s9_vector	32	188	0.4550
s9_vector	64	198	0.3014
s9_vector	128	219	0.1970
s9_vector	256	266	0.1470
s9_vector	512	361	0.1241
s18_vector	1	171	0.0593
s18_vector	2	186	0.0387
s18_vector	4	206	0.0287
s18_vector	8	235	0.0238
s18_vector	16	293	0.0215
s18_vector	32	412	0.0203
s18_vector	64	606	0.0197
rrr_vector	7	287	0.0815
rrr_vector	15	281	0.0517
rrr_vector	31	295	0.0391
rrr_vector	63	282	0.0406
rrr_vector	127	287	0.0563
rrr_vector	255	339	0.0837
sd_vector	-	187	0.3243

Tabla B.41: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.6$.

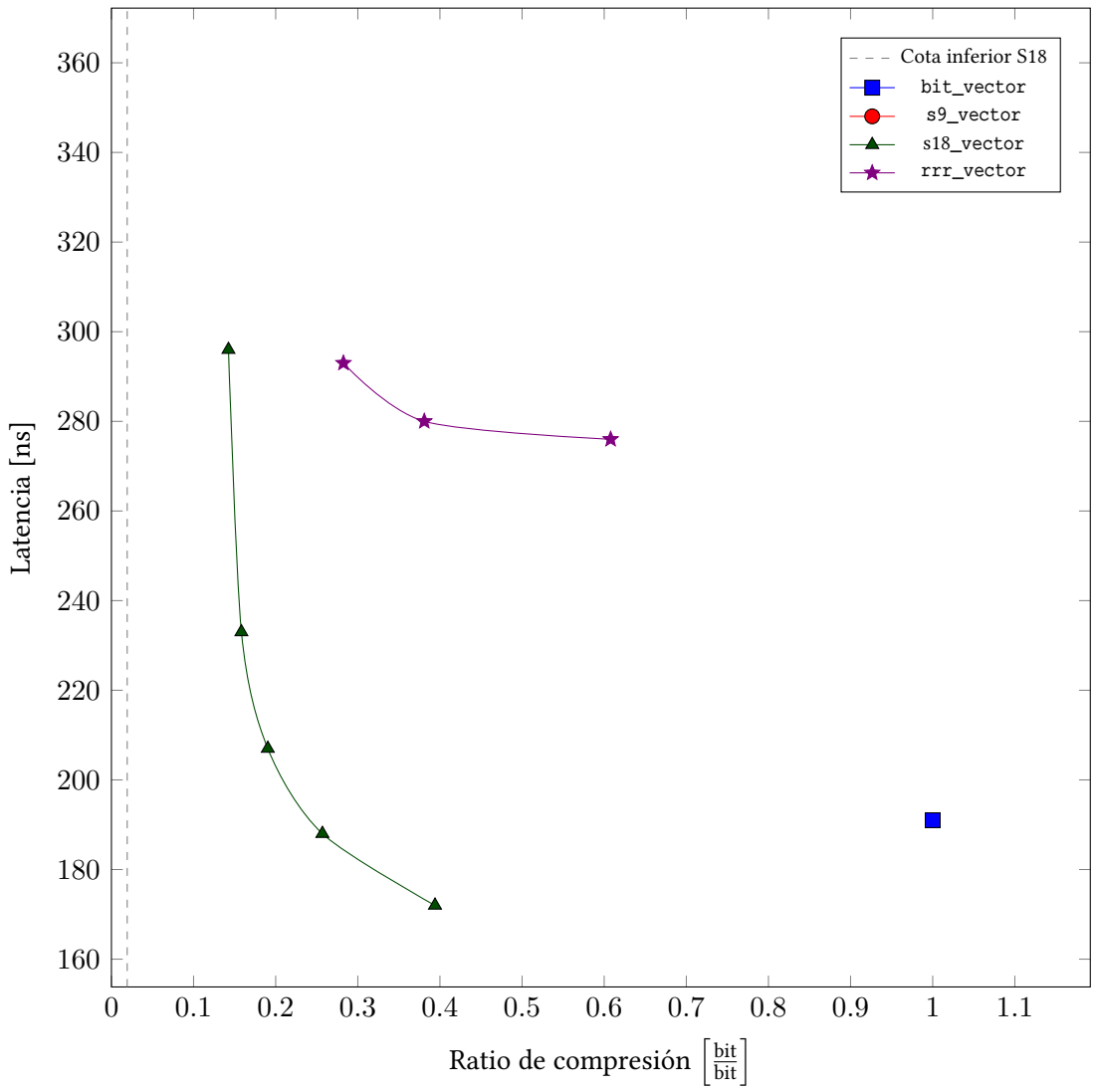


Figura B.42: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.7$. Los resutados completos se encuentran tabulados en la Tabla B.42.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	191	0.1288
s9_vector	8	175	2.0358
s9_vector	16	181	1.0258
s9_vector	32	189	0.5256
s9_vector	64	197	0.3485
s9_vector	128	216	0.2253
s9_vector	256	262	0.1653
s9_vector	512	354	0.1396
s18_vector	1	172	0.0508
s18_vector	2	188	0.0331
s18_vector	4	207	0.0245
s18_vector	8	233	0.0204
s18_vector	16	296	0.0184
s18_vector	32	406	0.0174
s18_vector	64	621	0.0169
rrr_vector	7	276	0.0783
rrr_vector	15	280	0.0491
rrr_vector	31	293	0.0364
rrr_vector	63	277	0.0364
rrr_vector	127	277	0.0499
rrr_vector	255	315	0.0729
sd_vector	-	188	0.3518

Tabla B.42: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.7$.

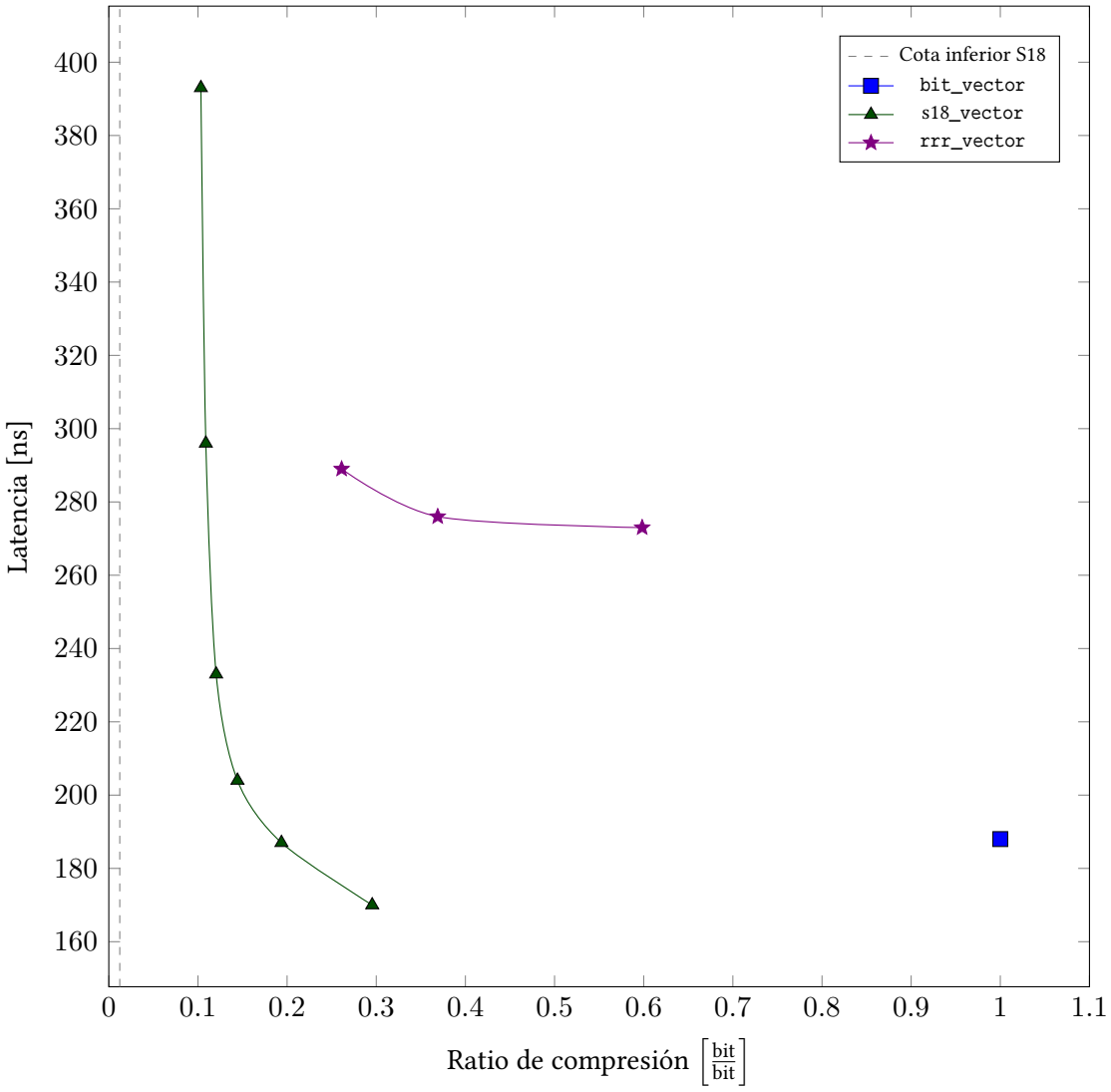


Figura B.43: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.8$. Los resutados completos se encuentran tabulados en la Tabla B.43.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	188	0.1252
s9_vector	8	176	2.3285
s9_vector	16	181	1.1701
s9_vector	32	191	0.5944
s9_vector	64	200	0.3945
s9_vector	128	221	0.2521
s9_vector	256	261	0.1821
s9_vector	512	351	0.1538
s18_vector	1	170	0.0370
s18_vector	2	187	0.0243
s18_vector	4	204	0.0181
s18_vector	8	233	0.0151
s18_vector	16	296	0.0136
s18_vector	32	393	0.0129
s18_vector	64	558	0.0126
rrr_vector	7	273	0.0749
rrr_vector	15	276	0.0462
rrr_vector	31	289	0.0327
rrr_vector	63	275	0.0303
rrr_vector	127	271	0.0385
rrr_vector	255	296	0.0561
sd_vector	-	188	0.3790

Tabla B.43: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.8$.

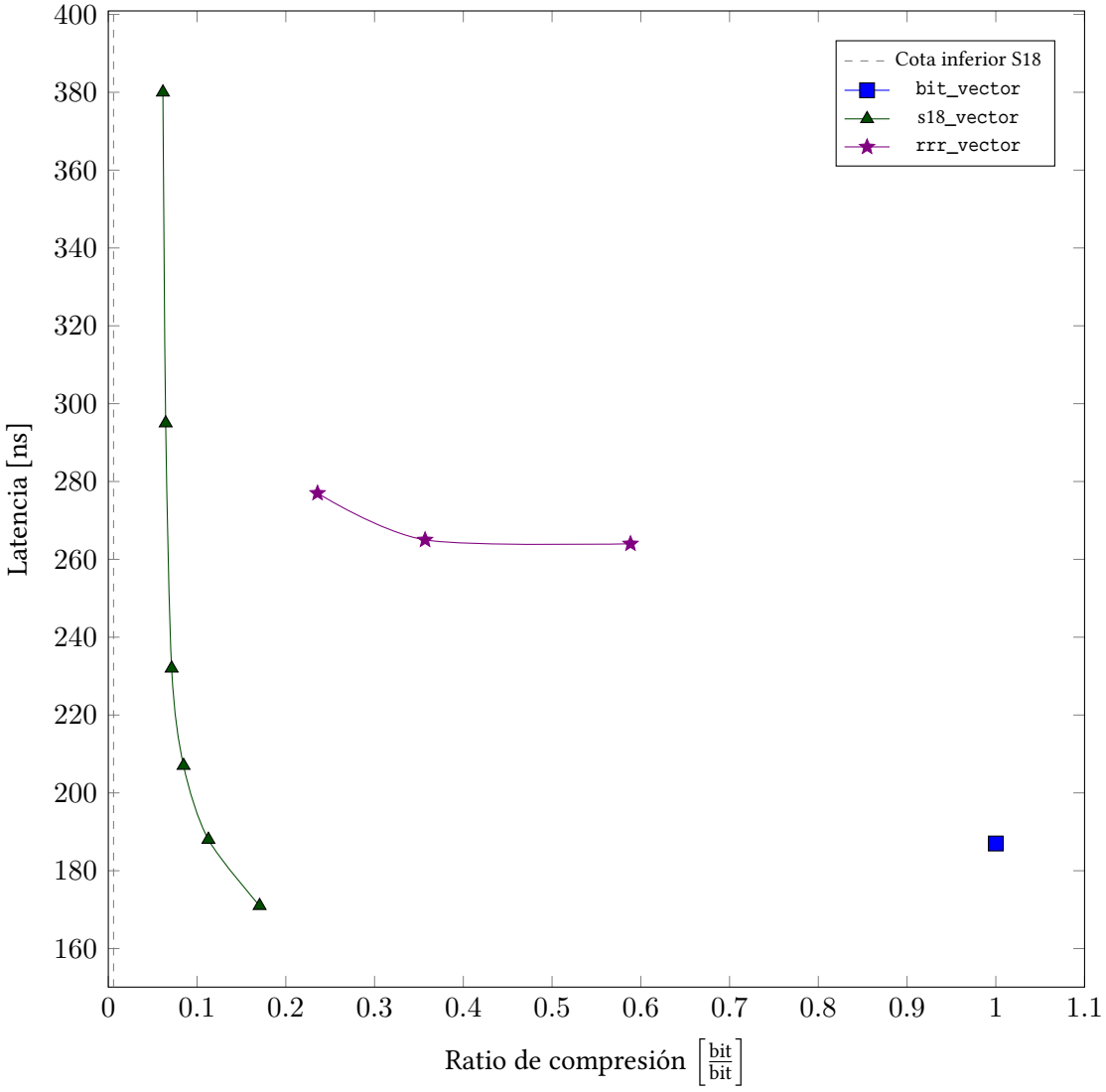


Figura B.44: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.9$. Los resutados completos se encuentran tabulados en la Tabla B.44.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	187	0.1227
s9_vector	8	173	2.5763
s9_vector	16	178	1.2914
s9_vector	32	190	0.6509
s9_vector	64	200	0.4328
s9_vector	128	217	0.2735
s9_vector	256	261	0.1948
s9_vector	512	345	0.1652
s18_vector	1	171	0.0209
s18_vector	2	188	0.0138
s18_vector	4	207	0.0104
s18_vector	8	232	0.0088
s18_vector	16	295	0.0079
s18_vector	32	380	0.0075
s18_vector	64	604	0.0074
rrr_vector	7	264	0.0722
rrr_vector	15	265	0.0438
rrr_vector	31	277	0.0289
rrr_vector	63	272	0.0231
rrr_vector	127	269	0.0256
rrr_vector	255	278	0.0344
sd_vector	-	190	0.3254

Tabla B.44: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.9$.

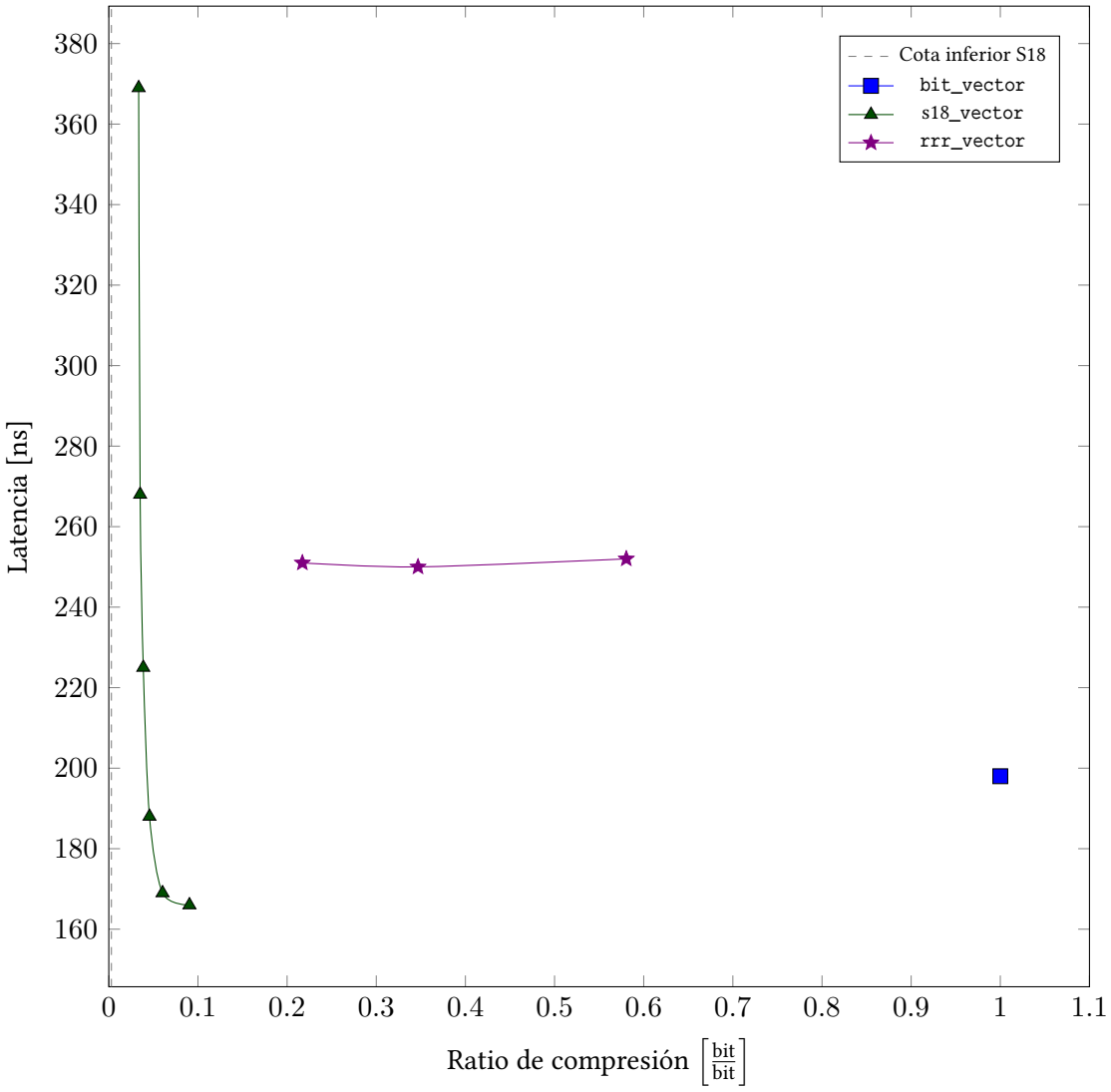


Figura B.45: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.95$. Los resutados completos se encuentran tabulados en la Tabla B.45.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	198	0.1208
s9_vector	8	184	2.7266
s9_vector	16	174	1.3651
s9_vector	32	189	0.6854
s9_vector	64	196	0.4562
s9_vector	128	213	0.2867
s9_vector	256	250	0.2023
s9_vector	512	333	0.1723
s18_vector	1	166	0.0109
s18_vector	2	169	0.0073
s18_vector	4	188	0.0055
s18_vector	8	225	0.0047
s18_vector	16	268	0.0042
s18_vector	32	369	0.0040
s18_vector	64	546	0.0040
rrr_vector	7	252	0.0701
rrr_vector	15	250	0.0419
rrr_vector	31	251	0.0262
rrr_vector	63	253	0.0187
rrr_vector	127	253	0.0171
rrr_vector	255	257	0.0203
sd_vector	-	181	0.3394

Tabla B.45: Rendimiento de la operación SELECT. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.95$.

B.4. Successor

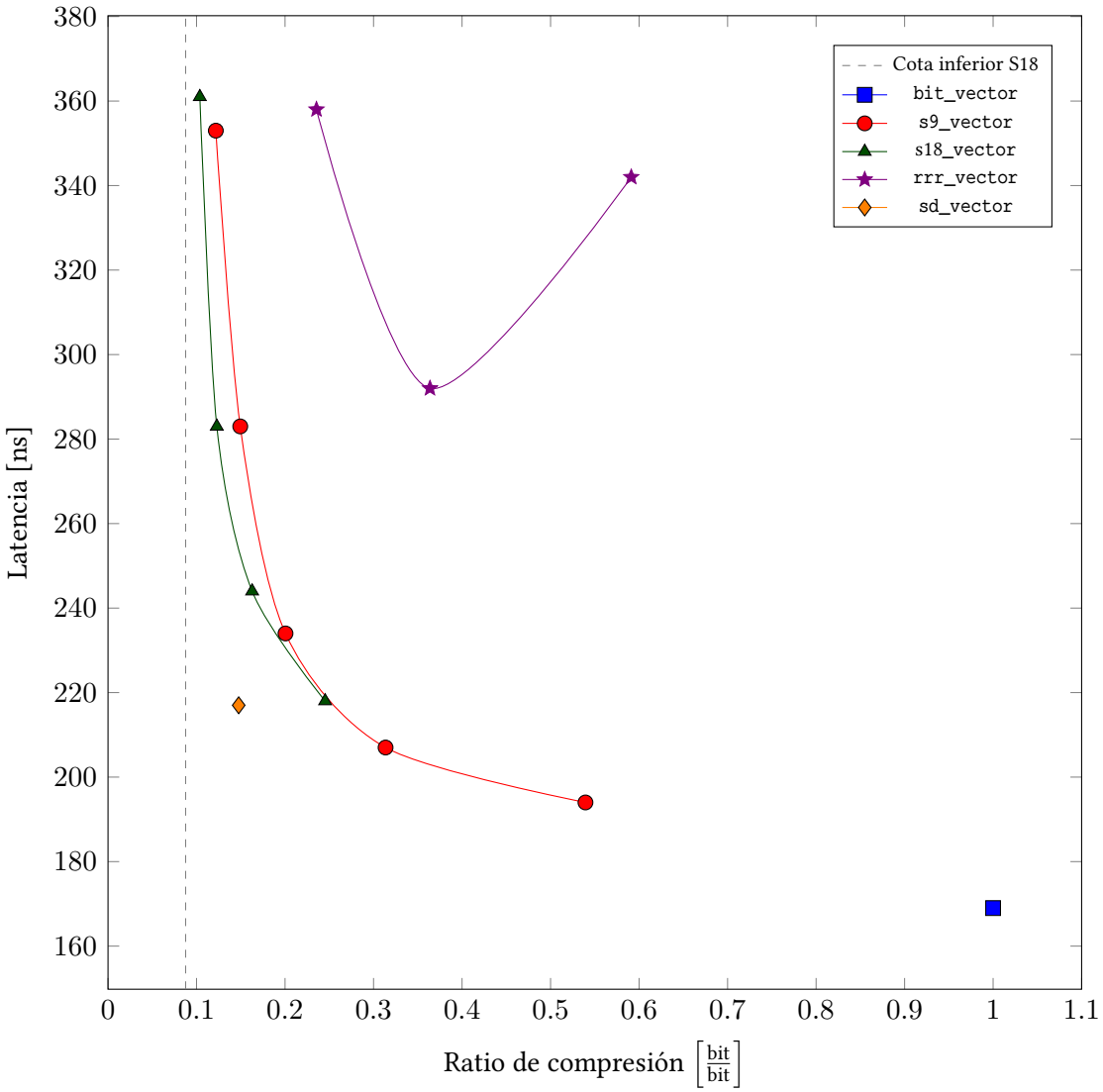


Figura B.46: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.01$. Los resutados completos se encuentran tabulados en la Tabla B.46.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	169	0.1512
s9_vector	8	194	0.0815
s9_vector	16	207	0.0474
s9_vector	32	234	0.0303
s9_vector	64	283	0.0226
s9_vector	128	353	0.0184
s9_vector	256	473	0.0164
s9_vector	512	705	0.0154
s18_vector	1	218	0.0371
s18_vector	2	244	0.0246
s18_vector	4	283	0.0186
s18_vector	8	361	0.0157
s18_vector	16	493	0.0143
s18_vector	32	783	0.0136
s18_vector	64	1328	0.0133
rrr_vector	7	342	0.0894
rrr_vector	15	292	0.0550
rrr_vector	31	358	0.0356
rrr_vector	63	358	0.0249
rrr_vector	127	395	0.0196
rrr_vector	255	526	0.0175
sd_vector	-	217	0.0223

Tabla B.46: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.01$.

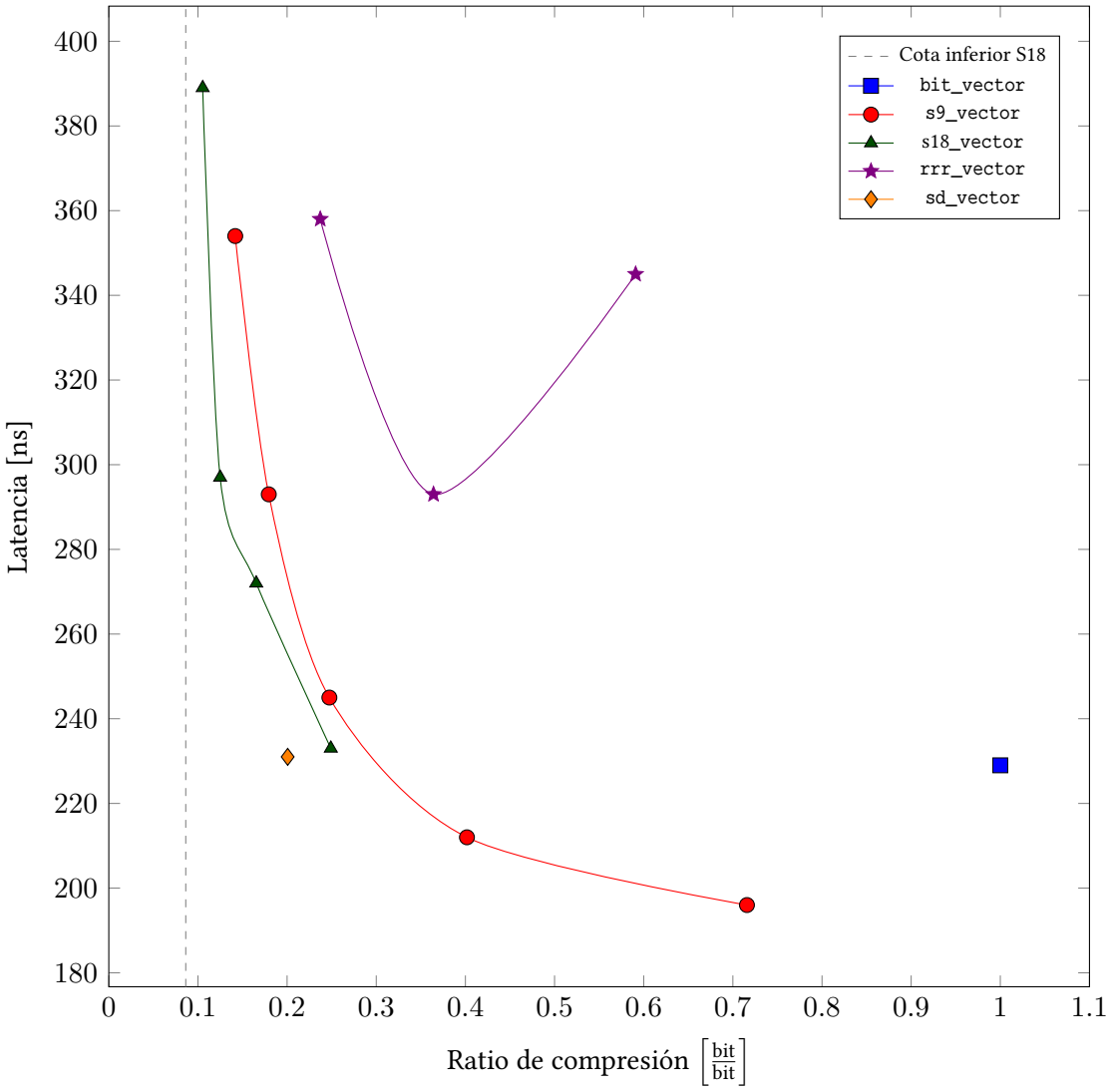


Figura B.47: Rendimiento de la operación SUCCESSOR. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.02$. Los resutados completos se encuentran tabulados en la Tabla B.47.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	229	0.1532
s9_vector	8	196	0.1096
s9_vector	16	212	0.0615
s9_vector	32	245	0.0379
s9_vector	64	293	0.0275
s9_vector	128	354	0.0217
s9_vector	256	483	0.0189
s9_vector	512	708	0.0175
s18_vector	1	233	0.0382
s18_vector	2	272	0.0253
s18_vector	4	297	0.0191
s18_vector	8	389	0.0161
s18_vector	16	506	0.0147
s18_vector	32	773	0.0140
s18_vector	64	1250	0.0137
rrr_vector	7	345	0.0905
rrr_vector	15	293	0.0558
rrr_vector	31	358	0.0363
rrr_vector	63	358	0.0260
rrr_vector	127	403	0.0218
rrr_vector	255	562	0.0205
sd_vector	-	231	0.0307

Tabla B.47: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.02$.

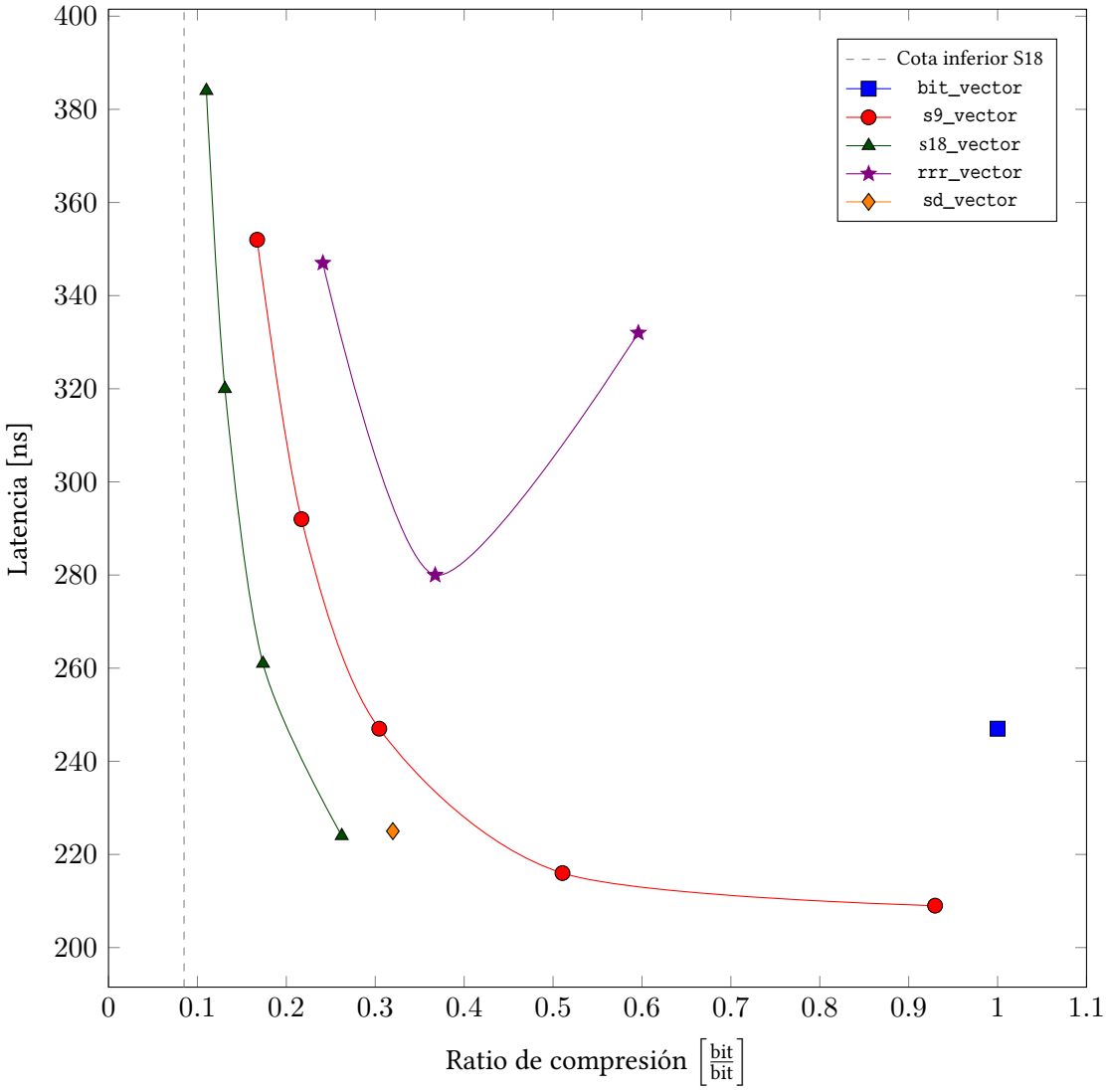


Figura B.48: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.03$. Los resutados completos se encuentran tabulados en la Tabla B.48.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	247	0.1506
s9_vector	8	209	0.1400
s9_vector	16	216	0.0769
s9_vector	32	247	0.0459
s9_vector	64	292	0.0327
s9_vector	128	352	0.0252
s9_vector	256	465	0.0216
s9_vector	512	675	0.0197
s18_vector	1	224	0.0395
s18_vector	2	261	0.0262
s18_vector	4	320	0.0197
s18_vector	8	384	0.0166
s18_vector	16	501	0.0151
s18_vector	32	816	0.0144
s18_vector	64	1379	0.0140
rrr_vector	7	332	0.0897
rrr_vector	15	280	0.0553
rrr_vector	31	347	0.0363
rrr_vector	63	353	0.0265
rrr_vector	127	394	0.0232
rrr_vector	255	560	0.0233
sd_vector	-	225	0.0482

Tabla B.48: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.03$.

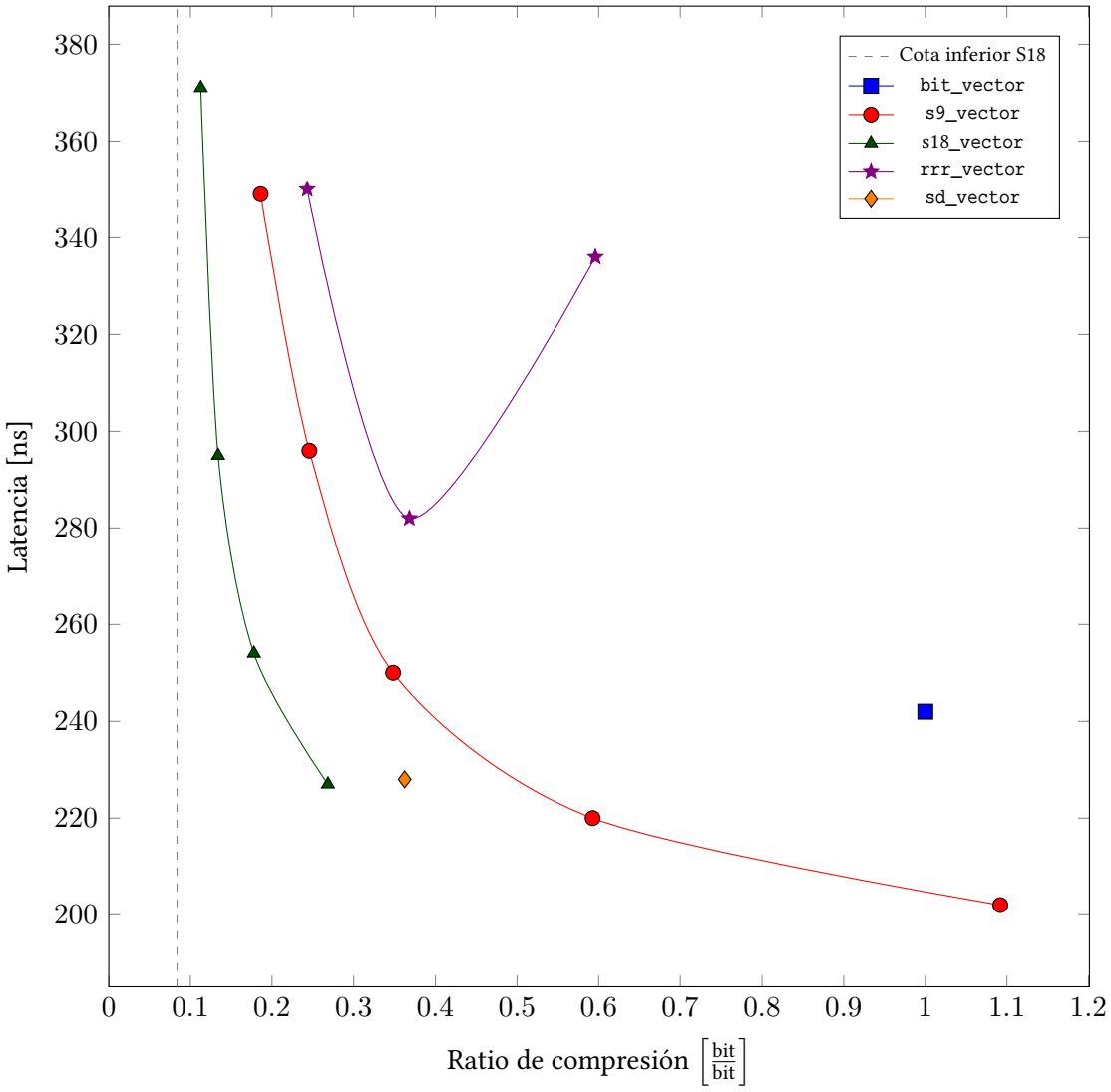


Figura B.49: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.04$. Los resutados completos se encuentran tabulados en la Tabla B.49.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	242	0.1509
s9_vector	8	202	0.1648
s9_vector	16	220	0.0894
s9_vector	32	250	0.0526
s9_vector	64	296	0.0371
s9_vector	128	349	0.0281
s9_vector	256	462	0.0238
s9_vector	512	672	0.0216
s18_vector	1	227	0.0405
s18_vector	2	254	0.0268
s18_vector	4	295	0.0202
s18_vector	8	371	0.0170
s18_vector	16	491	0.0155
s18_vector	32	776	0.0147
s18_vector	64	1343	0.0144
rrr_vector	7	336	0.0900
rrr_vector	15	282	0.0556
rrr_vector	31	350	0.0367
rrr_vector	63	359	0.0273
rrr_vector	127	396	0.0247
rrr_vector	255	573	0.0257
sd_vector	-	228	0.0547

Tabla B.49: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.04$.

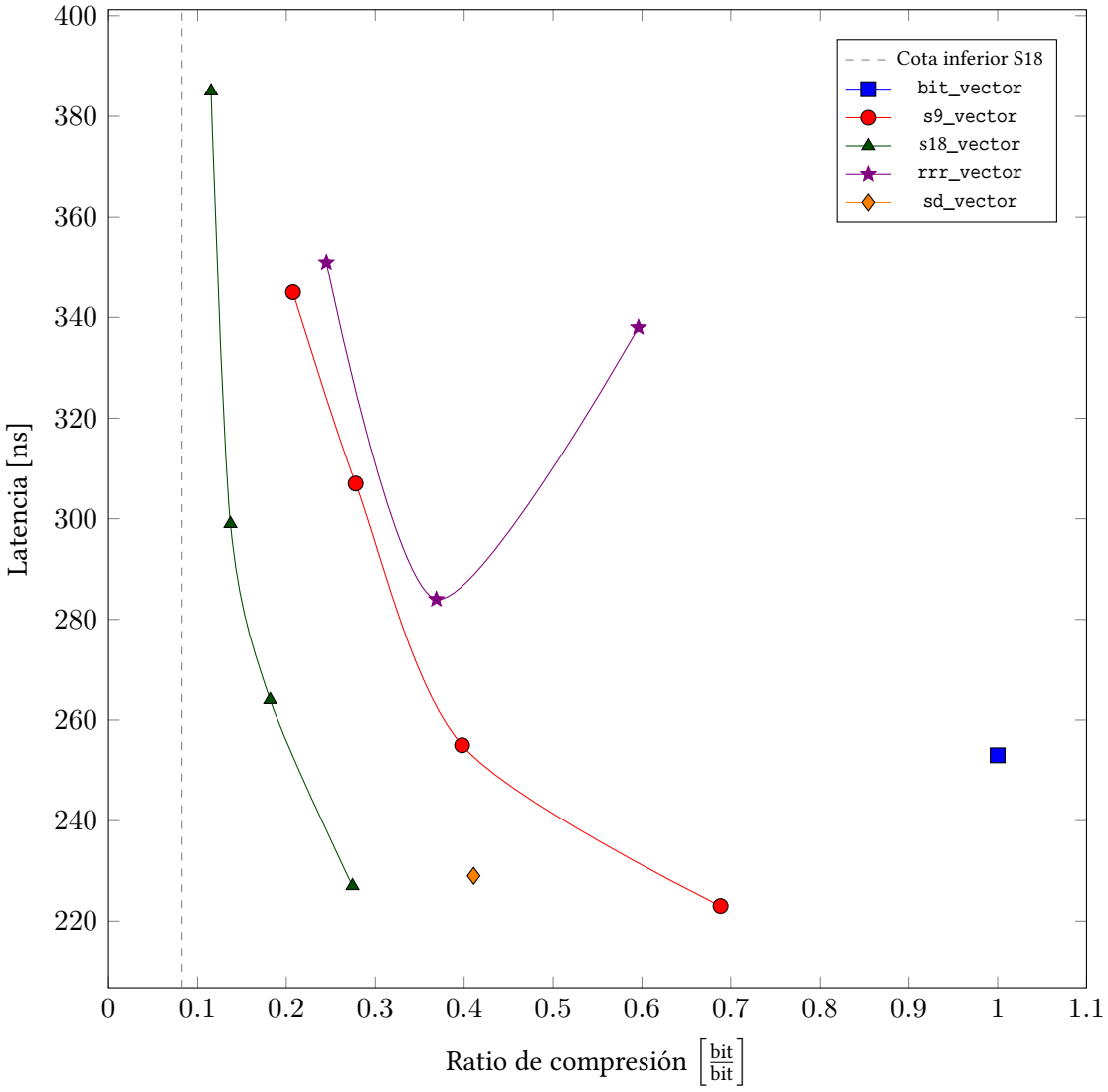


Figura B.50: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.05$. Los resutados completos se encuentran tabulados en la Tabla B.50.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	253	0.1514
s9_vector	8	210	0.1939
s9_vector	16	223	0.1042
s9_vector	32	255	0.0602
s9_vector	64	307	0.0421
s9_vector	128	345	0.0314
s9_vector	256	457	0.0263
s9_vector	512	671	0.0238
s18_vector	1	227	0.0416
s18_vector	2	264	0.0275
s18_vector	4	299	0.0207
s18_vector	8	385	0.0174
s18_vector	16	511	0.0159
s18_vector	32	798	0.0151
s18_vector	64	1271	0.0147
rrr_vector	7	338	0.0902
rrr_vector	15	284	0.0558
rrr_vector	31	351	0.0371
rrr_vector	63	356	0.0281
rrr_vector	127	406	0.0267
rrr_vector	255	602	0.0286
sd_vector	-	229	0.0622

Tabla B.50: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.05$.

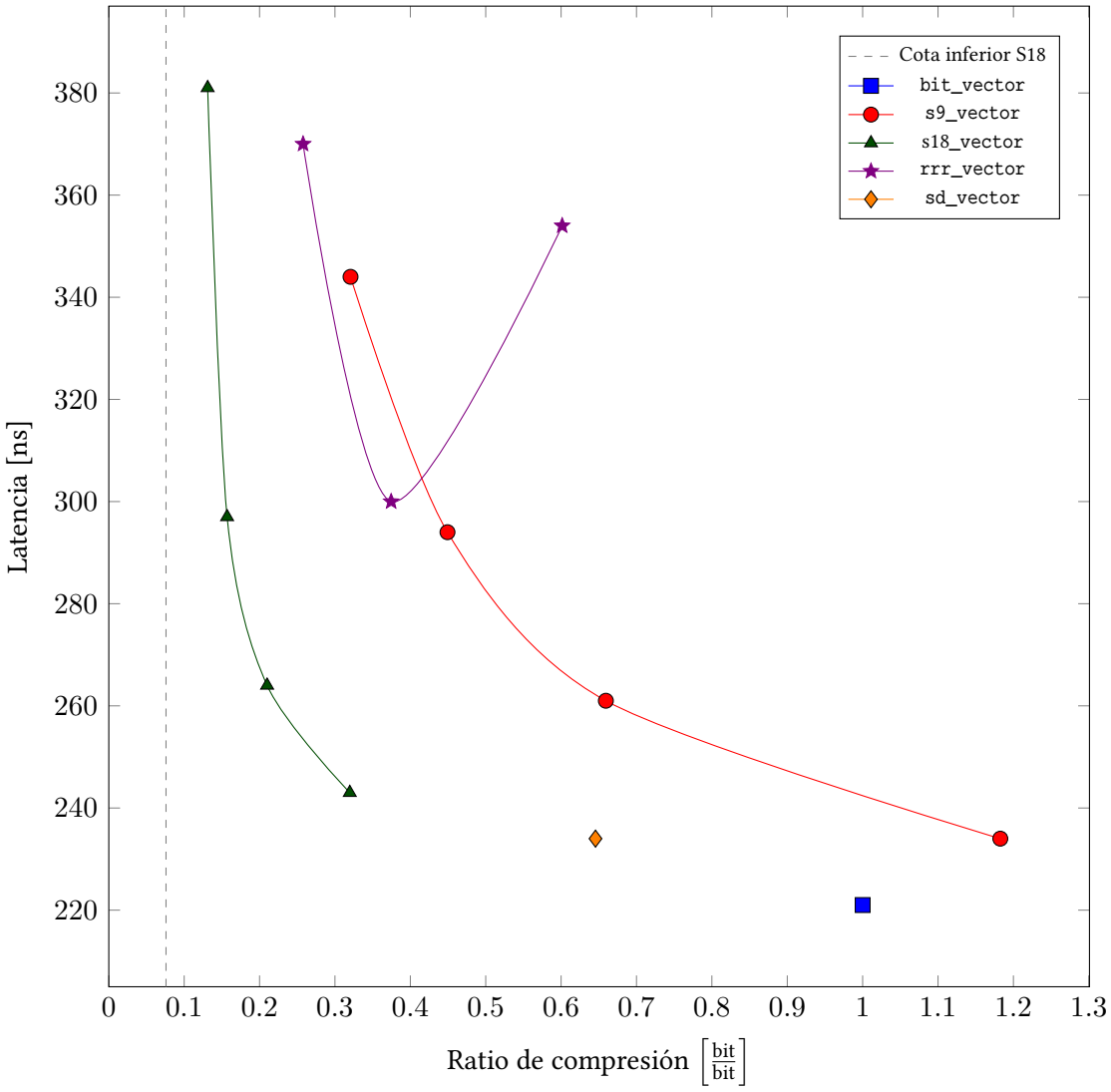


Figura B.51: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.1$. Los resutados completos se encuentran tabulados en la Tabla B.51.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	221	0.1495
s9_vector	8	222	0.3375
s9_vector	16	234	0.1768
s9_vector	32	261	0.0986
s9_vector	64	294	0.0672
s9_vector	128	344	0.0479
s9_vector	256	446	0.0389
s9_vector	512	648	0.0342
s18_vector	1	243	0.0478
s18_vector	2	264	0.0314
s18_vector	4	297	0.0235
s18_vector	8	381	0.0196
s18_vector	16	523	0.0177
s18_vector	32	804	0.0168
s18_vector	64	1333	0.0164
rrr_vector	7	354	0.0899
rrr_vector	15	300	0.0560
rrr_vector	31	370	0.0385
rrr_vector	63	368	0.0316
rrr_vector	127	434	0.0345
rrr_vector	255	699	0.0412
sd_vector	-	234	0.0965

Tabla B.51: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.1$.

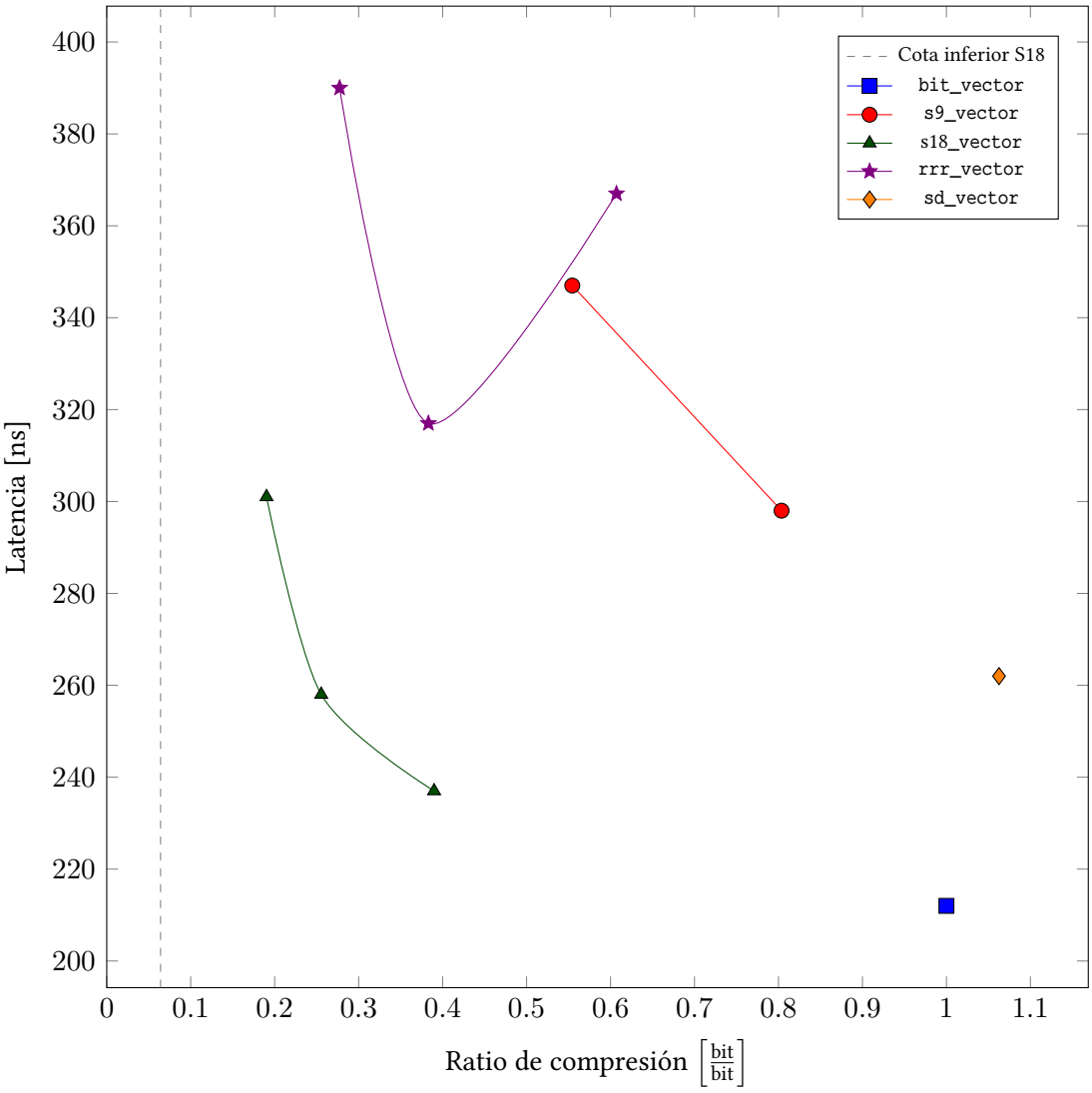


Figura B.52: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.2$. Los resutados completos se encuentran tabulados en la Tabla B.52.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	212	0.1454
s9_vector	8	242	0.6279
s9_vector	16	248	0.3231
s9_vector	32	268	0.1748
s9_vector	64	298	0.1169
s9_vector	128	347	0.0806
s9_vector	256	451	0.0635
s9_vector	512	631	0.0549
s18_vector	1	237	0.0567
s18_vector	2	258	0.0371
s18_vector	4	301	0.0277
s18_vector	8	419	0.0231
s18_vector	16	533	0.0209
s18_vector	32	855	0.0198
s18_vector	64	1493	0.0193
rrr_vector	7	367	0.0883
rrr_vector	15	317	0.0557
rrr_vector	31	390	0.0403
rrr_vector	63	381	0.0373
rrr_vector	127	482	0.0474
rrr_vector	255	947	0.0626
sd_vector	-	262	0.1545

Tabla B.52: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.2$.

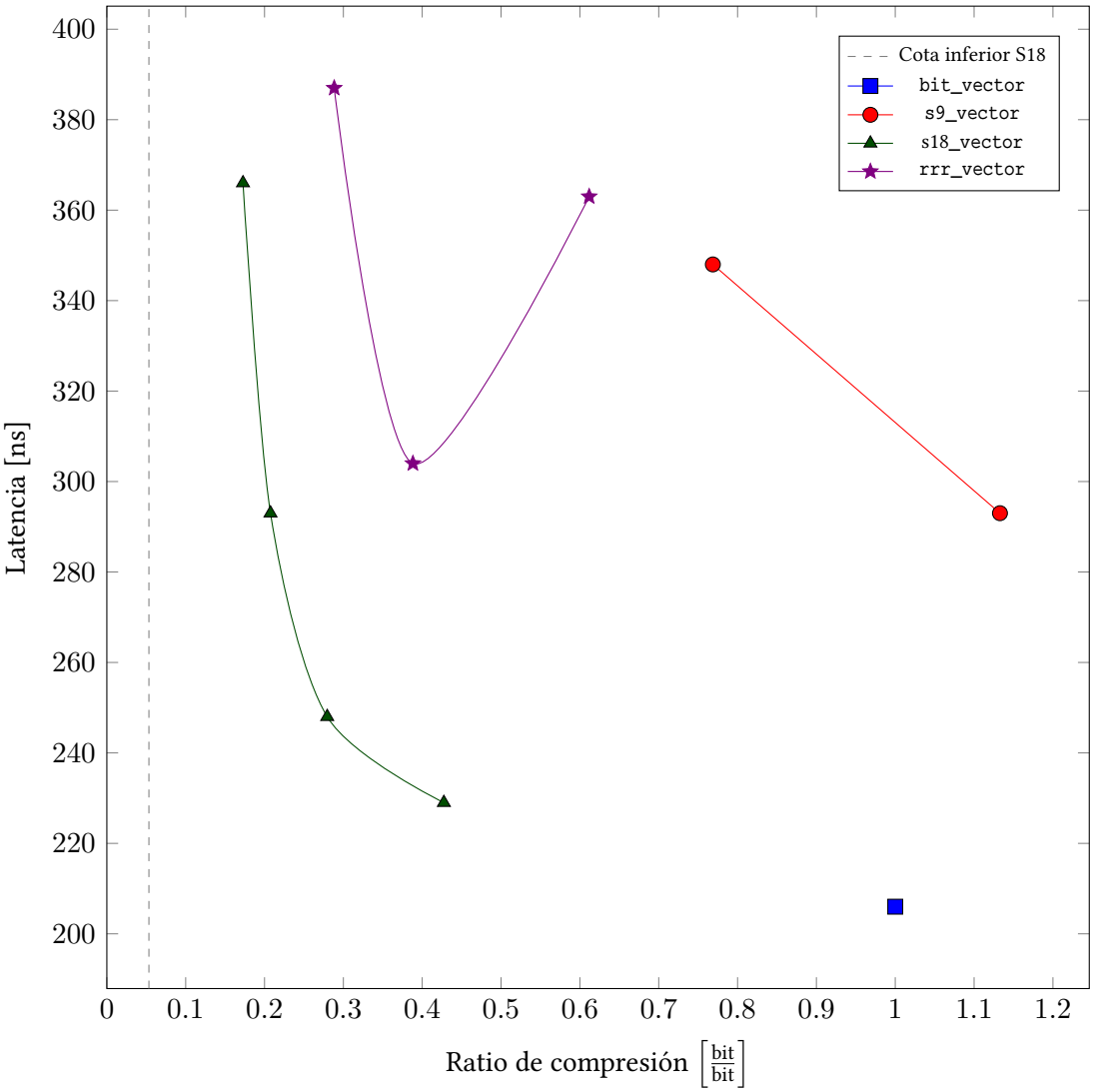


Figura B.53: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.3$. Los resutados completos se encuentran tabulados en la Tabla B.53.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	206	0.1424
s9_vector	8	253	0.8921
s9_vector	16	257	0.4563
s9_vector	32	267	0.2431
s9_vector	64	293	0.1614
s9_vector	128	348	0.1095
s9_vector	256	441	0.0845
s9_vector	512	626	0.0725
s18_vector	1	229	0.0609
s18_vector	2	248	0.0398
s18_vector	4	293	0.0296
s18_vector	8	366	0.0246
s18_vector	16	513	0.0222
s18_vector	32	823	0.0210
s18_vector	64	1348	0.0205
rrr_vector	7	363	0.0871
rrr_vector	15	304	0.0553
rrr_vector	31	387	0.0411
rrr_vector	63	382	0.0406
rrr_vector	127	464	0.0540
rrr_vector	255	967	0.0752
sd_vector	-	242	0.2040

Tabla B.53: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.3$.

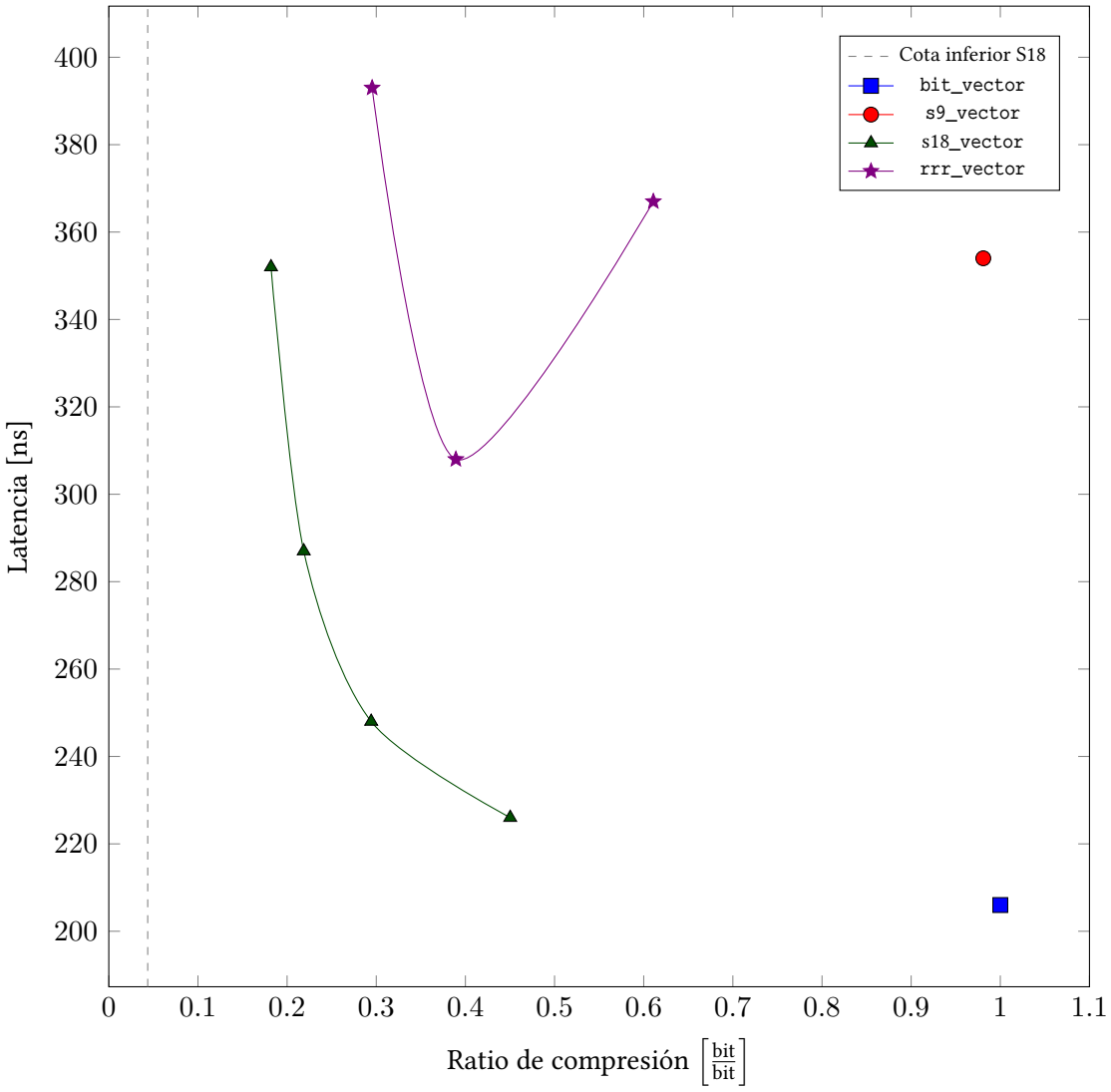


Figura B.54: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.4$. Los resutados completos se encuentran tabulados en la Tabla B.54.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	206	0.1409
s9_vector	8	303	1.1595
s9_vector	16	273	0.5902
s9_vector	32	292	0.3110
s9_vector	64	327	0.2060
s9_vector	128	354	0.1382
s9_vector	256	450	0.1054
s9_vector	512	695	0.0895
s18_vector	1	226	0.0634
s18_vector	2	248	0.0415
s18_vector	4	287	0.0308
s18_vector	8	352	0.0256
s18_vector	16	490	0.0231
s18_vector	32	802	0.0219
s18_vector	64	1205	0.0213
rrr_vector	7	367	0.0861
rrr_vector	15	308	0.0549
rrr_vector	31	393	0.0416
rrr_vector	63	390	0.0423
rrr_vector	127	479	0.0593
rrr_vector	255	1067	0.0844
sd_vector	-	251	0.2401

Tabla B.54: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.4$.

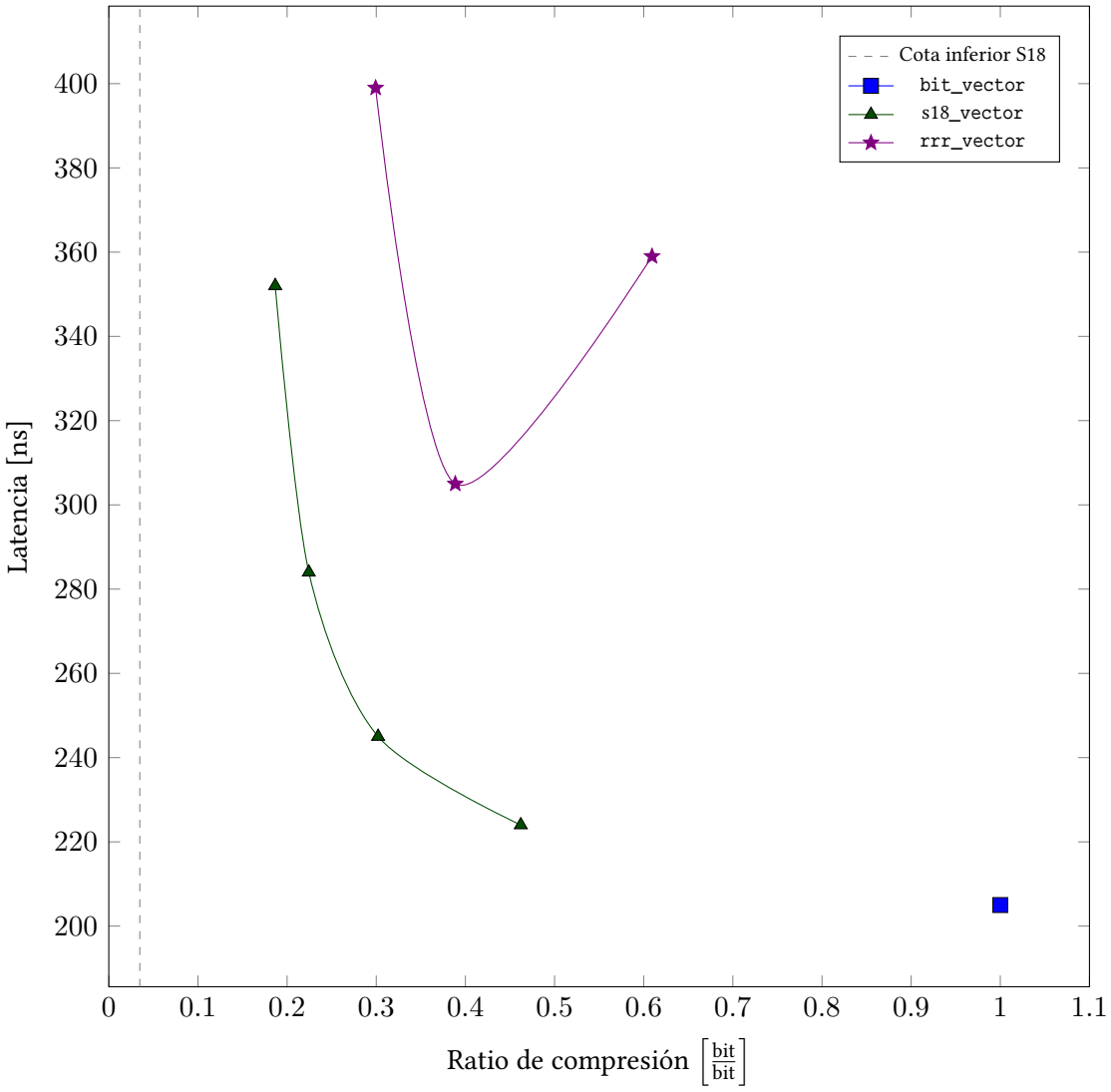


Figura B.55: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.5$. Los resutados completos se encuentran tabulados en la Tabla B.55.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	205	0.1354
s9_vector	8	269	1.4722
s9_vector	16	275	0.7466
s9_vector	32	287	0.3894
s9_vector	64	309	0.2578
s9_vector	128	358	0.1705
s9_vector	256	456	0.1286
s9_vector	512	611	0.1087
s18_vector	1	224	0.0626
s18_vector	2	245	0.0409
s18_vector	4	284	0.0304
s18_vector	8	352	0.0253
s18_vector	16	470	0.0228
s18_vector	32	714	0.0216
s18_vector	64	1202	0.0210
rrr_vector	7	359	0.0825
rrr_vector	15	305	0.0526
rrr_vector	31	399	0.0405
rrr_vector	63	393	0.0420
rrr_vector	127	484	0.0593
rrr_vector	255	1097	0.0880
sd_vector	-	248	0.2821

Tabla B.55: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.5$.

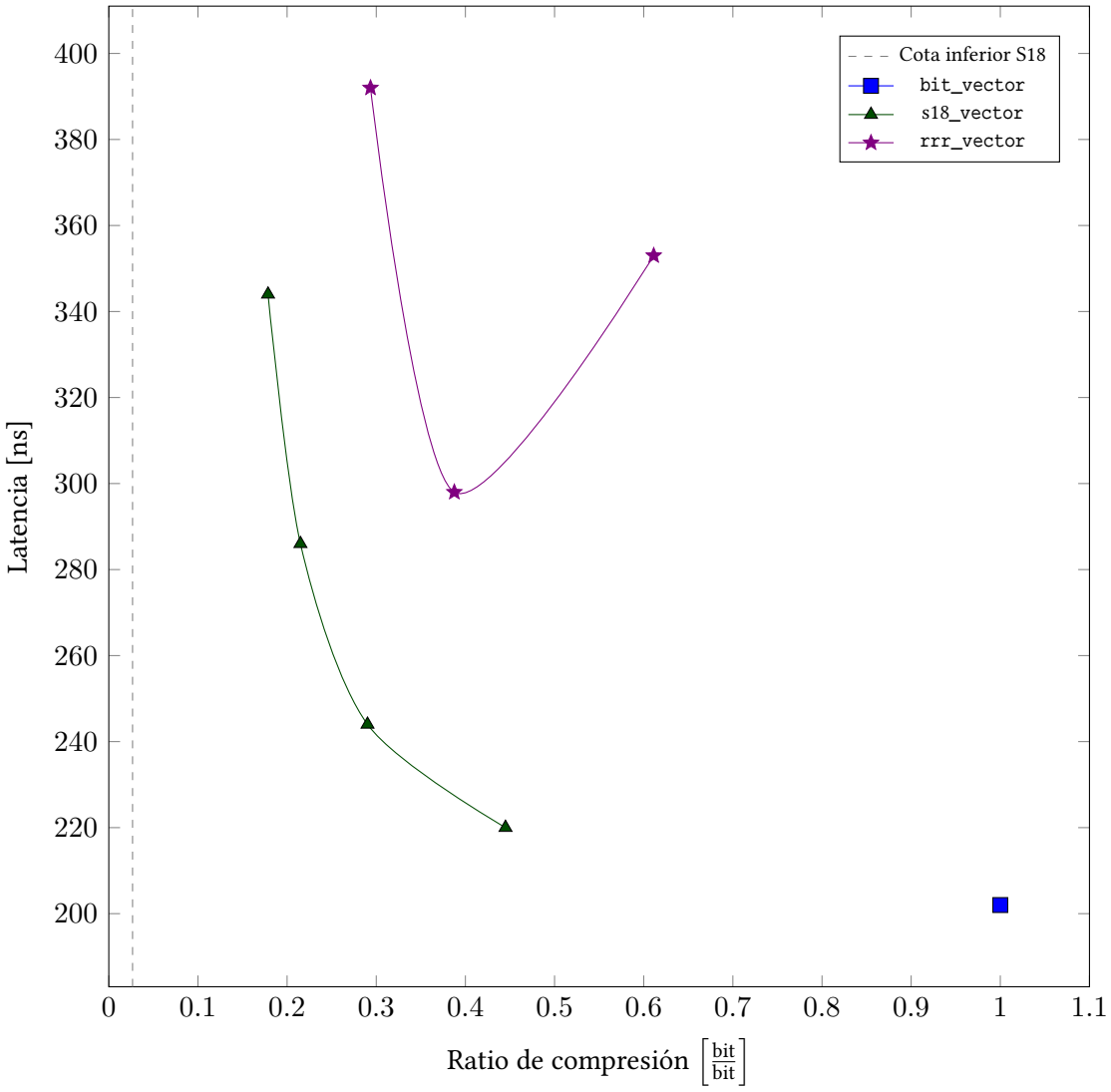


Figura B.56: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.6$. Los resutados completos se encuentran tabulados en la Tabla B.56.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	202	0.1333
s9_vector	8	280	1.7407
s9_vector	16	285	0.8799
s9_vector	32	301	0.4550
s9_vector	64	324	0.3014
s9_vector	128	361	0.1970
s9_vector	256	450	0.1470
s9_vector	512	604	0.1241
s18_vector	1	220	0.0593
s18_vector	2	244	0.0387
s18_vector	4	286	0.0287
s18_vector	8	344	0.0238
s18_vector	16	471	0.0215
s18_vector	32	689	0.0203
s18_vector	64	1111	0.0197
rrr_vector	7	353	0.0815
rrr_vector	15	298	0.0517
rrr_vector	31	392	0.0391
rrr_vector	63	385	0.0406
rrr_vector	127	482	0.0563
rrr_vector	255	1109	0.0837
sd_vector	-	250	0.3243

Tabla B.56: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.6$.

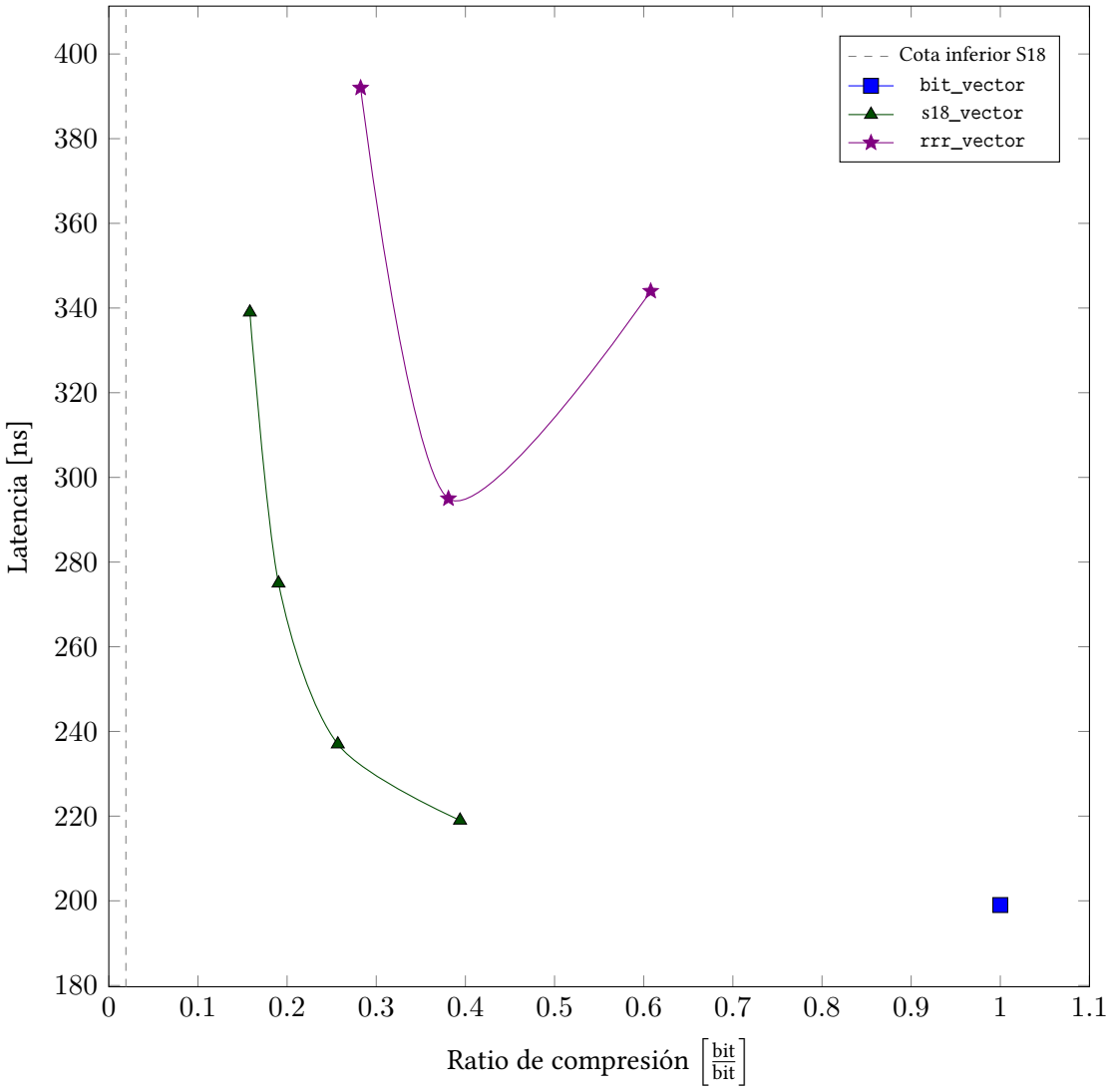


Figura B.57: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.7$. Los resutados completos se encuentran tabulados en la Tabla B.57.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	199	0.1288
s9_vector	8	289	2.0358
s9_vector	16	292	1.0258
s9_vector	32	304	0.5256
s9_vector	64	330	0.3485
s9_vector	128	363	0.2253
s9_vector	256	453	0.1653
s9_vector	512	600	0.1396
s18_vector	1	219	0.0508
s18_vector	2	237	0.0331
s18_vector	4	275	0.0245
s18_vector	8	339	0.0204
s18_vector	16	458	0.0184
s18_vector	32	701	0.0174
s18_vector	64	1050	0.0169
rrr_vector	7	344	0.0783
rrr_vector	15	295	0.0491
rrr_vector	31	392	0.0364
rrr_vector	63	382	0.0364
rrr_vector	127	469	0.0499
rrr_vector	255	1038	0.0729
sd_vector	-	247	0.3518

Tabla B.57: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.7$.

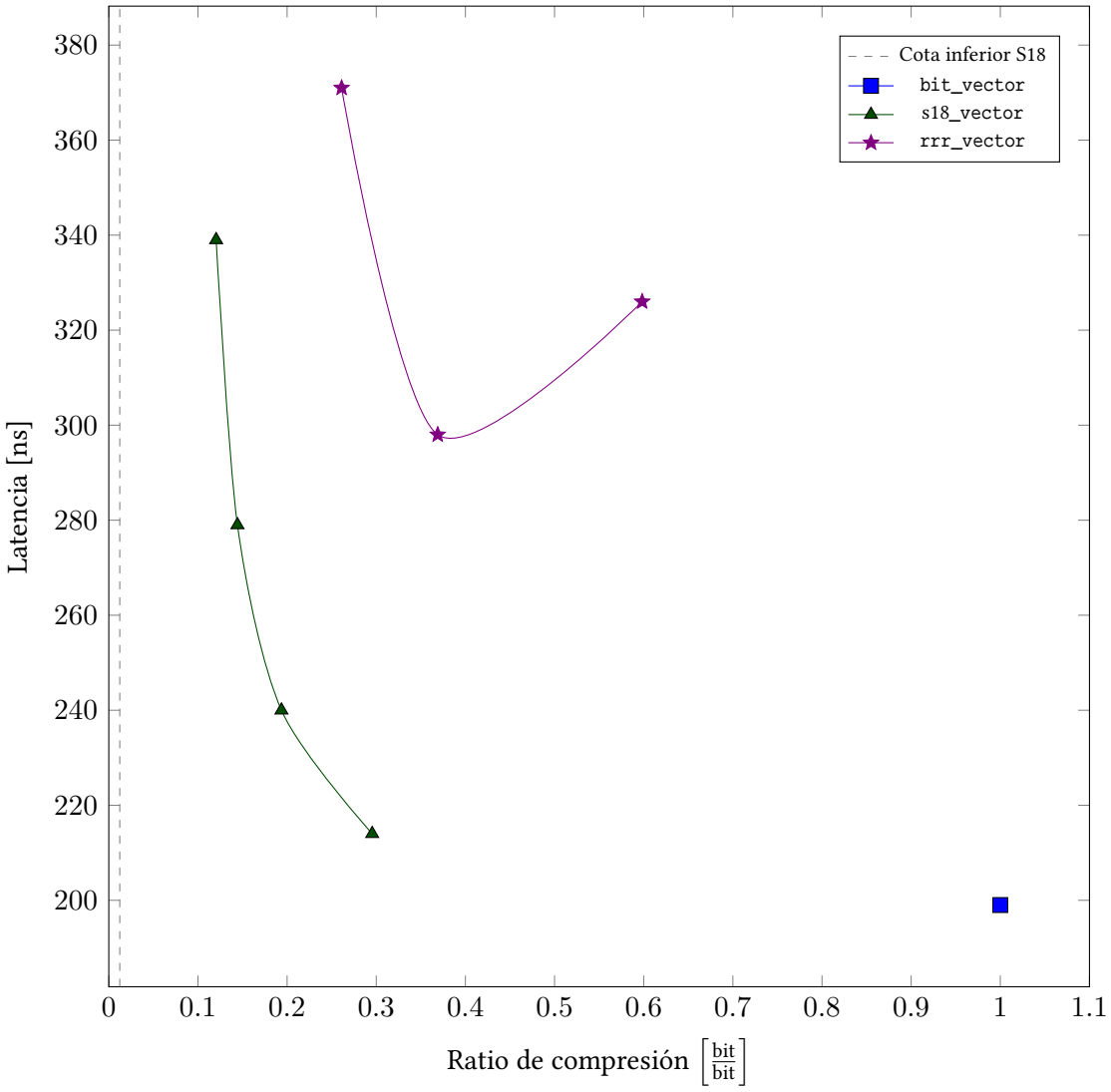


Figura B.58: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.8$. Los resutados completos se encuentran tabulados en la Tabla B.58.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	199	0.1252
s9_vector	8	295	2.3285
s9_vector	16	309	1.1701
s9_vector	32	316	0.5944
s9_vector	64	341	0.3945
s9_vector	128	371	0.2521
s9_vector	256	459	0.1821
s9_vector	512	589	0.1538
s18_vector	1	214	0.0370
s18_vector	2	240	0.0243
s18_vector	4	279	0.0181
s18_vector	8	339	0.0151
s18_vector	16	448	0.0136
s18_vector	32	688	0.0129
s18_vector	64	1129	0.0126
rrr_vector	7	326	0.0749
rrr_vector	15	298	0.0462
rrr_vector	31	371	0.0327
rrr_vector	63	369	0.0303
rrr_vector	127	435	0.0385
rrr_vector	255	899	0.0561
sd_vector	-	248	0.3790

Tabla B.58: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.8$.

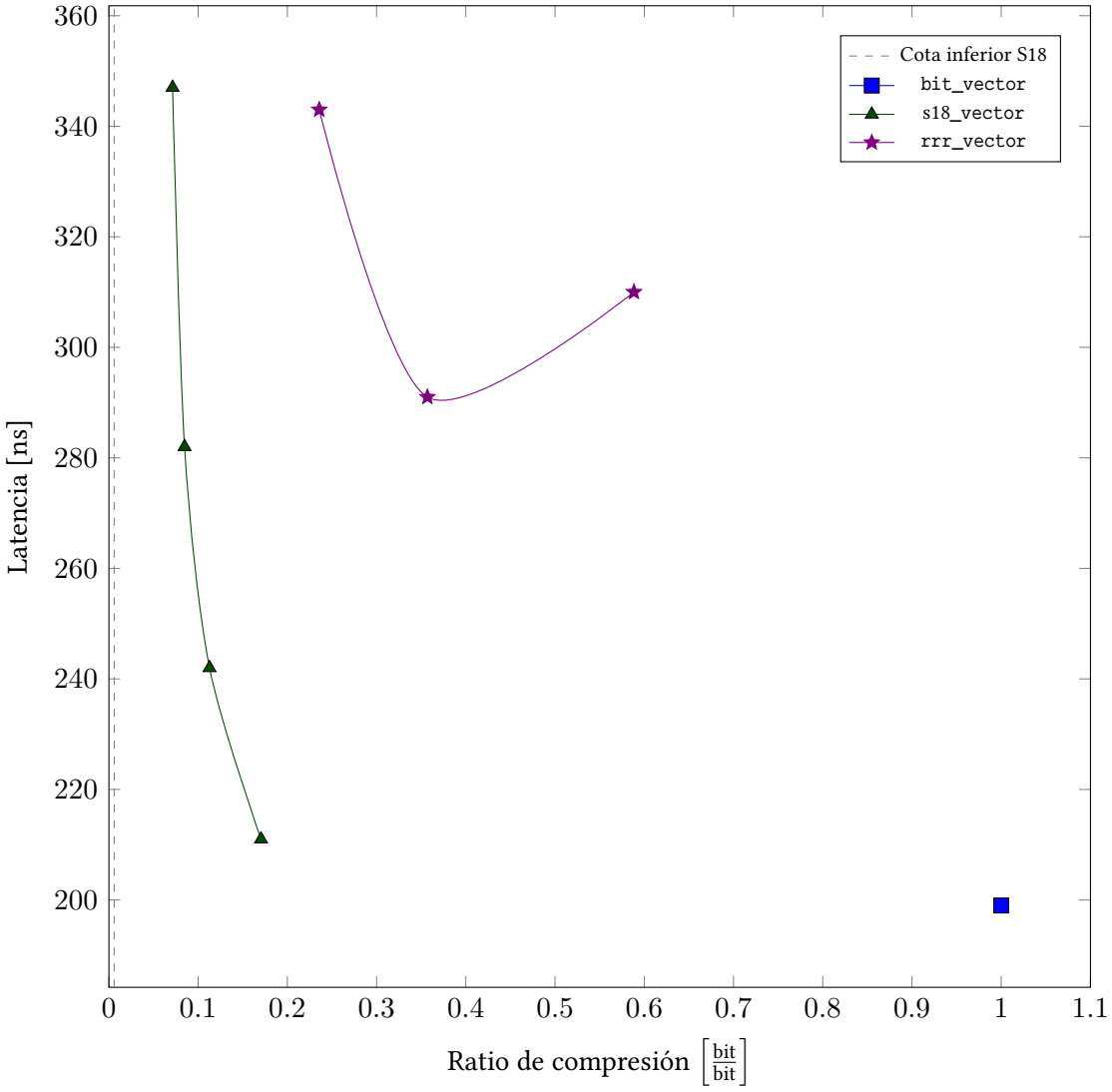


Figura B.59: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.9$. Los resutados completos se encuentran tabulados en la Tabla B.59.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	199	0.1227
s9_vector	8	318	2.5763
s9_vector	16	321	1.2914
s9_vector	32	331	0.6509
s9_vector	64	355	0.4328
s9_vector	128	381	0.2735
s9_vector	256	470	0.1948
s9_vector	512	598	0.1652
s18_vector	1	211	0.0209
s18_vector	2	242	0.0138
s18_vector	4	282	0.0104
s18_vector	8	347	0.0088
s18_vector	16	458	0.0079
s18_vector	32	689	0.0075
s18_vector	64	954	0.0074
rrr_vector	7	310	0.0722
rrr_vector	15	291	0.0438
rrr_vector	31	343	0.0289
rrr_vector	63	353	0.0231
rrr_vector	127	405	0.0256
rrr_vector	255	693	0.0344
sd_vector	-	249	0.3254

Tabla B.59: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.9$.

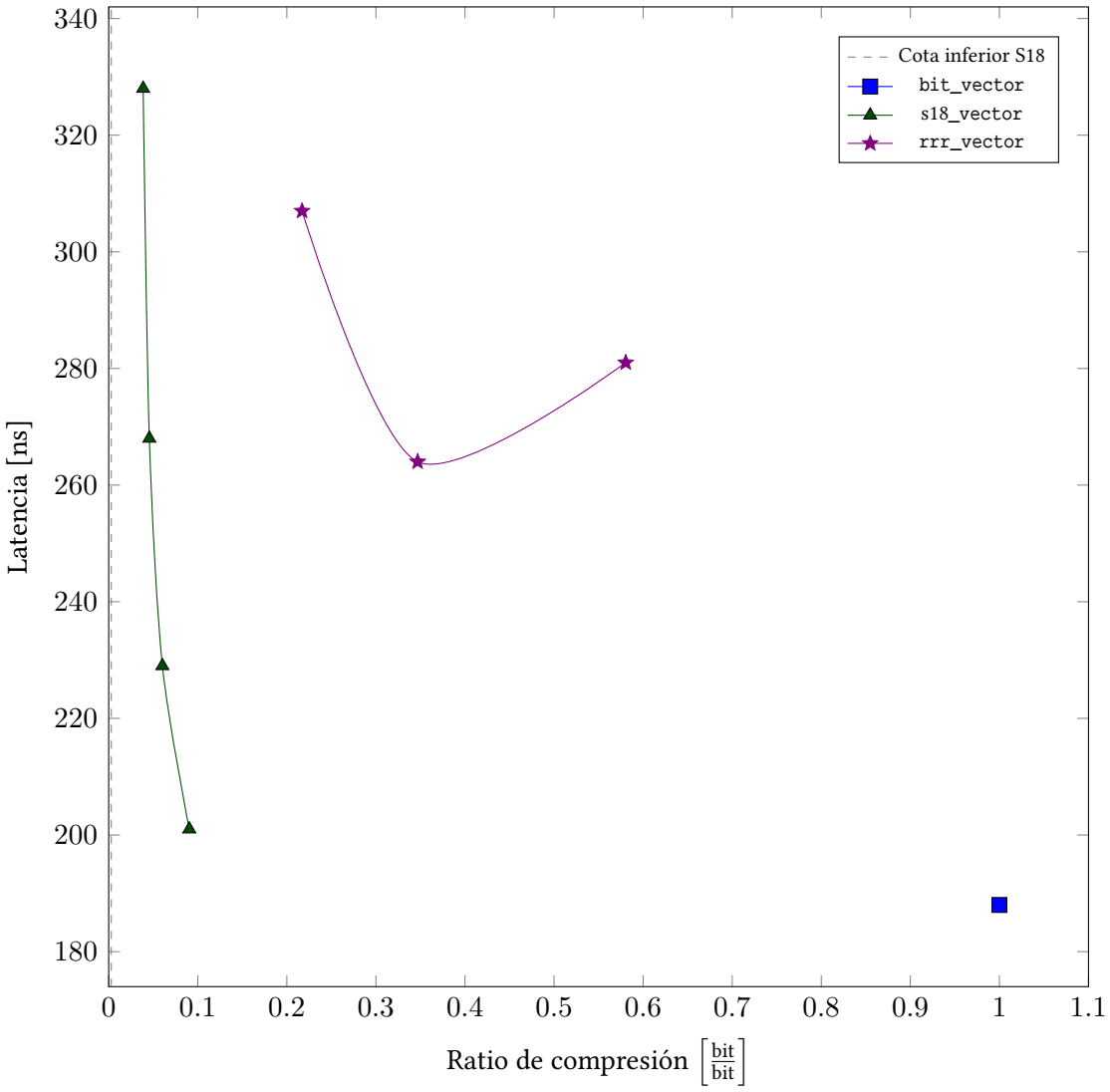


Figura B.60: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.95$. Los resutados completos se encuentran tabulados en la Tabla B.60.

Clase	Bloque	Latencia [ns]	Tamaño [Mb]
bit_vector	-	188	0.1208
s9_vector	8	330	2.7266
s9_vector	16	313	1.3651
s9_vector	32	324	0.6854
s9_vector	64	349	0.4562
s9_vector	128	374	0.2867
s9_vector	256	453	0.2023
s9_vector	512	577	0.1723
s18_vector	1	201	0.0109
s18_vector	2	229	0.0073
s18_vector	4	268	0.0055
s18_vector	8	328	0.0047
s18_vector	16	436	0.0042
s18_vector	32	593	0.0040
s18_vector	64	1131	0.0040
rrr_vector	7	281	0.0701
rrr_vector	15	264	0.0419
rrr_vector	31	307	0.0262
rrr_vector	63	310	0.0187
rrr_vector	127	362	0.0171
rrr_vector	255	515	0.0203
sd_vector	-	242	0.3394

Tabla B.60: Rendimiento de la operación Successor. El vector de bits contiene 10,000 elementos (*gaps* y *runs*). Los *gaps* son distribuidos uniformenmente entre 2 y 128 de largo. Los runs de ditribuyen Poisson con $\lambda = 100$. La probabilidad de que un elemento sea *run* es $P = 0.95$.