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

RING DINÁMICO MEJORADO

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

IGNACIO ALVEAL

PROFESOR GUÍA:
Gonzalo Navarro

MIEMBROS DE LA COMISIÓN:
Nancy Hitschfeld
Rodrigo Verschae

SANTIAGO DE CHILE
2025

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL EN
COMPUTACIÓN
POR: IGNACIO ALVEAL
FECHA: 2025
PROF. GUÍA: Gonzalo Navarro

RING DINÁMICO MEJORADO

Las bases de datos de grafos permiten almacenar, representar y consultar información mediante nodos y aristas para modelarla. En este contexto, es común realizar consultas utilizando *basic graph pattern* (BGP), equivalentes en términos generales a realizar *joins* naturales. Para resolver estas consultas de forma óptima, tanto en espacio como en tiempo, surge el *ring*, una base de datos compacta que logra tiempos bajos al implementar una variante del algoritmo *leapfrog triejoin*, el cual es *worst-case optimal*.

Sin embargo, todas estas características corresponden a una versión estática del *ring*, que no acepta actualizaciones durante su uso. Esto limita su aplicación en escenarios donde la información cambia frecuentemente, como bases de datos dinámicas o sistemas en tiempo real. Para superar esta limitación, se desarrolló el *ring* dinámico, que permite inserciones y eliminaciones de nodos y aristas, manteniendo la compacidad característica del *ring*. No obstante, esta versión dinámica presenta tiempos de consulta hasta un orden de magnitud mayores en comparación con la versión estática, lo que motivó la búsqueda de mejoras que preservaran la capacidad de actualización sin sacrificar el rendimiento.

Por ello, esta memoria presenta la implementación y evaluación de una versión mejorada del *ring* dinámico, una estructura compacta para el manejo eficiente de grafos dinámicos con grandes volúmenes de datos, como los provenientes de Wikidata. La mejora principal consiste en reemplazar el bitvector original por un adaptive dynamic bitvector y modificar el diccionario interno para utilizar árboles AVL balanceados. Se realizaron experimentos exhaustivos para determinar el mejor valor del parámetro θ y estudiar el impacto de la proporción entre consultas y actualizaciones en el desempeño de la estructura.

Los resultados muestran que la versión mejorada reduce los tiempos de consulta respecto al *ring* dinámico original, acercándose a los tiempos del *ring* estático en escenarios con alta proporción de consultas frente a actualizaciones. Sin embargo, las operaciones de actualización presentan un costo mayor, lo que hace que el rendimiento general dependa del contexto de uso, manteniendo un uso de memoria competitivo.

Finalmente, se concluye que el *ring* dinámico mejorado es adecuado para aplicaciones donde predominan las consultas, mientras que el *ring* dinámico original es preferible cuando hay un alto volumen de actualizaciones. Se sugieren futuras mejoras en la eficiencia del select y la incorporación de soporte para consultas SPARQL para ampliar su aplicabilidad.

Una frase de dedicatoria,
pueden ser dos líneas.

Saludos

Agradecimientos

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

Las bases de datos de grafos permiten almacenar, representar y consultar información mediante el uso de nodos y aristas para modelarla. Bajo este enfoque, es común realizar consultas utilizando *Basic Graph Patterns* (BGP), donde se define un conjunto finito de triples (sujeto-predicado-objeto) que especifican una estructura básica que se busca en el grafo.

Un aspecto importante a considerar es que, al evaluar un BGP (es decir, al realizar una consulta) sobre un grafo, cada triple pasa a representar una consulta atómica dentro del modelo. Por lo tanto, al evaluar un BGP con múltiples triples, el evaluador debe resolverlos mediante una operación equivalente a un *join* natural. En consecuencia, el *join* se convierte en una operación fundamental y frecuentemente utilizada en bases de datos de grafos [1]. Si a esto se suma el alto costo asociado a realizar *joins* sobre grandes volúmenes de información, se hace evidente la importancia de optimizar esta operación lo más posible en este contexto.

Un caso de mala optimización se da cuando un evaluador tradicional resuelve un BGP procesando pares de triples conectados por alguna variable en común, lo que equivale a realizar múltiples *joins* naturales entre dos tablas a la vez, hasta obtener el resultado final. Este enfoque puede llevar a procesar más información de la necesaria, lo que representa un problema común en la implementación de estos algoritmos. Por ello, se investiga constantemente en nuevos algoritmos que minimicen el procesamiento de información redundante. En particular, se busca diseñar algoritmos denominados *WCO* (*Worst-Case Optimal*). Un algoritmo se considera *WCO* cuando toma tiempo $O(T)$, y existe al menos un caso en el que no procesa información extra, es decir, la cardinalidad de la tabla resultante es proporcional a T. Cabe destacar que estos algoritmos no son óptimos para todas las consultas, pero sí lo son para al menos una, y en ese caso, incluso un algoritmo perfecto requeriría como mínimo $O(T)$ tiempo para resolverla.

Actualmente, los algoritmos *WCO* para *joins* son capaces de procesar consultas en un tiempo $\tilde{O}(\text{AGM})$ [2, 3], donde *AGM* representa el tamaño máximo que puede alcanzar el resultado del *join* para una consulta dada. Cabe destacar que no puede existir un algoritmo *WCO* con un orden de complejidad mejor que el límite *AGM*, ya que, en el peor de los casos, existen bases de datos cuya consulta genera un resultado de tamaño exactamente igual a *AGM*.

Un problema típico de los algoritmos *WCO* es su alta demanda de espacio. Para abordar esta limitación, Navarro et al. propusieron una nueva estructura llamada *Ring* [4], la cual implementa una variación del algoritmo *Leapfrog Triejoin*, siendo tanto el caso base como su variante algoritmos *WCO* [5]. La principal ventaja del *Ring* radica en su capacidad para utilizar una cantidad reducida de espacio adicional, limitada principalmente al almacenamiento del modelo. En experimentos comparativos con otras alternativas similares (utilizando grafos *RDF*), el *Ring* obtuvo resultados sobresalientes tanto en términos de espacio como de tiempo de ejecución.

Aunque el *Ring* demuestra ser altamente eficiente en términos de tiempo y espacio, los resultados anteriores se obtuvieron basándose en la versión estática del *Ring*. Esto implica que cualquier actualización, como inserciones o eliminaciones, requiere reconstruir la estructura desde cero, lo que limita considerablemente sus aplicaciones, dado que en muchos casos se necesita una estructura dinámica.

Para superar esta limitación, Yuval Linker propuso una versión dinámica del *Ring* [6], en la que se modificaron las estructuras subyacentes, particularmente el bitvector utilizado, con el fin de permitir inserciones y eliminaciones tanto de nodos como de aristas. Además, se añadió una estructura compacta y dinámica que permite traducir los IDs a cadenas de texto y viceversa, dado que era necesario transformar los IDs con los que trabaja el *Ring* en las cadenas que contienen los grafos.

Sin embargo, aunque la implementación del dinamismo fue exitosa y se logró mantener un uso de espacio cercano al del *Ring* original, la versión dinámica resultó ser un orden de magnitud más lenta que la versión estática al resolver consultas BGP. Esto se debió a que los bitvectors dinámicos resultaron ser más lentos de lo esperado en comparación con su contraparte estática.

Por estas razones, el presente trabajo propone utilizar un nuevo bitvector dinámico como estructura subyacente para implementar el *Ring* dinámico, en particular, el *adaptive dynamic bitvector* propuesto recientemente por Navarro [7]. Este bitvector logra un tiempo amortizado de $O(\log(n/q))$ para consultas y actualizaciones, en comparación con los bitvectors dinámicos habituales que, en la práctica, alcanzan tiempos de $O(\log(n))$. De esta manera, se busca conseguir un *Ring* dinámico que mantenga tanto el espacio utilizado como una velocidad de procesamiento más cercana a la versión estática. Cabe mencionar que, teóricamente, se ha probado que los bitvectors pueden alcanzar tiempos de $\Omega\left(\frac{\log(n)}{\log(\log(n))}\right)$, aunque aún no se ha logrado implementar esta mejora en la práctica.

Adicionalmente, se propone una mejora en la estructura encargada de realizar las traducciones entre el alfabeto y los IDs utilizados por el *Ring*, mediante modificaciones que permitan mantener el árbol utilizado en dicha estructura siempre balanceado.

Objetivos

Objetivo General

El objetivo general del trabajo es utilizar el *adaptive dynamic bitvector* en las estructuras subyacentes del *Ring*, con el fin de aumentar la eficiencia en los tiempos de procesamiento del *Ring* dinámico, acercándose más a los tiempos del *Ring* estático, mientras se conserva la competitividad en el espacio ocupado y las características de inserción y eliminación de tuplas del *Ring* dinámico.

Objetivos Específicos

1. Reemplazar los bitvectors del *Ring* dinámico por el *adaptive dynamic bitvector*.
2. Realizar tunning para el *adaptive dynamic bitvector* en el caso de consultas con actualizaciones sobre Wikidata.
3. Mejorar el diccionario encargado de transformar el alfabeto del grafo en los identificadores utilizados por el *Ring*, de modo que el árbol que lo contiene se mantenga siempre balanceado, mejorando así los tiempos de búsqueda en grafos extensos.
4. Realizar experimentos que evalúen las capacidades del *Ring* dinámico mejorado en escenarios realistas, a partir de consultas y actualizaciones sobre Wikidata (usando diversas proporciones), y comparar los resultados con otras soluciones existentes.
5. Diseñar experimentos para obtener datos precisos que permitan comparar, en tiempo y espacio, el desempeño de este *Ring* dinámico mejorado, el *Ring* dinámico desarrollado por Yuval Linker y el *Ring* estático.

Capítulo 2

Estado del arte

2.1. Bases de datos de grafos

Una base de datos de grafos es un modelo en el que los datos se representan mediante grafos, o bien, estructuras que generalizan la noción de grafo. En estas bases de datos, el elemento básico para realizar consultas es el *Basic Graph Pattern* (BGP). En particular, para consultas que involucran múltiples triples, se ejecuta una operación equivalente a un *join*.

En este contexto, actualmente existen distintos motores de bases de datos diseñados para soportar grafos, tales como Apache Jena, Blazegraph, Neo4j, Virtuoso, RDF-3X, entre otros. Cada uno presenta diversas características en cuanto a almacenamiento, indexado, *joins* y consultas, lo que los diferencia entre sí. Sin embargo, son pocas las alternativas que logran realizar *joins WCO* (worst case optimal) empleando un bajo consumo de almacenamiento.

2.2. Algoritmos *worst case optimal*

Los algoritmos *WCO* (*worst case optimal*) para *join* consiguen resolver consultas en un tiempo $\tilde{O}(Q^*)$, siendo Q^* la cota AGM [2, 3]. Es importante notar que, a pesar de compartir el mismo orden de complejidad, puede haber diferencias significativas en los tiempos de ejecución, pudiendo algunos algoritmos ser hasta un orden de magnitud más costosos que otros.

Algunas opciones que cumplen con ser *WCO* son Tentris [8], Qdag [9] y Jena LTJ (una modificación de Apache Jena que utiliza Leapfrog Triejoin). Cada uno de estos algoritmos presenta distintos desafíos: Tentris solo posee una implementación estática; Qdag, aunque es *WCO*, tiene un costo mayor en comparación con los algoritmos más eficientes; y Jena LTJ presenta la desventaja de un alto consumo de memoria debido a su implementación basada en LTJ.

Finalmente, se consideran las versiones anteriores del *Ring*: la primera es el *Ring* estático, que, como su nombre indica, no acepta inserciones ni eliminaciones; y la última, el *Ring*

dinámico implementado por Yuval Linker, que, si bien ofrece dinamismo, pierde un orden de magnitud en la eficiencia del algoritmo.

2.3. Leapfrog Triejoin

El algoritmo *Leapfrog Triejoin* (LTJ) aborda la consulta de *join* realizando una iteración a través de los atributos, en lugar de las relaciones, que es lo habitual. Esto reduce la cantidad de combinaciones a comparar y aumenta la eficiencia del algoritmo. LTJ es un algoritmo *WCO* con un factor logarítmico adicional, es decir, su costo es $O(Q^* \log(n))$, donde Q^* es el límite AGM para la consulta y n es la cardinalidad más grande entre las relaciones involucradas en el *join* [5].

Para implementar este enfoque, es necesario almacenar $d!$ tries por cada tabla, siendo d la cantidad de atributos de la tabla. Cada uno de estos tries representa una combinación diferente de los atributos. Por ejemplo, si una tabla M tuviera los atributos X e Y, y 3 elementos, se almacenarían las 2 representaciones del trie que se muestran en la figura 2.1.

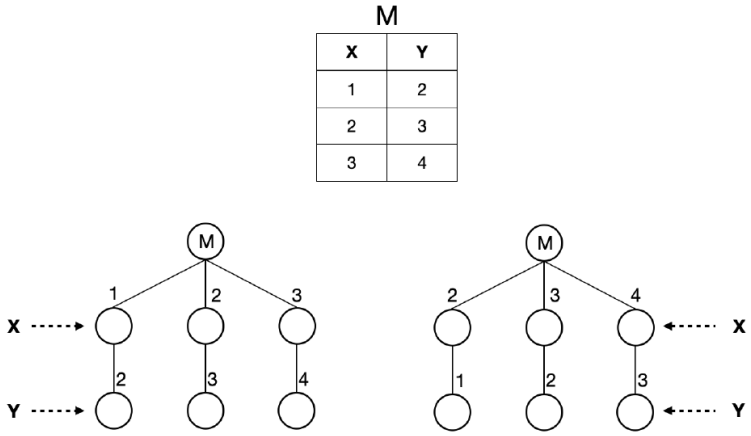

Figura 2.1: Ejemplo de almacenamiento de una tabla para usar LTJ

Una vez almacenadas las representaciones para las tablas, se itera a través de los atributos utilizando para el *join* aquellos tries que tengan los atributos en común más cercanos a la raíz. Se instancia una variable a la vez de los atributos, desplazándose en todas las relaciones que contienen dicha variable. Por ejemplo, si a la tabla M se le quisiera realizar un *join* natural con una tabla N que tiene los atributos X y Z, y 2 elementos, se seleccionarían los 2 tries que se muestran en la figura 2.2.

Como se puede observar, el atributo común es X, por lo que se buscará el primer valor que comparten ambos en X, que corresponde a 1. Luego, se descenderá en ambos tries; dado que en este caso solo se comparte un atributo, se asignarán a las variables Y y Z los valores correspondientes, que en este caso son 2 y 7, obteniendo la tupla $(1, 2, 7)$. Después, se vuelve a subir al atributo X y se repite el proceso sin considerar el valor 1 (de izquierda a derecha). En este caso, el siguiente valor común es 3, agregando la tupla $(3, 4, 9)$ y obteniendo así los dos elementos resultantes del *join*.

En el caso específico de evaluar un BGP sobre un grafo, las relaciones poseen tres atributos

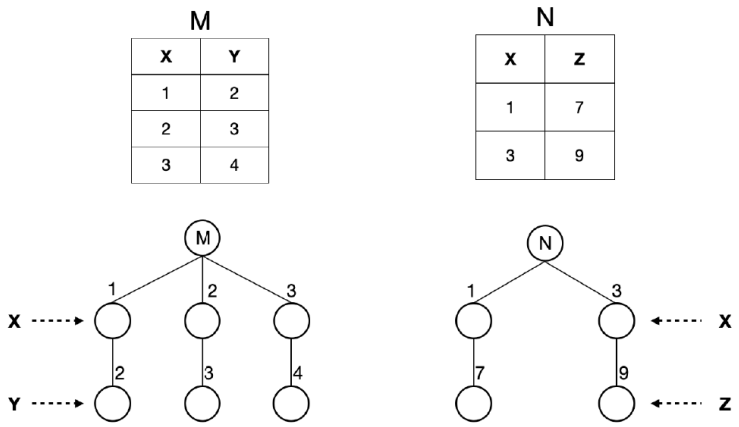

Figura 2.2: Ejemplo de relaciones para LTJ, extraido de [5]

(sujeto, predicado y objeto) y hasta n filas, por lo que se necesitarían 6 tries, almacenando cada uno hasta n cadenas.

Cabe destacar que, aunque *Leapfrog Triejoin* es un algoritmo eficiente y ha sido implementado exitosamente para el lenguaje SPARQL [10], para que funcione de manera óptima requiere almacenar una cantidad factorial de tries, lo que conlleva un uso significativo de espacio.

2.4. Ring

Para solucionar los problemas de espacio del *LTJ*, se creó una variante llamada *Ring*, que mantiene la eficiencia del *LTJ* pero utiliza una cantidad muy reducida de espacio adicional para resolver el problema. Para ello, se aplicaron diversas técnicas de compactación e indexación de texto, que permiten construir una estructura equivalente al *LTJ* en su funcionamiento.

El primer paso para construir el *Ring* consiste en crear un diccionario que mapea todos los símbolos a un conjunto de enteros positivos consecutivos (suponiendo que existan U símbolos, se mapearán al conjunto $[1..U]$). Bajo este contexto, el *Ring* almacenaría un conjunto de n tuplas pertenecientes a $[1..U]^d$, siendo d la cantidad de atributos. Dado que en este caso nos enfocamos en grafos, hablaremos del *Ring* especializado para almacenar grafos, el cual tiene una aridad de $d = 3$, almacenando las relaciones como triples de la forma S (sujeto), P (predicado) y O (objeto).

A partir de este punto, surge la principal diferencia con el *LTJ*, ya que en lugar de almacenar 6 tries (debido a que la aridad es 3), se almacena una tabla T con una cantidad de atributos equivalente a la aridad, es decir, 3 atributos para este caso. En esta tabla T, cada fila representa un ordenamiento distinto de las tuplas. En particular, para el caso de los grafos, se tienen 3 atributos (S, P y O), donde cada columna representa:

- El atributo C_O, que representa una tupla ordenada en el orden SPO; por lo tanto, las tuplas se ordenan de menor a mayor por S, con un primer desempate por P y un último desempate por O.

- El atributo C_P, que representa una tupla ordenada en el orden OSP; por lo tanto, las tuplas se ordenan de menor a mayor por O, con un primer desempate por S y un último desempate por P.
- El atributo C_S, que representa una tupla ordenada en el orden POS; por lo tanto, las tuplas se ordenan de menor a mayor por P, con un primer desempate por O y un último desempate por S.

A continuación, se puede observar un ejemplo de esta tabla con sus respectivos atributos (véase la figura 2.3):

Figura 2.3: Ejemplo visual del *Ring*, extraído de [6]

Notar que C_O, C_P y C_S corresponden únicamente al último elemento de la tupla, el cual se guarda en formato de *Wavelet Matrix* (explicado en la sección 2.5.6) para ocupar la menor cantidad de espacio posible. Sin embargo, esto impide identificar la misma tupla entre distintas columnas, por lo que es necesaria la introducción de una estructura adicional em el *Ring* que permita moverse entre columnas.

Para cada columna C_j se crea un arreglo A_j, definido como:

$$A_j[c] = |\{i \in [1..n] : C_j[i] < c\}| \tag{2.1}$$

De este modo, el arreglo A_j almacena en su c-ésimo elemento la cantidad de ocurrencias de símbolos menores que c, utilizando para ello *gap-encoded bitvectors*, tal como se describe en la sección 2.5.4.

Con esta nueva estructura añadida al *Ring*, es posible movilizarse entre las distintas columnas del *Ring*. En particular, se usa una función biyectiva $F_j : [1..n] \to [1..n]$, que se define como:

$$F_j(i) = A_j[c_i] + rank_{c_i}(C_j, i) \tag{2.2}$$

Donde c_i es el i-ésimo elemento del vector C_j. Sea T' un reordenamiento de la representación de las tuplas de T según la columna j, e $i' := F_j(i)$. Entonces, las filas $T[i]$ y $T'[i']$

corresponden a la misma tupla [4]. En particular, esto permite pasar de C_O a C_P usando F_O, de C_P a C_S usando F_P, y de C_S a C_O usando F_S.

Un ejemplo del uso de esta función sería el siguiente: en la figura 2.3, si quisiéramos saber cuál es el predicado de una tupla, sabiendo que esta es la primera tupla que aparece representada en C_O, entonces, dado que $C_O[1] = 2$, su c_i es 2. Luego, consultamos el vector A_O asociado a C_O, en particular $A_O[2]$, que da 1. A continuación, calculamos $rank_2(C_j, 1)$, que para este caso es 1 (equivalente a contar la cantidad de veces que aparece el símbolo 2 en $C_j[1..1]$). Sumando estos resultados obtenemos 2. Sabiendo que $C_P[2] = 1$, concluimos que el predicado de la tupla corresponde al símbolo 1 en el diccionario.

Dado que la función F es biyectiva, posee una inversa, definida como:

$$F_j^{-1}(i') = select_{c_i}(C_j, i' - A_j[c_i]) \tag{2.3}$$

Donde c_i se obtiene de la desigualdad $A_j[c_i] < i' \leq A_j[c_i+1]$. Esta función permite invertir el proceso realizado por F, posibilitando moverse de una columna a otra en la tabla. Así, dada una tupla, es posible obtener las 6 ordenaciones posibles, haciendo que el *Ring* sea una representación equivalente a la información almacenada en los 6 tries del *LTJ*.

Un ejemplo de uso de la función inversa sería el caso contrario al anterior: si quisiéramos conocer el objeto de una tupla sabiendo que esta es la segunda en C_P (es decir, $i' = 2$), entonces c_i debe cumplir con la desigualdad $A_O[c_i] < 2 \leq A_O[c_i + 1]$. Sabiendo que $A_O[2] = 1$ y $A_O[3] = 2$, concluimos que $c_i = 2$. Luego calculamos:

$$select_2(C_O, 2 - A_O[2]) = select_2(C_O, 1) \tag{2.4}$$

Lo que equivale a encontrar la primera ocurrencia del número 2 en C_O, dando como resultado 1. Como $C_O[1] = 2$, sabemos que el objeto de la tupla corresponde al símbolo 2 en el diccionario.

Cabe notar que esta transformación de índices puede extenderse a intervalos (cuando el intervalo tiene el mismo número), transformando un intervalo contiguo $[s, e]$ en un intervalo $[s', e']$ en otra columna. Esto permite al *Ring* simular el descenso por los nodos de los tries utilizados en el *LTJ*.

Finalmente, cuando en *LTJ* se evalúa una variable x, esta debe poder intersectar los hijos de los nodos de x. Esto equivale, en el *Ring*, a encontrar los valores comunes en todos los rangos de las columnas C_j correspondientes, operación que se realiza usando la función *range-next-value*. Con esto, se logra el mismo funcionamiento que al utilizar *LTJ*, pero empleando las estructuras y funciones definidas en el *Ring*.

2.5. Estructuras compactas

Las estructuras compactas son una parte fundamental en los algoritmos mencionados anteriormente, ya que estos no solo deben ser competitivos en términos de tiempos de ejecución, sino también en cuanto al uso eficiente del almacenamiento. Por ello, es esencial emplear estructuras capaces de compactar la información sin sacrificar eficiencia en las operaciones.

Estas estructuras compactas constituyen la base sobre la cual se construye la arquitectura del *Ring*, y, por lo tanto, tienen un impacto directo en sus propiedades. Influyen tanto en el rendimiento como en la capacidad de dinamismo del *Ring*, determinando en gran medida su eficiencia general.

2.5.1. Bitvectors

Los *bitvectors* se definen como una secuencia de bits $B[1..n]$ [11], que soporta las siguientes operaciones fundamentales:

- $access(B, i)$: retorna el bit $B[i]$, $\forall i \in \{1, \ldots, n\}$.
- $rank_v(B, i)$: retorna el número de ocurrencias del bit $v \in \{0, 1\}$ en $B[1..i]$, $\forall i \in \{1, \ldots, n\}$.
- $select_v(B, i)$: retorna la posición en B de la i-ésima ocurrencia del bit $v \in \{0, 1\}$, $\forall i \geq 0$.

La principal característica de los *bitvectors* es su alta compacidad, al utilizar secuencias de bits para representar la información. En este contexto, existe una distinción clave entre los *bitvectors* estáticos y dinámicos. En particular, los *bitvectors* estáticos pueden realizar las operaciones descritas anteriormente en tiempo constante $O(1)$, mientras que los *bitvectors* dinámicos —que además permiten inserciones y eliminaciones— presentan un costo teórico por operación de $\Omega(\log(n)/\log\log(n))$, aunque en la práctica solo se alcanzan tiempos de $O(\log n)$.

Esta diferencia en los tiempos de operación provoca que todas las estructuras compactas basadas en *bitvectors* sufran una degradación significativa en su rendimiento al ser adaptadas a versiones dinámicas. Este fenómeno también afecta al *Ring*, cuya versión dinámica pierde eficiencia en un orden de magnitud respecto a su contraparte estática.

No obstante, si las actualizaciones son menos frecuentes que las consultas —por ejemplo, si se realizan q veces más consultas que actualizaciones—, es posible lograr un tiempo amortizado por operación de $O(\log(n/q))$ [7], lo que puede permitir una mejora sustancial en escenarios de uso donde predominan las consultas.

2.5.2. Bitvectors estáticos

Una implementación clásica de *bitvectors* estáticos fue propuesta por Guy Jacobson, quien demostró cómo obtener tiempos constantes $O(1)$ para la operación *rank* utilizando $O(n)$

espacio, y tiempos muy eficientes para la operación *select* [12]. Esta implementación se basa en dos estructuras auxiliares llamadas *superbloques* y *bloques*, que almacenan información acumulada de unos en el bitvector en distintos niveles de granularidad.

Para construir estas estructuras, se comienza dividiendo el bitvector $B[1..n]$ en *superbloques* de tamaño $\log_2^2 n$ bits. Así, se tienen $\frac{n}{\log_2^2 n}$ superbloques. Para cada uno, además de almacenar sus $\log_2^2 n$ bits, se guarda el número total de unos acumulados hasta el inicio del superbloque, lo que requiere $\log_2 n$ bits (suficientes para representar cualquier valor entre 0 y n). Por lo tanto, cada superbloque ocupa $\log_2^2 n + \log_2 n$ bits, y el total para todos los superbloques es:

$$\frac{n}{\log_2^2 n} \cdot (\log_2^2 n + \log_2 n) = n + \frac{n}{\log_2 n} = O(n) \tag{2.5}$$

Cada superbloque se divide a su vez en *bloques* de tamaño $\frac{\log_2 n}{2}$ bits, lo que da $2 \cdot \log_2 n$ bloques por superbloque. Cada bloque, además de almacenar sus $\frac{\log_2 n}{2}$ bits, guarda la cantidad de unos acumulados desde el inicio del superbloque hasta justo antes de ese bloque. Para representar este valor, se necesitan $\log_2 \log_2^2 n = 2 \log_2 \log_2 n$ bits. Así, cada bloque requiere $\frac{\log_2 n}{2} + 2 \cdot \log_2 \log_2 n$ bits, y por ende, cada superbloque requiere:

$$2 \cdot \log_2 n \cdot (\frac{\log_2 n}{2} + 2 \log_2 \log_2 n) + log_2 n = \log_2^2 n + 4 \cdot \log_2 n \cdot \log_2 \log_2 n + log_2 n \tag{2.6}$$

Finalmente, sumando todos los superbloques, el total de espacio ocupado por el bitvector con las estructuras auxiliares es:

$$n + \frac{n}{\log_2 n} + \frac{4n \cdot \log_2 \log_2 n}{\log_2 n} = O(n) \tag{2.7}$$

Lo cual mantiene la estructura dentro de espacio lineal. Estas estructuras permiten realizar *rank* en tiempo constante y *select* en tiempo logarítmico o casi constante, dependiendo de la implementación exacta [13, 14, 15]. Un ejemplo de esta estructura se puede encontrar en la figura 2.4.

Una vez construidas estas estructuras auxiliares, es posible calcular la operación $rank_1(i)$ en tiempo $O(1)$. Para ello, se siguen los siguientes pasos:

Primero, se determina en qué *superbloque* se encuentra el bit i-ésimo. Esto se calcula como $j = \lfloor i / \log_2^2 n \rfloor$, donde $\log_2^2 n$ es el tamaño de cada superbloque. Dado que la cantidad de unos acumulados hasta el inicio del superbloque está precomputada, esta operación toma tiempo constante.

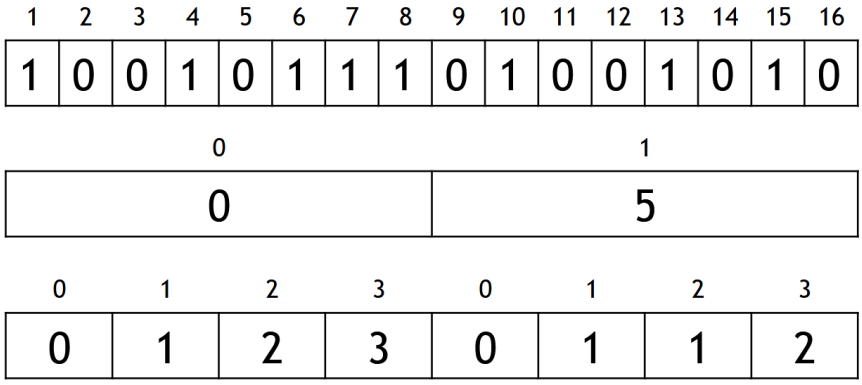

Figura 2.4: Ejemplo de la estructura auxiliar de superbloque y bloque para un bitvector estatico de 16 bits, extraido de [15]

Luego, se determina en qué *bloque* dentro del superbloque se encuentra dicho bit. Esto se obtiene con $k = \lfloor 2 \cdot i / \log_2 n \rfloor$, ya que el tamaño de los bloques es $(\log_2 n)/2$. La cantidad de unos acumulados desde el inicio del superbloque hasta el bloque k también está almacenada, por lo que esta consulta también se realiza en tiempo $O(1)$.

Finalmente, se debe contar cuántos unos hay desde el inicio del bloque hasta la posición i. Para esto, se presentan dos casos:

- Si el bloque es lo suficientemente pequeño para caber en una palabra de máquina, se puede usar una instrucción a nivel de hardware (como *popcount*) para contar los unos en tiempo constante.
- Si el bloque es más grande, se puede usar una tabla de respuestas precomputadas. Como hay $2^{\log_2 n/2} = \sqrt{n}$ combinaciones posibles de bits en un bloque de tamaño $\log_2 n/2$, y para cada combinación se almacenan $\log_2 n/2$ respuestas (una por cada posición), el total de respuestas es $\sqrt{n} \cdot (\log_2 n/2)$. Si cada respuesta ocupa $\log_2(\log_2 n/2)$ bits, el total es $\sqrt{n} \cdot \frac{\log_2 n}{2} \cdot \log_2 \left(\frac{\log_2 n}{2}\right)$, que sigue siendo $O(n)$ en espacio, y permite responder consultas en $O(1)$.

En consecuencia, se puede calcular $rank_1(i)$ en tiempo $O(1)$. Para $rank_0(i)$, el proceso es análogo, pero contando ceros. Alternativamente, se puede utilizar la identidad $rank_0(i) = i - rank_1(i)$, ya que todo bit es un 0 o un 1, por lo que su suma siempre da i. Como $rank_1(i)$ se obtiene en tiempo constante, $rank_0(i)$ también.

En cuanto a la operación $select_1(i)$, se puede implementar usando las mismas estructuras auxiliares. Una implementación común logra tiempo $O(\log_2 n)$ en el peor caso, aunque en la práctica resulta bastante eficiente. Cabe destacar que existen alternativas que alcanzan tiempos $O(1)$ usando espacio $O(n)$, como la implementación descrita por Clark [16][17, 18].

Para realizar la operación $select_1(i)$ utilizando esta estructura, el proceso puede dividirse en tres pasos principales. El primero consiste en localizar el superbloque que contiene el i-ésimo uno del bitvector. Para ello, se realiza una aproximación inicial basada en la suposición de que los unos están distribuidos de forma homogénea en el bitvector. Sea #ones el número

total de unos, entonces en promedio hay $n/\#\text{ones}$ bits por cada uno. Bajo esta suposición, el i-ésimo uno se encontraría aproximadamente en el rango:

$$\left[\left\lfloor i \cdot \frac{n}{\#\text{ones}} \right\rfloor, \left\lceil i \cdot \frac{n}{\#\text{ones}} \right\rceil\right] \tag{2.8}$$

Aunque en la práctica la distribución de los unos no necesariamente es homogénea, esta aproximación resulta útil y tiende a ser más precisa en bitvectors con distribuciones más uniformes.

Sea j el j-ésimo *superbloque* a aproximar. Se puede calcular una aproximación inicial de su posición mediante la expresión:

$$j = i \cdot \frac{n}{\#\text{ones}} \cdot \frac{1}{\log_2^2(n)} \tag{2.9}$$

Esta fórmula permite estimar la ubicación del i-ésimo uno dentro del bitvector y determinar el superbloque que lo contiene. Una vez realizada esta aproximación, si el valor calculado está antes o después del verdadero i-ésimo uno, se ejecuta una búsqueda exponencial en la dirección correspondiente, seguida de una búsqueda binaria sobre el rango identificado.

Notar que la aproximación inicial toma tiempo $O(1)$, y tanto la búsqueda exponencial como la binaria requieren $O(\log_2(n))$, resultando en un tiempo total de búsqueda de $O(\log_2(n))$.

Una vez localizado el *superbloque* en donde se encuentra el i-ésimo uno del *bitvector*, se procede al segundo paso, que consiste en buscar el *bloque* en donde se encuentra dicho uno. Para ello, se realiza un procedimiento equivalente al anterior, pero restringido al interior del superbloque.

Sea k el k-ésimo bloque a aproximar, j el j-ésimo superbloque donde se encuentra el i-ésimo bit, y $S(x)$ la cantidad de unos acumulados hasta antes del superbloque x. Entonces, se puede calcular la aproximación inicial al bloque como:

$$k = (i - S(j)) \cdot \frac{\log_2^2(n)}{S(j+1) - S(j)} \cdot \frac{2}{\log_2(n)} \tag{2.10}$$

Una vez calculada esta aproximación inicial, se realiza una búsqueda del bloque que contiene al i-ésimo uno, de manera análoga a la realizada en los superbloques: se ejecuta una búsqueda exponencial seguida de una búsqueda binaria.

Finalmente, una vez localizado el bloque en donde se encuentra el i-ésimo uno del *bitvector*, se procede con el último paso, que consiste en encontrar dicho uno dentro del bloque. Para ello, se recorre el bloque palabra por palabra, contando la cantidad de unos presentes en cada una, hasta encontrar la palabra que contiene al i-ésimo uno del *bitvector*; es decir, aquella donde la cantidad acumulada de unos alcanza o supera a i.

Una vez identificada dicha palabra, se recorre de manera secuencial, contando los unos

hasta llegar al i-ésimo uno. Notar que, por construcción de la estructura, el tamaño de cada bloque es $\log_2(n)$ bits, por lo que la búsqueda dentro del bloque toma un tiempo $O(\log_2(n))$.

Y ya que cada paso toma un tiempo $O(\log_2(n))$, el proceso completo para calcular $select_1(i)$ toma un tiempo $O(\log_2(n))$. Notar, a su vez, que como se utiliza la misma estructura auxiliar empleada para el cálculo de $rank_1(i)$, esta implementación requiere un espacio $O(n)$.

Esta es una implementación muy similar a la utilizada en la librería SDSL, que se emplea en la versión estática del *ring*, así como en la sección estática del adaptive dynamic bitvector.

2.5.3. Bitvectors Dinamicos

Para el caso del *ring* dinámico, el bitvector dinámico utilizado es una variación desarrollada por Yuval Linker, basada en la implementación propuesta por Nicola Prezza en la librería DYNAMIC para bitvectors dinámicos, la cual está construida sobre la utilización de árboles B (B-Trees) [19, 6].

En particular, se emplea una estructura diseñada para resolver el problema de *sumas parciales buscables con inserción* (SPSI, por sus siglas en inglés), en el cual se requiere mantener una estructura P que almacene una secuencia de enteros no negativos $s_1, s_2, \ldots, s_m$, y soporte eficientemente las siguientes operaciones:

- $P.sum(i) = \sum_{j=1}^{i} s_j$.
- $P.search(x)$: retorna el menor i tal que la suma parcial $\sum_{j=1}^{i} s_j \geq x$.

Además, dado que se desea tener dinamismo, se requiere que la estructura admita tanto inserciones como actualizaciones, es decir, poder cambiar el valor de algún s_i o agregar un nuevo entero a la secuencia.

Bajo estas condiciones, si en lugar de permitir cualquier entero no negativo se restringe la secuencia a contener únicamente valores 0 y 1, entonces la operación $P.sum(i)$ pasa a ser equivalente a $rank_1(i)$, y la operación $P.search(x)$ equivale a $select_1(x)$.

Por lo tanto, una estructura que resuelve el problema SPSI es también una estructura que resuelve el problema de mantener un bitvector dinámico, es decir, un bitvector que soporta actualizaciones e inserciones.

También es importante notar que dentro del problema SPSI existen ciertas garantías teóricas sobre el tamaño que utiliza la estructura. En particular, sea m la cantidad de elementos en la secuencia, s_i el i-ésimo elemento, y $M = m + \sum_{i=1}^{m} s_i$, entonces la estructura de sumas parciales tiene como cota superior de espacio:

$$2m \cdot \log\left(\frac{M}{m}\right) + \log(\log(m)) + O\left(\frac{\log(M)}{\log(m)}\right). \tag{2.11}$$

Sin embargo, considerando que se utiliza para sumar bits que solo pueden ser 0 o 1, el valor máximo de M es $2 \cdot m$. Reemplazando en la expresión anterior, la cota se convierte en:

$$2m \cdot \log(2) + \log(\log(m)) + O\left(\frac{\log(2m)}{\log(m)}\right), \tag{2.12}$$

Lo cual implica que el espacio utilizado es $O(m)$ [19]. Notar que, como se mencionó anteriormente, los bitvectors dinámicos como el propuesto por Nicola Prezza solo consiguen tiempos $O(\log(n))$ para las operaciones de $select_1(i)$ y $rank_1(i)$ en la práctica, aunque tienen una cota inferior teórica de $\Omega\left(\frac{\log(n)}{\log(\log(n))}\right)$ [20].

2.5.4. Gap-Encoded Bitvectors

Una de las estructuras compactas que se pueden construir usando bitvectors es la estructura llamada *gap-encoded bitvectors*. Dada una secuencia creciente de m enteros pertenecientes al intervalo $[1, n]$, se puede representar cada elemento $x_1, x_2, \ldots, x_m$ de la secuencia almacenando la secuencia como el siguiente bitvector:

$$0^{x_1}10^{x_2-x_1}1 \ldots 10^{x_{m-1}-x_{m-2}}10^{x_m-x_{m-1}} \tag{2.13}$$

Donde 0^k corresponde a tener k ceros consecutivos [19]. Es decir, dado el elemento i de la secuencia, este se representará entre el $(i-1)$-ésimo uno e i-ésimo uno del bitvector, representado a través de $x_i - x_{i-1}$ ceros, notando que el primer y último elemento son la excepción, pues no existe el 0-ésimo uno ni el m-ésimo uno.

De esta forma se puede obtener el i-ésimo elemento de la secuencia en el bitvector al realizar un $select_1(i)$, en particular, $x_i = select_1(i) - i$, con lo cual el tiempo que se demora en encontrar el i-ésimo elemento del gap-encoded bitvector es equivalente al tiempo que tarda el bitvector usado en realizar un $select_1(i)$.

Una desventaja de esta estructura es el espacio requerido cuando la cantidad de elementos en la secuencia es muy baja en comparación con el tamaño de los números de la secuencia. En particular, el tamaño de la estructura es $\sum_{i=1}^{m} s_i + m$, o equivalentemente, $O(n+m)$, por lo cual, cuando $n \gg m$ esta estructura usa más espacio del necesario, y cuando $n \approx m$ se puede decir que la estructura es de orden $O(n)$.

2.5.5. Wavelet Trees

Otra estructura compacta que se puede implementar usando bitvectors, y que es utilizada en la implementación del *ring*, es la estructura llamada *Wavelet Tree*. Dada una secuencia de símbolos provenientes de un alfabeto $[1, \tau]$, esta estructura representa la secuencia mediante un árbol binario, donde cada nodo del árbol corresponde a un rango específico de símbolos del alfabeto, y cada hoja contiene un símbolo único del alfabeto.

Por lo tanto, si un nodo representa los símbolos $[a, b]$ del alfabeto, entonces su hijo izquierdo representa los símbolos $[a, \lfloor (a+b)/2 \rfloor]$, mientras que su hijo derecho representa los símbolos $[\lfloor (a+b)/2 \rfloor + 1, b]$.

Almacenando en el nodo, un bitvector que indica para cada elemento del rango $[a, b]$, a cuál de los dos hijos pertenece. Es decir, si el i-ésimo bit del bitvector es 0, el símbolo correspondiente está en el rango del hijo izquierdo; si es 1, pertenece al hijo derecho. Un ejemplo visual de un wavelet tree se puede observar en la figura 2.5.

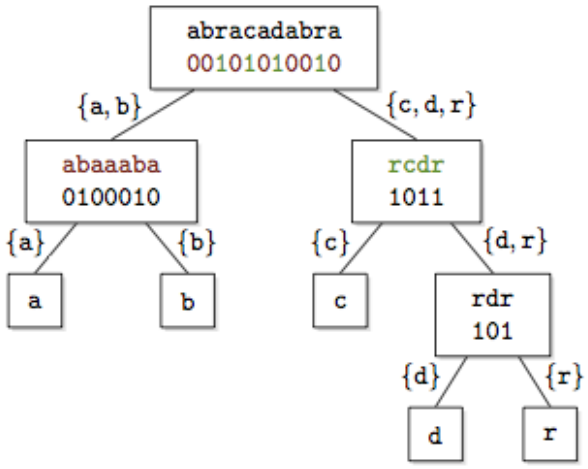

Figura 2.5: Ejemplo de un wavelet tree de la palabra abracadabra, extraido de Wikipedia.

Notar que el espacio utilizado por esta estructura es $O\big(m \cdot \lceil \log_2(\tau) \rceil\big)$ [21], ya que en cada nivel del árbol, la suma de los bits almacenados en los bitvectors de todos los nodos de ese nivel es a lo más m. Esto se debe a que en cada nivel, cada elemento de la secuencia $x_1, x_2, \ldots, x_m$ contribuye a lo más con un solo bit, que indica si se dirige al hijo izquierdo o derecho del nodo correspondiente. Luego, la altura del árbol es $\lceil \log_2(\tau) \rceil$ porque en cada nivel el rango de símbolos se divide a la mitad, repitiendo el proceso hasta alcanzar nodos que representan un único símbolo.

2.5.6. Wavelet Matrix

Para minimizar aún más el espacio ocupado por los wavelet trees, es importante notar que toda la información relevante está contenida en los bitvectors de cada nodo. Por lo tanto, si en la implementación se logra omitir tanto los punteros como los nodos, se podría mantener la misma funcionalidad, pero utilizando menos bits.

Al implementar la idea anterior surgen los *Wavelet Matrix* [22] [23], que en lugar de utilizar un árbol binario para almacenar los bitvectors, emplean una matriz acompañada de un vector. En esta estructura, cada fila de la matriz representa un nivel del árbol binario original. En particular, sea B una wavelet matrix, se denota la l-ésima fila de B como B_l, observando que cada fila corresponde a un bitvector y, por lo tanto, dentro de cada fila se pueden realizar las mismas operaciones que en un bitvector convencional.

Para la construcción de esta estructura, el primer paso consiste en convertir los símbolos de la secuencia a una representación binaria. Es importante notar que la cantidad de bits utilizados en estas representaciones binaras determinará la altura de la matriz, y que todos los símbolos deben ser representados con la misma cantidad de bits. Para facilitar la explicación,

asumiremos que se cuenta con una conversión perfecta; es decir, dado un alfabeto con τ símbolos, la longitud de la representación binaria de cada símbolo es $\lceil \log_2(\tau) \rceil$.

Entonces, para construir la primera fila de la matriz, se extrae el primer bit de cada elemento de la secuencia y se colocan en el mismo orden en que aparecen en la secuencia. Es decir, dado el i-ésimo elemento de la secuencia, y denotando como $x_i[1]$ el primer bit de ese elemento, se define $B_1(i) = x_i[1]$.

Posteriormente, para construir la fila B_{l+1} de la matriz, se realiza un reordenamiento de la secuencia basado en el orden establecido por la fila B_l. En este reordenamiento, se colocan a la izquierda todos los elementos cuyo bit correspondiente en B_l sea 0, y a la derecha todos aquellos cuyo bit sea 1, manteniendo el orden relativo entre los elementos que comparten el mismo bit en B_l. Haciendo este proceso de manera recursiva hasta el último nivel de la matriz, es decir, hasta que los símbolos ya no tengan más bits por procesar, se consigue construir completamente la matriz.

Finalmente, se construye un vector de valores que es fundamental para el correcto funcionamiento de la wavelet matrix. Para obtener este vector, se recorre cada fila y se cuenta la cantidad de ceros que posee cada una. Notar que, dado que cada fila es un bitvector, basta con calcular $rank_0(m)$ en el bitvector correspondiente a cada fila, guardando dicho valor asociado a su respectiva fila. Entonces, sea B_l la fila l-ésima de la matriz, y z_l el l-ésimo elemento del vector, entonces:

$$z_l = B_l.rank_0(m) \tag{2.14}$$

Un ejemplo visual de esta estructura se puede observar en la figura 2.6.

4	7	6	5	3	2	1	0	1	4	1	7
1	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**
3	2	1	0	1	1	4	7	6	5	4	7
1	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**1**
1	0	1	1	4	5	4	3	2	7	6	7
1	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**
0	4	4	2	6	1	1	1	5	3	7	7

Figura 2.6: Ejemplo de un wavelet matrix de la secuencia (4, 7, 6, 5, 3, 2, 1, 0, 1, 4, 1, 7) usando los primeros 3 bits de cada numero y marcando con una linea el número guardado por fila, extraído de [23].

Notar que el tamaño usado por la estructura en general es $O(m \cdot \lceil \log_2(\tau) \rceil)$, ya que la matriz ocupa $m \cdot \lceil \log_2(\tau) \rceil$ bits, pues tiene m columnas y $\lceil \log_2(\tau) \rceil$ filas, y el vector ocupa $\lceil \log_2(\tau) \rceil^2$ bits, dado que guarda un número por fila, habiendo $\lceil \log_2(\tau) \rceil$ filas, y cada número ocupa $\lceil \log_2(\tau) \rceil$ bits. Aquí es importante notar que $m \geq \tau$, pues si τ fuera mayor, habría más símbolos que elementos en la secuencia, por lo que se cumple que $m \cdot \lceil \log_2(\tau) \rceil \geq \lceil \log_2(\tau) \rceil^2$, y por lo tanto el tamaño total de la wavelet matrix es $O(m \cdot \lceil \log_2(\tau) \rceil)$.

En cuanto a las operaciones que soporta el *wavelet matrix* (al igual que el *wavelet tree*), son las operaciones de *access*, *rank* y *select*, pero a diferencia de los bitvectores normales, un *wavelet matrix* soporta estas operaciones a nivel de símbolos de una cadena S, es decir:

- access(S, i): retorna el i-ésimo elemento de la cadena S, el cual corresponde a un símbolo.
- rank$_a(S, i)$: retorna el número de ocurrencias del símbolo a en la cadena S dentro del rango $[1, i]$.
- select$_a(S, i)$: retorna la posición de la i-ésima ocurrencia del símbolo a en la cadena S.

Para la realización del access(S, i), se accede al i-ésimo bit de B_0. Si el valor de este bit es 0, el índice i a utilizar para la siguiente fila es $i = B_0.\text{rank}_0(i)$; en caso de que sea 1, el índice para la siguiente fila es $i = z_0 + B_0.\text{rank}_1(i)$. Luego, de forma equivalente a esta, se avanza fila por fila hasta llegar a la última fila, donde se puede conocer el símbolo accedido concatenando los bits obtenidos al acceder al i-ésimo bit de cada fila, construyendo el símbolo a medida que se recorre la matriz.

Notar que para realizar access(S, i) se requiere hacer un rank por cada fila, por lo que el costo de esta operación es $\lceil \log_2(\tau) \rceil \cdot O(\text{rank})$, siendo $O(\text{rank})$ el costo de realizar rank en el bitvector utilizado en la estructura.

Una operación un poco más difícil es el rank$_a(S, i)$, ya que no se puede calcular directamente la cantidad de ocurrencias de un símbolo dada una posición i como en el caso del access(S, i), pero se puede notar que si en la última fila del wavelet matrix seguimos calculando una siguiente fila, se obtiene una nueva fila imaginaria, como se puede ver en la figura 2.6, en la que todos los elementos con el mismo símbolo están contiguos (sus representaciones en esa última fila).

Y usando esta última fila imaginaria se puede calcular el rank$_a(S, i)$, en particular, con ella se puede calcular una operación intermedia, llamemos a esta operación posicion$_a(S, i)$, en donde dado un símbolo a se calcula la posición en esta fila imaginaria de la última ocurrencia de a en el intervalo $[1, i)$.

Para ello se accede al i-ésimo bit de B_0, si el valor del primer bit del símbolo a es 0, el i a utilizar para la siguiente fila es $i = B_0.\text{rank}_0(i)$, en caso de que sea 1, el i a utilizar para la siguiente fila es $i = z_0 + B_0.\text{rank}_1(i)$. Luego, de forma equivalente a esta se va avanzando fila por fila, hasta llegar a la última fila, en donde nuevamente se aplica este procedimiento, siendo finalmente el valor que se tiene en i el resultado de esta operación intermedia. Notar que esta operación toma una operación de rank (del bitvector) más que el tiempo que tarda access(S, i).

Una vez teniendo esta operación intermedia, la forma de calcular rank$_a(S, i)$ es calcular primero posicion$_a(S, i)$, que corresponde en esta fila imaginaria a la posición de la última ocurrencia del símbolo a en el intervalo $[1, i)$. Luego, también se calcula posicion$_a(S, 0)$, que corresponde en esta fila imaginaria a la posición previa del primer símbolo a.

De esta forma, se obtiene rank$_a(S, i)$ como:

$$\text{rank}_a(S, i) = \text{posicion}_a(S, i) - \text{posicion}_a(S, 0), \tag{2.15}$$

ya que, como los símbolos están contiguos en esta fila imaginaria, esta resta obtiene la cantidad de ocurrencias del símbolo a en el intervalo $[1, i)$.

Y la última operación que debe soportar es $\text{select}_a(S, i)$, en la cual también se utiliza la operación intermedia $\text{posicion}_a(S, i)$ descrita anteriormente. Para calcular $\text{select}_a(S, i)$, lo primero que se realiza es calcular $\text{posicion}_a(S, 0)$, que corresponde en la fila imaginaria a la posición previa del primer símbolo a, por lo que para llegar a la posición del i-ésimo elemento con el símbolo a en esta fila imaginaria, basta con sumarle i al resultado.

Una vez llegados a este punto, basta con devolverse por la wavelet matrix para llegar a la posición del i-ésimo elemento con el símbolo a en la secuencia. Para devolverse de una fila en el nivel $l + 1$ al nivel l, primero se observa la posición del elemento en la fila actual, digamos que esta posición es i, posteriormente se observa cuál es el l-ésimo bit del símbolo. Si este bit es 0, entonces la posición del elemento en la fila previa es $B_l.\text{select}_0(i)$. En caso de que este bit sea 1, entonces la posición del elemento en la fila previa es $B_l.\text{select}_1(i - z_l)$, siendo esta posición el nuevo i a utilizar para seguir subiendo en la wavelet matrix.

De esta forma se va subiendo aplicando este procedimiento de forma reiterativa hasta llegar a la primera fila, donde la posición del elemento en la fila corresponde también a la posición del elemento en la secuencia.

Finalmente, cabe señalar que esta estructura también habilita operaciones esenciales para llevar a cabo la versión del algoritmo LTJ utilizada en el *ring*, tales como *range-next-value* y *range-intersection-queries* [24].

2.5.7. Adaptive dynamic bitvector

Como se mencionó anteriormente, el resto de implementaciones del *ring* utilizaron bitvectors que imponían ciertas limitaciones en las características del *ring*. A pesar de esto, dichas limitaciones se deben a un problema intrínseco de los bitvectors y no a una mala elección. Sin embargo, recientemente Navarro desarrolló una nueva representación de bitvectors dinámicos, con la cual, dado un listado de operaciones en que se realizan q consultas por cada actualización, se obtiene un tiempo amortizado de $O(\log(n/q))$ para sus operaciones [7] y utiliza a lo más una cantidad de $4n + o(n)$ bits. Esto resulta especialmente beneficioso en casos donde las actualizaciones son poco frecuentes, pudiendo eliminar en gran medida las limitaciones del *ring* en estos escenarios.

Para conseguir esto, la estructura se basa en un árbol binario que contiene dos tipos de hojas: las hojas estáticas y las hojas dinámicas. Las hojas estáticas poseen respuestas precalculadas, por lo que no permiten actualizaciones directamente (teniendo que transformarse en hojas dinámicas cuando se requiere una actualización). Al ser estáticas, presentan un bajo costo para las operaciones de *rank* y *select*. Por el contrario, las hojas dinámicas, al no poseer respuestas precalculadas, permiten realizar actualizaciones de forma más eficiente, pero debido a su naturaleza dinámica, implican un mayor costo para las operaciones de *rank* y *select*.

Además de las hojas, los nodos internos del árbol binario, además de guardar punteros a su hijo izquierdo y a su hijo derecho, almacenan cuatro datos adicionales: el tamaño en bits del subárbol que tiene al nodo actual como raíz, la cantidad de unos que contiene este subárbol, la cantidad de hojas del subárbol (considerando que las hojas estáticas pueden contar como múltiples), y la cantidad de consultas desde la última actualización que han pasado por este nodo.

Usando esta estructura, se introducen dos operaciones que permiten conseguir el tiempo amortizado mencionado anteriormente:

- *Flattening*, que consiste en transformar un subárbol (con nodo raíz V) en una hoja estática. En particular, esta operación ocurre cuando el nodo V alcanza una cantidad de consultas (desde la última actualización) mayor o igual a la cantidad de bits del subárbol multiplicada por una constante fija (siendo esta constante un parámetro de la estructura).
- *Splitting*, que consiste en transformar una hoja estática en un subárbol. En particular, esta operación ocurre cuando se realiza una actualización sobre una hoja estática.

Pudiendo observar una representación de lo que realizan estas operaciones en la figura 2.7.

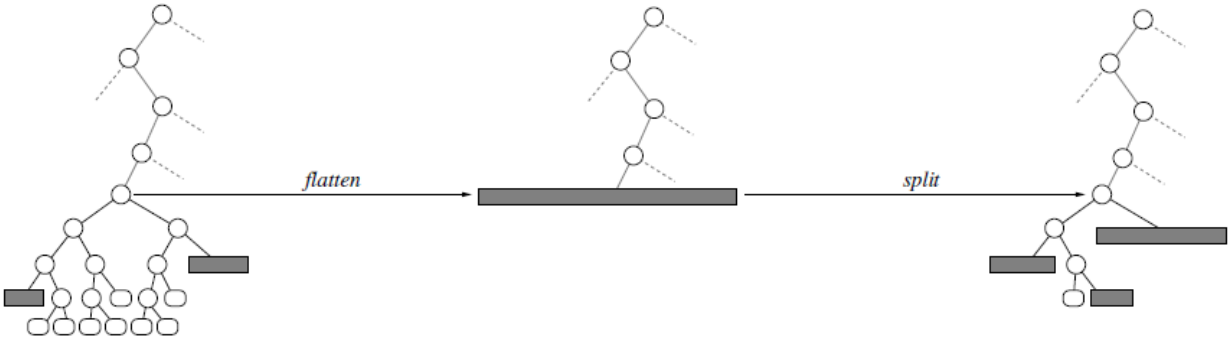

Figura 2.7: Ejemplo visual de flattening y splitting, extraido de [7]

Finalmente, cabe destacar que estas operaciones no solo sirven para transformar un tipo de hoja en otro, sino que también se utilizan en casos donde el árbol está desbalanceado. En dichos casos, se realiza un *flattening* en el nodo desbalanceado y posteriormente un *splitting* en el mismo nodo, logrando así balancear el árbol. De esta forma, al aplicar estas operaciones cuando corresponde, el árbol consigue los tiempos amortizados mencionados anteriormente.

2.6. Diccionarios compactos

Los diccionarios son estructuras que almacenan un mapeo entre strings e identificadores, permitiendo realizar las siguientes dos operaciones fundamentales:

- *locate*(s): función que recibe un string s y devuelve su identificador asociado en el diccionario (notando que el string s debe existir en el diccionario).

- *extract*(i): función que recibe un identificador i y devuelve su string asociado en el diccionario (notando que el identificador i debe existir en el diccionario).

Por lo tanto, un diccionario compacto es un diccionario en el que se aplican distintas técnicas para reducir el almacenamiento necesario, intentando generalmente mantener los tiempos de respuesta. Para más información y ejemplos de diccionarios compactos, se puede consultar [25].

El uso de un diccionario es necesario, ya que el *ring* requiere representar sus elementos mediante identificadores numéricos en lugar de strings. Esto permite facilitar ciertas operaciones clave en su funcionamiento y, además, obtener una mayor eficiencia en el uso de espacio. Dado que se pretende utilizar el *ring* sobre grafos de gran tamaño, emplear un diccionario tradicional implicaría un elevado consumo de memoria. Considerando que uno de los objetivos del *ring* es reducir el espacio al mínimo posible, se vuelve fundamental el uso de diccionarios compactos que optimicen al máximo la representación de los datos.

2.6.1. Plain Front Coding

Plain Front Coding (o PFC) [25], es un diccionario compacto que utiliza una técnica para comprimir los strings de un diccionario ordenado lexicográficamente. En particular, ya que el diccionario está ordenado lexicográficamente, es probable que dos strings consecutivos compartan un prefijo común. Por lo tanto, cada string del diccionario se codifica almacenando únicamente las diferencias con respecto al string anterior.

Cada string del diccionario se representa con dos valores: un número entero que indica la longitud del prefijo que comparte con el string anterior, y los caracteres restantes necesarios para completar la palabra (es decir, aquellos que vienen después del prefijo compartido). Un ejemplo de esta codificación se puede observar en la figura 2.8.

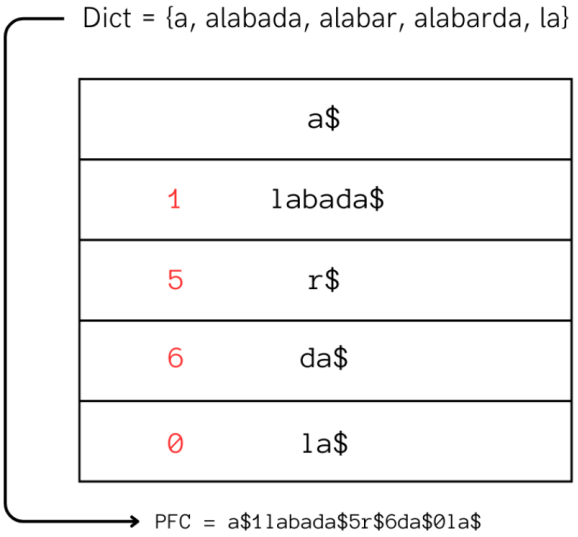

Figura 2.8: Ejemplo de compresión usando Plain Front Coding, extraido de [6]

Notar que este método, a pesar de comprimir de buena manera los strings, no cumple con mapear los strings a sus identificadores, por lo que se realiza una modificación en la que se concatena el identificador del string con la representación codificada del mismo. De esta forma, la relación entre string e identificador queda representada por tres valores que se concatenan.

Sin embargo, esto introduce un nuevo problema, ya que al ser una concatenación, es necesario poder diferenciar el final del identificador con el inicio de la longitud del prefijo compartido. Una primera solución sería fijar un tamaño constante (por ejemplo, un byte) para cada uno de estos valores. No obstante, esto impone una restricción en el tamaño del diccionario, ya que, por ejemplo, si se limita a un byte, tanto los identificadores como las longitudes de los prefijos estarían limitados a valores entre 0 y 255.

Para evitar esta limitación, se utiliza una codificación de enteros positivos llamada VByte [26], en la cual se representa un número utilizando uno o más bytes. Cada byte almacena 7 bits de información del número, utilizando el bit más significativo para indicar si ese byte es el último de la codificación (bit en 1) o si aún quedan bytes por leer (bit en 0). De esta forma se puede identificar claramente el final de cada número codificado, permitiendo así distinguir entre los tres elementos concatenados. Finalmente, los caracteres restantes que completan la palabra se codifican terminando con un carácter reservado, asegurando una delimitación clara entre los componentes, sin imponer restricciones al tamaño del diccionario.

Un ejemplo de codificación de un número usando VByte sería codificar el número 307, cuya representación en binario es 100110011. Primero, se extraen los primeros 7 bits menos significativos, es decir, 0110011. Como aún quedan bits por procesar, se antepone un 0 como bit más significativo (indicando que no es el último byte), obteniendo así el primer byte: 00110011.

Luego, se toman los siguientes bits. En este caso, sólo quedan dos bits: 10, los cuales se completan con ceros a la izquierda hasta formar 7 bits: 0000010. Como ya no quedan más bits por procesar, se antepone un 1 al principio (indicando que este es el último byte), obteniendo el segundo byte: 10000010. Por lo tanto, la representación VByte del número 307 son los dos bytes: 00110011 y 10000010.

Finalmente, sea D el conjunto de strings del diccionario ordenados lexicográficamente, en el que $D = s_0, s_1, ..s_U$ y donde $\forall i \in [0, U-1], s_i < s_{i+1}$, se definen las funciones:

- $\text{lcp}(s_i, s_j)$: Función que calcula el largo del máximo prefijo en común entre los strings s_i y s_j.
- $\text{suffix}(s, i)$: Función que retorna el sufijo del string s desde el índice i.
- $\text{encode}(x)$: Función que codifica el número x usando VByte.

Entonces, tomando los strings como arreglos de caracteres indexados desde 0, en donde x_i es el identificador correspondiente al string s_i, podemos definir el PFC modificado como la siguiente estructura recursiva:

$$PFC[0] = encode(x_0)|s_0 \tag{2.16}$$
$$PFC[i] = encoder(x_i)|encode(lcp(s_{i-1}, s_i))|suffix(s_i, lcp(s_{i-1}, s_i)) \tag{2.17}$$

Un ejemplo en donde se aplica la construcción anterior se puede observar en la figura 2.9.

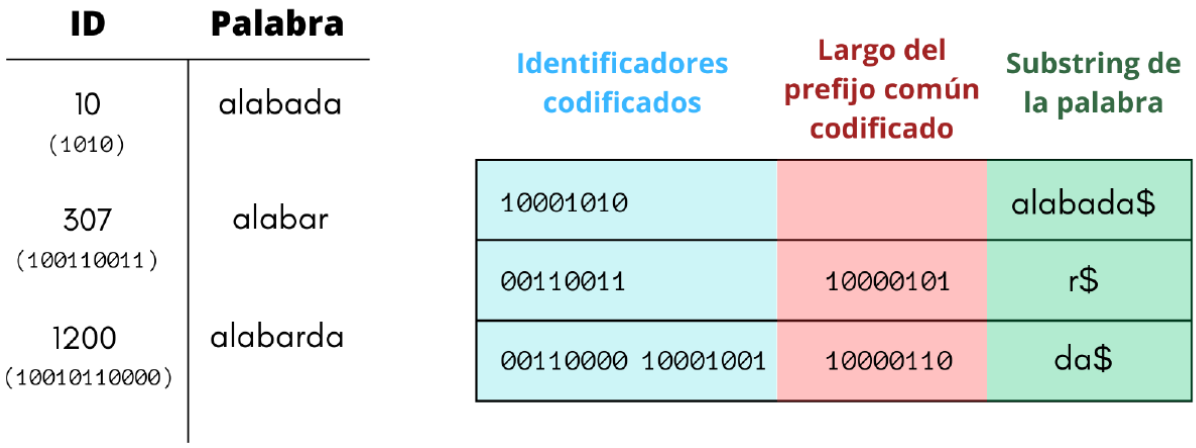

Figura 2.9: Ejemplo de un Plain Front Coding que concatena ids que usan VByte, extraido de [6].

Es importante señalar que, si bien esta estructura permite cumplir las propiedades de un diccionario y al mismo tiempo consigue una alta compresión de los strings, cuando se quiere hacer acceso a un elemento aleatorio del diccionario (algún PFC[i], para un i aleatorio), se requiere realizar la lectura de todas las cadenas anteriores, haciendo que el costo de descomprimir un string que se encuentra al final del diccionario sea sumamente costoso.

Una alternativa que se ha planteado es la utilización de *buckets*, en los que cada bucket es un PFC que posee una cantidad máxima de palabras, teniendo que buscar primero el bucket donde se almacena la palabra y luego buscar o insertar la palabra en el bucket. Para almacenar los distintos buckets se han hecho propuestas de arreglos [25] o de árboles binarios [6].

2.6.2. Árbol PFC

Debido a las ventajas que ofrece mantener el orden lexicográfico al realizar la división y fusión de los buckets, para la implementación del *ring* dinámico (hecha por Yuval Linker) se utilizó un diseño de árbol binario para los buckets. Este árbol binario tiene hojas que corresponden a diccionarios PFC, mientras que los nodos internos almacenan una palabra del diccionario que es lexicográficamente mayor que todas las palabras en el subárbol izquierdo y, a su vez, lexicográficamente menor que todas las palabras en el subárbol derecho.

Notar que si la palabra almacenada en los nodos internos es lexicográficamente la menor palabra del subárbol de su hijo derecho, entonces cumple con las restricciones mencionadas

anteriormente. Además, dado que en los diccionarios PFC ya se guarda la menor palabra en su cabecera, se puede acceder a dicha palabra en tiempo $O(1)$. Por lo tanto, en lugar de almacenar la palabra directamente en el nodo interno, se puede guardar un puntero a la hoja más a la izquierda del subárbol de su hijo derecho —que es precisamente la que contiene la menor palabra de ese subárbol—. De esta forma, se accede a la palabra en tiempo $O(1)$ almacenando únicamente un puntero por nodo interno, y si se requiere acceder a la palabra, se consulta directamente el PFC correspondiente.

De esta forma, dado un $P_{\max}$ y un $P_{\min}$, que corresponden a la cantidad máxima y mínima de palabras que puede contener el diccionario PFC de la hoja, si se tienen U palabras guardadas en este árbol PFC, entonces el tiempo promedio que tarda en buscar una palabra cualquiera en el árbol es $O\Big(\log\Big(\lceil\frac{U}{P_{\max}}\rceil\Big)\Big)$, pero dado que no es un árbol balanceado, el tiempo en el peor caso es $O\left(\frac{U}{P_{\max}}\right)$.

Otro problema que se produce de esta forma es que, si se quiere realizar *extract*(i) sobre esta estructura, no se puede saber en qué hoja se almacena el identificador i a buscar, lo que obliga a recorrer secuencialmente todas las hojas. Esto resulta especialmente problemático, considerando que la operación *extract*(i) es la más relevante para el caso de uso que se busca implementar.

Para solucionar este problema, Yuval Linker agrega a la estructura un arreglo A_{ID} [6] con tamaño U, donde el i-ésimo elemento del arreglo almacena un puntero al PFC que contiene el identificador i. En caso de que ningún PFC contenga ese identificador, el puntero se deja como *null*. Notar que se apunta a un PFC, por lo que todavía es necesario realizar una búsqueda secuencial dentro del PFC para encontrar el identificador y posteriormente reconstruir su palabra asociada.

Un último problema a notar es que, al ser un diccionario dinámico, debe aceptar eliminaciones. Sin embargo, actualmente si se elimina una palabra del diccionario, el elemento correspondiente en el arreglo A_{ID} queda con puntero *null*, lo cual es un comportamiento normal y esperado. Pero si el diccionario sigue en uso, todos estos elementos con punteros *null* se convierten en espacio reservado con contenido basura que no puede reutilizarse.

La solución a esto consiste en reutilizar estos elementos del arreglo, es decir, reutilizar identificadores correspondientes a palabras eliminadas. Para ello, un árbol PFC crea dos nuevos punteros: uno al primer elemento eliminado y otro al último elemento eliminado del arreglo. Luego, en vez de guardar un puntero *null* en estos elementos eliminados, se guarda un puntero al siguiente elemento a eliminar (excepto en el último, que se mantiene un puntero *null*). De esta forma se forma una cola que permite reutilizar los identificadores eliminados, optimizando el uso completo del arreglo.

Un ejemplo visual con todos los elementos de esta estructura puede verse en la figura 2.10.

Se debe mencionar que este arreglo no es una estructura compacta, a diferencia del resto de estructuras utilizadas en el *ring*, consumiendo un espacio $O(U)$. En particular, dado que los punteros tienen tamaño p, el tamaño máximo que puede alcanzar esta sección de la estructura es $2 \cdot U \cdot p$, ya que al ser un arreglo dinámico se reserva el doble de espacio cada vez que se llena, logrando así un tiempo amortizado constante para las inserciones.

Por otro lado, la operación *extract*(i) tiene un tiempo equivalente al que tarda realizar *extract*(i) sobre los PFC almacenados en las hojas, puesto que encontrar el puntero a la hoja correspondiente toma tiempo $O(1)$.

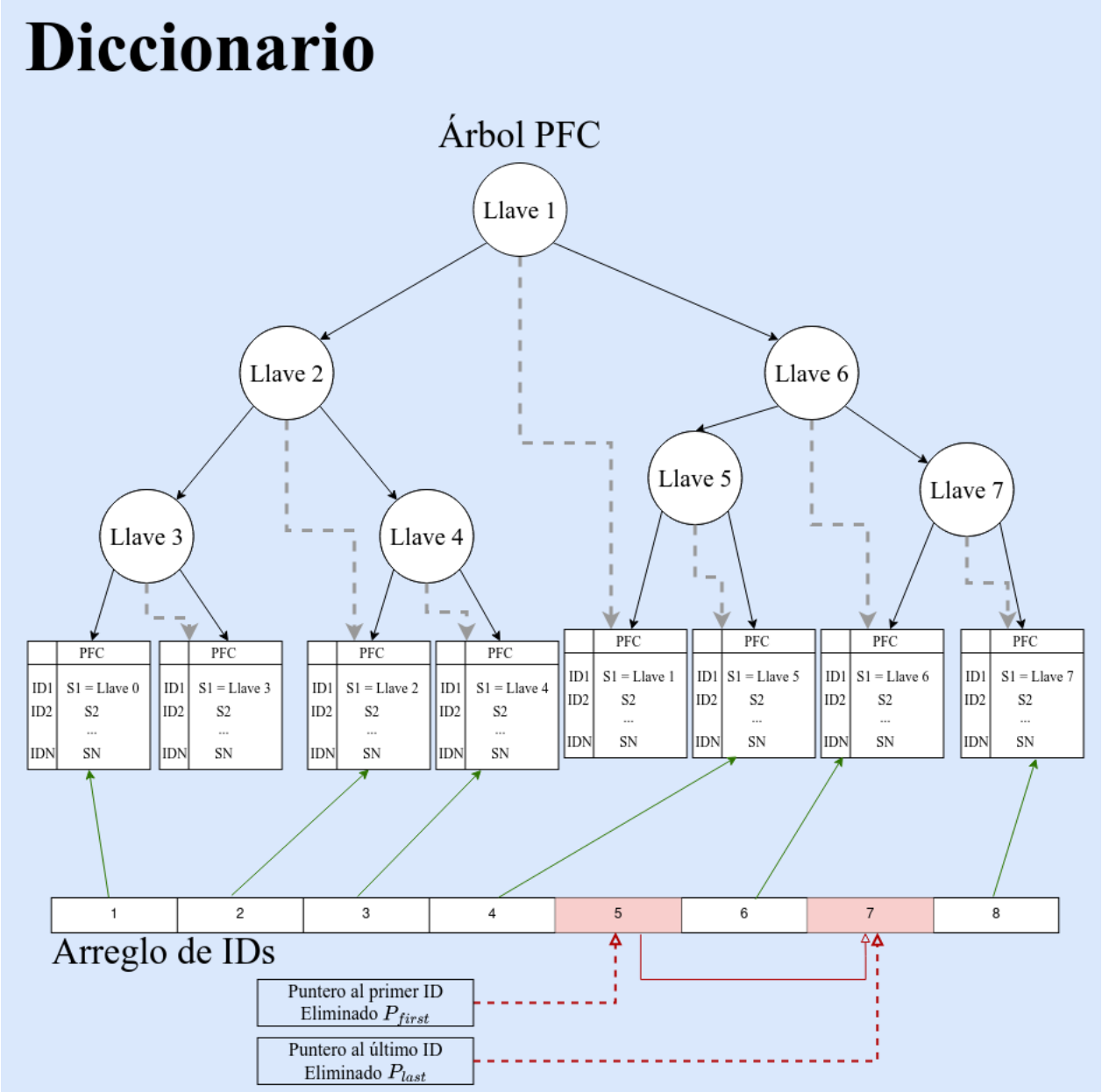

Figura 2.10: Esquema de la arquitectura del diccionario mencionado, con el árbol PFC, el arreglos de IDs y los punteros. Extraido de [6].

2.6.3. Actualizaciones

Como se mencionó anteriormente, se busca tener un diccionario que, además de tener las operaciones típicas de un diccionario, pueda realizar operaciones en donde se actualicen elementos del diccionario. En particular, se busca la implementación de las siguientes operaciones:

- *get_or_insert*(x): función que busca la palabra x en el diccionario y, en caso de no encontrarla, inserta una nueva palabra en el diccionario.
- *eliminate*(i): función que elimina la palabra que se identifica con el identificador i.

Para el caso de la operación *get_or_insert*(x), a diferencia de una búsqueda normal, se debe iniciar con la búsqueda de un nuevo identificador para la palabra. Para ello, se revisa si la cola contiene un identificador disponible; en caso afirmativo, se toma un identificador de la cola, y en caso contrario, se crea un nuevo identificador.

Posteriormente, se realiza una búsqueda normal de la palabra x en el árbol hasta llegar al diccionario PFC correspondiente, dividiéndose en dos casos:

El primero, en donde se encuentra la palabra x en el PFC, retornando el identificador de dicha palabra. El segundo, en donde al recorrer secuencialmente el PFC, antes de encontrar la palabra x se encuentra una palabra lexicográficamente mayor, o se llega al final del PFC, significando ambos casos que la palabra x no se encuentra en el diccionario, por lo que se debe insertar en el PFC.

Para insertar un string x en la posición j del PFC, donde la secuencia de strings almacenados en el PFC es $s_1, s_2, \ldots, s_n$, primero se debe calcular $new\_lcp_{j-1} = lcp(x, s_{j-1})$ y $new\_lcp_j = lcp(x, s_j)$. Teniendo esto, se reemplaza el valor actual de s_j en el PFC (que corresponde a $i_j \mid lcp_j \mid suffix(s_j, lcp_j)\$$) por:

$$i_{new}|new\_lcp_{j-1}|suffix(x, new\_lcp_{j-1})\$|i_j|new\_lcp_j|suffix(s_j, new\_lcp_j)\$ \tag{2.18}$$

Siendo i_{new} el identificador de x (obtenido al inicio de la operación) e i_j el identificador de la j-ésima palabra guardada en el diccionario. Notar que cuando se inserta en la primera o última posición del PFC se usa un reemplazo diferente; en particular, se omite el uso de new_lcp_{j-1} en caso de ser la primera posición (ya que no hay un elemento anterior), y para el caso de la última posición se omite lo puesto después de la primera llave (ya que no hay un siguiente elemento).

Algo a considerar durante la inserción de un nuevo elemento al PFC, es cuando la cantidad de palabras almacenadas en el PFC supera a P_{max}, en donde se convierte la hoja actual en un nodo interno con dos hojas con la misma cantidad de palabras cada una; en particular, la hoja izquierda se queda con la primera mitad y la hoja derecha con la mitad restante. Notar que al hacer esto también se deben cambiar los punteros en el arreglo A_{ID} asociados a las palabras de la hoja que fue dividida. Algo a mencionar es que en la práctica se crea un nuevo nodo y se reutiliza la hoja en vez de convertirla.

Para el caso de la operación de *eliminate*(i), lo primero a notar es que, como ya se tiene el identificador, se puede usar el arreglo A_{ID} para encontrar inmediatamente el PFC correspondiente a la palabra a eliminar (guardando la dirección de este PFC). Pero antes de borrar la palabra del PFC, primero se elimina la asociación en el arreglo A_{ID}, en particular, $A_{ID}[i]$ se deja como null y se encola la dirección del arreglo en la cola de identificadores libres, pudiendo luego de este proceso eliminar la palabra del PFC correspondiente.

Para eliminar un string con el identificador i en un PFC, lo primero es recorrer secuencialmente el PFC para buscar la posición del string. Una vez encontrado, se tiene tanto el string como la posición; digamos que para este caso su string es x y su posición en el PFC es j. Entonces, para eliminar el string x en la posición j, se necesita calcular primero $new\_lcp_{j-1} = lcp(x, s_{j-1})$ y $new\_lcp_{j+1} = lcp(x, s_{j+1})$, siendo estas las cantidades de prefijo que comparten con x. Luego, se calcula $new\_lcp = \min(new\_lcp_{j-1}, new\_lcp_{j+1})$, que corresponde al prefijo común que tiene la $(j-1)$-ésima y la $(j+1)$-ésima palabra del PFC. Finalmente, se elimina el valor actual de s_j y se reemplaza el valor de s_{j+1} por:

$$i_{j+1}|new\_lcp|suffix(s_{i+1}, new\_lcp)\$ \tag{2.19}$$

Habiendo nuevamente dos casos bordes, cuando se elimina en la primera o última posición del PFC, en caso de eliminar la palabra en la primera posición se modifica el segundo elemento para dejarlo como cabecera, y si es la palabra en la última posición se elimina directamente s_j sin luego tener que modificar ningún valor.

Algo a considerar durante la eliminación de un elemento del PFC, es cuando la cantidad de palabras almacenadas en el PFC es menor a P_{min}, en ese caso se deben fusionar tanto la hoja actual en donde se está eliminando como su hoja hermana, terminando con una única hoja que almacena todas las palabras de ambas hojas, reemplazando esta hoja la posición del padre de las anteriores hojas. Notar que al hacer esto también se deben cambiar los punteros en el arreglo A_{ID} asociados a las palabras de la hoja que fueron fusionadas, en particular, ahora todos deben apuntar hacia la hoja resultante de la fusión.

Capítulo 3

Problema

El *ring* se ha demostrado experimentalmente como una estructura altamente eficiente tanto en términos de espacio — pudiendo almacenar las tripletas de un grafo utilizando un orden de espacio menor que el resto de soluciones actuales — como en términos de tiempo para la realización de consultas utilizando BGP. En particular, el *ring* estático alcanza tiempos promedio por consulta cercanos a las soluciones más eficientes actuales, mientras que el *ring* dinámico logra tiempos cercanos a otras soluciones que tienen un orden de magnitud mayor en los tiempos que estos últimos [6].

Estos resultados se obtienen porque el *ring* utiliza el algoritmo Leapfrog Triejoin, mientras que sus estructuras base están construidas sobre estructuras compactas. De esta manera, el Leapfrog Triejoin aporta eficiencia temporal, y las estructuras base contribuyen a la eficiencia espacial, convirtiendo al *ring* en una solución muy sólida en comparación con otras bases de datos de grafos disponibles tanto en la academia como en el mercado.

Sin embargo, ambas versiones del *ring* presentan limitaciones que restringen su caso de uso. En el caso del *ring* estático, la incapacidad para aceptar actualizaciones en los datos implica tener que reconstruir la estructura completa ante cualquier modificación. Por su parte, el *ring* dinámico sufre una pérdida de eficiencia en los tiempos de consulta, requiriendo un orden de magnitud más tiempo que el *ring* estático u otras soluciones igual de eficientes. Esto limita su uso en escenarios típicos donde una base de datos de grafos recibe muchas más consultas que actualizaciones.

Ambos problemas se originan principalmente por las características de las estructuras compactas empleadas. Por ello, se propone utilizar un nuevo bitvector dinámico, en particular el *adaptive dynamic bitvector* desarrollado recientemente por Gonzalo Navarro [7], con el objetivo de mantener el dinamismo del *ring* dinámico y al mismo tiempo acercarse a los tiempos de consulta del *ring* estático.

Adicionalmente, el diccionario utilizado para traducir entre el alfabeto y los IDs empleados por el *ring* tiene una estructura interna en forma de árbol que puede desbalancearse, lo que podría afectar el rendimiento general. Por esta razón, se propone ajustar dicha estructura para evitar que se desbalancee.

La solución planteada se considerará satisfactoria si logra realizar operaciones de actuali-

zación, manteniendo un uso de espacio cercano al de las demás implementaciones del *ring* y tiempos de consulta próximos a los alcanzados por la versión estática.

Capítulo 4

Solución

En este capítulo, se abordará el diseño e implementación de la arquitectura del *ring* dinámico amortizado. Se proporcionará una descripción detallada de la arquitectura del *ring* dinámico amortizado, poniendo especial énfasis en los cambios realizados para reemplazar el bitvector utilizado e implementar el *adaptive dynamic bitvector*. Además, se explicará la mejora realizada al diccionario asociado al *ring* y las razones detrás de su elección. Por último, se presentarán algunas modificaciones que se tuvieron que hacer al *ring* dinámico para solucionar algunos errores encontrados.

4.1. Arquitectura del ring dinámico amortizado

La arquitectura del *ring* se basa en dos estructuras fundamentales, los wavelet matrix y los bitvectors. Por lo tanto, para asegurar el correcto funcionamiento de los algoritmos presentados en la implementación del ring dinámico desarrollado por Yuval Linker [6], es crucial que la implementación actual de dichas estructuras pueda llevar a cabo las mismas operaciones que sus contrapartes.

En particular, las operaciones que debe soportar el nuevo bitvector a utilizar son:

- $access(i)$: Retorna el i-ésimo bit del bitvector B.
- $rank_v(i)$: Retorna el número de ocurrencias del bit $v \in \{0, 1\}$, en el intervalo que hay entre el primer elemento y el i-ésimo elemento del bitvector B.
- $select_v(i)$: Retorna la posición de la i-ésima ocurrencia del bit $v \in \{0, 1\}$, en el bitvector B.
- $write_v(i)$: Reescribe el bit en la i-ésima posición del bitvector B con $v \in \{0, 1\}$, notando que como el bit ya existía no hay desplazamiento de los siguientes bits.
- $insert_v(i)$: Inserta un bit en la i-ésima posición del bitvector B con $v \in \{0, 1\}$, desplazando los elementos siguientes un bit hacia adelante.

- *remove*(i): Remueve el bit en la i-ésima posición del bitvector B, desplazando los elementos siguientes un bit hacia atrás.

De aquí se puede notar que las primeras tres operaciones corresponden a las esenciales de un bitvector, y las últimas tres operaciones son las necesarias para que pueda soportar el dinamismo, explicando más adelante cómo el adaptive dynamic bitvector consigue realizar cada una de estas operaciones.

En cuanto a la implementación del wavelet matrix a utilizar para el *ring* dinámico amortizado, se decide mantener la misma implementación que la del *ring* dinámico, es decir, una variación hecha por Yuval Linker de la librería DYNAMIC de Nicola Prezza [19].

Las principales razones para mantener esta implementación se deben, en primer lugar, a que los wavelet matrix, al igual que el *ring*, son una estructura que utiliza como pieza clave en su construcción a los bitvectors, haciendo que varias de sus características se basen en el bitvector utilizado para su construcción.

Bajo este contexto, es especialmente útil la implementación realizada por Nicola Prezza, en la que se consigue abstraer la implementación del wavelet matrix a cualquier bitvector, pudiendo cambiar fácilmente el bitvector utilizado.

Otra ventaja de utilizar la variación hecha por Yuval Linker es que, al ser utilizada para implementar el *ring* dinámico, se incorporaron operaciones claves para el correcto funcionamiento del *ring* que no estaban en su versión original, como *select next*, *range next value* o *all values in range*.

4.2. Adaptive dynamic bitvector

En el *ring* dinámico, el bitvector utilizado era uno basado en la utilización de B-Trees, en donde las hojas eran bitvectores pequeños que guardaban la información y se utilizaba el árbol para realizar las operaciones de los bitvectors de la forma más eficiente posible, pudiendo configurar tanto la cantidad de hijos por nodo, como la cantidad de bits almacenados en sus hojas.

Ofreciendo esta solución, para un bitvector de tamaño m un tiempo de $O(\log(m))$ para el $rank_v(i)$ y $select_b(i)$, debido a que para poder realizar estas operaciones, se necesita descender por el árbol hasta la hoja correcta y luego realizar la operación correspondiente en esa hoja, notar que este tiempo termina siendo constante pero depende de la cantidad de bits almacenados en las hojas.

De forma similar, el adaptive dynamic bitvector también propone la utilización de un árbol, pero en este caso se propone un árbol binario balanceado con la particularidad de poder tener dos tipos de hojas con características distintas, que se vayan intercambiando según las necesidades que se tengan del árbol.

Por un lado están las hojas dinámicas que almacenan una secuencia corta de bits, manejando las consultas escaneando su secuencia (producto de su tamaño), pero que como dice

su nombre permiten el dinamismo, refiriéndose a ellas usualmente como hoja dinámica (o *leaf*). Y por otro lado están las hojas estáticas que almacenan una secuencia larga de bits, manejando las consultas como un bitvector estático, es decir, en tiempo constante (o muy cercano en la práctica), pero sin poder tener modificaciones, refiriéndose a ellas usualmente como hoja estática (o *static*).

Para pasar de un tipo de hoja a la otra, existen dos operaciones claves: el *flattening*, que consiste en convertir todo un subárbol en una hoja estática, realizando esta operación cuando se procesan las suficientes consultas consecutivas para amortizar el costo de reconstruir el subárbol como una hoja estática; y el *splitting*, que consiste en transformar una hoja estática en un subárbol, realizada principalmente cuando se tiene que hacer una modificación de un elemento en la hoja estática. Se puede observar un ejemplo visual de ambas operaciones en la figura 2.7.

Además de las hojas mencionadas anteriormente, el árbol del adaptive dynamic bitvector se construye a través de nodos internos (o también llamados *dynamic*). Llamemos a un nodo interno cualquiera del árbol como v, entonces para que el árbol funcione de manera correcta, este nodo tiene que tener almacenado los siguientes valores:

- $v.left$: Hijo izquierdo de v, pudiendo ser un nodo interno o una hoja.
- $v.right$: Hijo derecho de v, pudiendo ser un nodo interno o una hoja.
- $v.size$: Cantidad de bits que se ven representados por el subárbol que nace en v.
- $v.ones$: Cantidad de unos que se ven representados por el subárbol que nace en v.
- $v.leaves$: Cantidad de hojas que posee el subárbol que nace en v, contando a las hojas estáticas como varias hojas distintas.
- $v.queries$: Cantidad de consultas que pasaron por v desde la última modificación que pasó por v (partiendo en 0 cuando se crea).

Notar que para mantener balanceado el árbol se utiliza un balanceo por peso [27][28], en particular, dada una constante $\alpha \in \left(\frac{1}{2}, 1\right)$, se restringe que cada nodo interno cumpla con que $v.left.size \leq \alpha \cdot v.size$ y $v.right.size \leq \alpha \cdot v.size$. Para ello, cada vez que se detecte un desbalanceo en el árbol, en vez de hacer una reconstrucción sobre el subárbol sesgado para volver a tener un árbol balanceado, se realiza un *flattening* sobre el subárbol, consiguiendo que ya no haya desbalanceo. Notar que si luego la hoja estática resultante del *flattening* es modificada, se usa *splitting*, terminando reconstruyendo el subárbol, puesto que el *splitting* crea un subárbol perfectamente balanceado, cumpliendo por ende con la restricción.

Usaremos este mismo mecanismo para restringir que $v.leaves = O\left(\frac{v.size}{w^2}\right)$, siendo w^2 el tamaño máximo que puede tener una hoja dinámica. Notar que debido a esta restricción que reduce la cantidad de hojas demasiado vacías, y ya que la cantidad de bits que utiliza un nodo interno es $O(w)$, se consigue que el árbol utilice $O\left(\frac{n}{w}\right)$ bits extras para almacenar los nodos internos. Esto es así pues la cantidad de nodos internos en un árbol binario es equivalente a la cantidad de hojas del árbol menos uno.

4.2.1. Consultas

Para la realización de las operaciones típicas de un bitvector, es decir, para las operaciones de $access(i)$, $rank_v(i)$ y $select_v(i)$, se sigue el mecanismo estándar del resto de bitvectors dinámicos que utilizan árboles binarios [29].

Para resolver $access(i)$, se recorre el árbol desde la raíz hasta una hoja, habiendo tres casos posibles. El primero es si se está en un nodo interno e $i \leq v.left.size$, en tal caso se busca $access(i)$ sobre el subárbol del hijo izquierdo. El segundo caso es si estamos en un nodo interno e $i > v.left.size$, en tal caso se redefine i como $i = i - v.left.size$ y posteriormente se busca $access(i)$ sobre el subárbol del hijo derecho. Hacemos esto reiterativamente hasta llegar a una hoja, siendo este el tercer caso, en donde se realiza $access(i)$ dentro de la hoja, completando la operación dentro de la hoja en tiempo $O(1)$. Notar entonces que la operación entera de $access(i)$ toma a lo más $O(\log(n))$, puesto que cada nivel del árbol cuesta $O(1)$ y, como se usa un árbol balanceado, a lo más hay $\log(n)$ niveles.

En el caso de $rank_1(i)$, se hace de forma muy similar a $access(i)$ pero manteniendo un contador de la cantidad de unos que hay hasta el i-ésimo elemento del bitvector. Por lo que, de forma muy similar, se recorre el árbol desde la raíz hasta una hoja, habiendo tres casos posibles. El primero es si estamos en un nodo interno e $i \leq v.left.size$, en tal caso se busca $rank_1(i)$ sobre el subárbol del hijo izquierdo. El segundo caso es si estamos en un nodo interno e $i > v.left.size$, en tal caso redefinimos i como $i = i - v.left.size$, luego al contador de unos se le suma $v.left.ones$ y finalmente se busca $rank_1(i)$ sobre el subárbol del hijo derecho. Hacemos esto reiterativamente hasta llegar a una hoja, siendo este el tercer caso, en donde se realiza $rank_1(i)$ dentro de la hoja, siendo este rank equivalente a haber contado la cantidad de unos dentro de la hoja hasta el i-ésimo bit. Finalmente, sumando este resultado al contador que se tenía, se obtiene la cantidad de unos que hay hasta el i-ésimo bit pero de todo el bitvector, siendo este número lo que se devuelve en el $rank_1(i)$ sobre el bitvector. Notar que algo equivalente se puede hacer con $rank_0(i)$ pero reemplazando el contador de unos por contador de ceros, aunque en este caso es más rápido y sencillo calcularlo como $rank_0(i) = i - rank_1(i)$.

Finalmente, para $select_1(i)$ se hace de forma análoga al $rank_1(i)$, pero manteniendo un contador sobre la cantidad de bits recorridos y guiándose por la cantidad de unos para recorrer el árbol. Por lo que se recorre el árbol desde la raíz hasta una hoja, habiendo tres casos posibles. El primero es si estamos en un nodo interno e $i \leq v.left.ones$, en tal caso se busca $select_1(i)$ sobre el subárbol del hijo izquierdo. El segundo caso es si estamos en un nodo e $i > v.left.ones$, en tal caso se redefine i como $i = i - v.left.ones$, luego al contador de bits se le suma $v.left.size$, y finalmente se busca $select_1(i)$ sobre el subárbol del hijo derecho. Haciendo esto reiterativamente hasta llegar a una hoja, siendo este el tercer caso, en donde se realiza $select_1(i)$ dentro de la hoja, siendo este select equivalente a haber contado la cantidad de bits dentro de la hoja hasta el i-ésimo uno. Finalmente, si se suma este resultado al contador que se tenía, se obtiene la cantidad de bits que hay hasta el i-ésimo uno pero de todo el bitvector, siendo este número lo que se devuelve en el $select_1(i)$ sobre el bitvector. Notar que algo equivalente se puede hacer con $select_0(i)$ pero reemplazando los unos por ceros, o equivalentemente reemplazar los usos de $v.left.ones$ por $v.left.size - v.left.ones$.

Notar que tanto la operación de *rank* como *select* toman un tiempo de $O(\log(n))$ más el

tiempo que se tarda en realizar la misma operación pero dentro de la hoja, siendo $O(\log(n))$ el tiempo que tarda en recorrer el árbol. Y como ya se ha mencionado previamente, a nivel teórico una hoja que usa un bitvector estático puede conseguir solucionar todas estas operaciones en tiempo $O(1)$, utilizando una precomputación adecuada en tiempo lineal [16]. En la sección 2.5.2 se puede ver un ejemplo para la operación de *rank*. Por el contrario, para la hoja dinámica que contiene $O(b) = O(w^2)$ bits, toma un tiempo $O(w)$ para realizar las operaciones de *rank* y *select*, utilizando la técnica de popcounting, que permite contar palabras de w bits en tiempo constante [7].

4.2.2. Actualizaciones

Para la realización de las operaciones de actualización del bitvector, es decir, $write_v(i)$, $insert_v(i)$ y $remove(i)$, se sigue el mismo principio que en trabajos anteriores de bitvectors usando árboles binarios [29].

Para realizar la operación de $write_v(i)$ sobre un bitvector que usa árbol binario, se recorre el árbol de forma equivalente a como se hace con la operación de $access(i)$, solo que en vez de retornar el bit en la i-ésima posición cuando llega a una hoja del árbol, se reescribe el i-ésimo bit en la hoja correspondiente, y se actualiza el valor de $v.ones$ cuando se regrese por los nodos en los que se pasó al recorrer el árbol. Notar que como se usa una recursión para llegar a la hoja, basta con actualizarlos cuando se esté regresando de la recursión. Luego, al igual que $access(i)$, esta operación toma $O(\log(n))$ puesto que el árbol es balanceado tanto al recorrer el árbol como al regresar de la recursión. Este balanceo se consigue mantener, ya que la operación $write_v(i)$ solo modifica los valores, no la cantidad de bits.

Sin embargo, para nuestro caso, hay que agregar un comportamiento especial para cuando la hoja a la que se llega es una hoja estática, ya que como se mencionó anteriormente, estas hojas no aceptan modificaciones. En este caso, se realizará un split (o también llamado splitting) sobre la hoja estática en la posición i, con la cual se creará un camino hacia una hoja dinámica que contiene al i-ésimo bit, habilitando con ello la modificación del mismo. Notar que más adelante se explicará el funcionamiento del split.

Para el caso de las operaciones de $insert_v(i)$ y $remove_v(i)$, se realiza de forma análoga a $write_v(i)$, teniendo que recorrer el árbol hasta la hoja que contiene al i-ésimo bit, con la diferencia que para el $insert_v(i)$ en vez de modificar el i-ésimo bit se realiza la inserción del bit v en esa posición, y para el $remove_v(i)$ en vez de modificar el i-ésimo bit se realiza la eliminación del mismo. Luego, al igual que en $write_v(i)$, se tienen que actualizar valores al regresar por los nodos, pero esta vez se tienen que actualizar tanto el valor de $v.ones$ como el de $v.size$. Notar que al eliminar o insertar bits se tienen que desplazar en una posición los bits siguientes, siendo hacia adelante para la inserción y hacia atrás para la eliminación. En particular, si se tienen que desplazar una cantidad b de bits, esto se realizará en un tiempo $O(b/w)$, ya que se desplazan los bits palabra por palabra.

Ya que esta vez las operaciones descritas anteriormente sí afectan el tamaño del árbol, pueden ocurrir en la hoja afectada por la operación los casos de desbordamiento para la operación de $insert_v(i)$, y subdesbordamiento para la operación de $remove_v(i)$. Dándose un desbordamiento cuando se inserta un bit en una hoja h que ya contiene la máxima cantidad

b de bits. Para resolver este problema se divide la hoja en dos, quedándose cada una con $\lfloor b/2 \rfloor$ y $\lfloor b/2 \rfloor + 1$ bits respectivamente, y convirtiendo a h en un nodo interno que es padre de ambas hojas. Y dándose un subdesbordamiento cuando se elimina un bit en una hoja h que solo contenía ese único bit. Para resolver este problema se elimina la hoja h (puesto que ya no tiene bits), se elimina a su nodo padre que corresponde a un nodo interno, y finalmente el otro hijo de su padre se convierte en hijo de su abuelo, sustituyendo de esta forma al nodo interno eliminado.

También se agrega una nueva restricción, donde dada una constante $\gamma \in \left[\frac{1}{2}, 1\right[$ y b la cantidad máxima de bits que puede tener una hoja, si la eliminación del bit de una hoja dinámica que tiene de hermano a otra hoja dinámica, hace que la suma de bits de ambas hojas sume a lo más $\gamma \cdot b$ bits, entonces ambas hojas se concatenan en una sola, por lo que se reemplaza al padre de las antiguas hojas con esta nueva hoja, eliminando posteriormente al padre. Notar que una vez que se elimina un bit, esta nueva operación se puede ir aplicando recursivamente mientras se van fusionando las hojas dinámicas, en particular esto da como resultado el invariante de que todo nodo interno v cumple con que $v.size \geq \gamma \cdot b$. Notar que de esta forma el tamaño mínimo del bitvector estático que se obtiene de realizar flattening sobre cualquier nodo interno es mayor o igual a $\gamma \cdot b$, siendo este tamaño el mínimo que se permitirá para las hojas estáticas.

4.2.3. Flattening

Como ya se ha dicho anteriormente, todos los nodos internos van a tener una variable llamada *queries*, en donde se almacenará un contador con la cantidad de consultas que han pasado por ese nodo interno, siendo las operaciones de consultas $access(i)$, $rank_v(i)$, y $select_v(i)$. A su vez, cada vez que una operación de actualización atraviese un nodo interno, se va a reiniciar este contador a cero.

Si dada una constante θ en el modelo, se cumple que después de una consulta un nodo interno v tiene su $v.queries \geq \theta \cdot v.size$, entonces se convierte todo el subárbol que nace del nodo v en una hoja estática, llamando a este proceso como *flattening*.

Para la realización del *flattening*, se van recorriendo todas las hojas del subárbol que nace del nodo v, escribiendo los bits encontrados palabra por palabra en un nuevo arreglo, que luego va a ser usado para construir una hoja estática. El tiempo que se tarda en realizar esta operación es $O(v.size)$, ya que tiene que escribir palabra por palabra los $v.size$ bits del subárbol. Sin embargo, este costo se ve amortizado por las $\theta \cdot v.size$ consultas previas requeridas para realizar el *flattening* [7].

Notar que esta operación mantiene temporalmente tanto a las hojas del subárbol como al nuevo arreglo del bitvector estático, incrementando durante ese intervalo el uso de espacio hasta en $v.size$ bits, haciendo que si intentáramos hacer esta operación en la raíz del árbol se puedan ocupar hasta n bits adicionales, siendo n la cantidad total de bits que posee el bitvector. Para evitar el caso anterior y conseguir un uso de espacio máximo de $(1+\epsilon) \cdot n$ bits (como es deseado), se introduce una nueva restricción en la que si un nodo interno v tiene un $v.size > \epsilon \cdot n$, entonces no puede ser realizada la operación de *flattening* sobre ese nodo, consiguiendo con esto un espacio máximo de $(1 + \epsilon) \cdot n + O(n)$ bits, incluso cuando ϵ es una

constante [7].

4.2.4. Splitting

Para el caso de la operación del *split*, dado una hoja estática v y una posición i en el bitvector que está dentro de esta hoja, llamaremos realizar *split* sobre v en la posición i, al proceso iterativo en donde se va dividiendo el bitvector de la hoja v en mitades, formando con la mitad que no contiene al i-ésimo bit una hoja estática, siendo esta hoja hijo de un nuevo nodo interno que ocupa el puesto de v en el árbol (se puede decir que v se transforma en un nodo interno), y tratando a la mitad restante como una hoja estática a la que inmediatamente se le hace *split* en la posición i, haciendo este proceso hasta que la hoja resultante sea lo suficientemente pequeña para no poder ser una hoja estática, recordando que las hojas estáticas tienen un tamaño mínimo definido de $\gamma \cdot b$.

Notar que al hacer *split* sobre una hoja estática, el subárbol resultante de la operación mantiene la misma cantidad de bits, haciendo que el nuevo nodo interno en la posición de v tenga el mismo $v.size$ que antes de *split*, por lo que no se ve afectado el balance del árbol. Finalmente, al igual que en la operación anterior, realizar *split* toma un tiempo $O(v.size)$, ya que se copia el arreglo de la hoja estática en bloques de w bits hacia las hojas del subárbol resultante al hacer *split*, además de realizar la construcción de los diversos bitvectores en las hojas, teniendo en el peor caso un costo de $\Theta(n)$, pero que consigue tener costos logarítmicos al amortizarse con el resto de operaciones [7].

4.2.5. Implementación

Para la implementación del adaptive dynamic bitvector, hay que mencionar en primer lugar que este nuevo *ring* dinámico mejorado se va a construir en base al *ring* dinámico hecho por Yuval Linker, pudiendo de esta forma reutilizar componentes claves en su creación, haciendo que la solución se tenga que implementar en C++ [6]. Si bien la principal razón por la que se implementa en C++ es lo dicho anteriormente, este lenguaje tiene varias ventajas que lo vuelven ideal para realizar la implementación del nuevo *ring* dinámico mejorado, como lo son su alto rendimiento al compilar un código máquina optimizado, y su facilidad para controlar los recursos.

Otra cosa a mencionar, es que la implementación del adaptive dynamic bitvector se basa en la realizada por Gonzalo Navarro en el lenguaje C [7]. Notar que esto nuevamente favorece la implementación del nuevo *ring* dinámico mejorado en C++, pues ambos lenguajes son bastante compatibles, facilitando la migración de un lenguaje a otro.

En cuanto a la implementación en sí, se usan palabras en la máquina de 64 bits para tener estructuras de datos livianas, pues el tamaño de estas afecta el tiempo (y el tamaño) con el que se construyen las estructuras de datos necesarias por las hojas estáticas, afectando finalmente el rendimiento general de la estructura, pues se tienen que ir construyendo en el instante en que se realiza la operación.

Para la implementación de las hojas estáticas como tal, se realiza una implementación muy similar a la mencionada en la sección 2.5.2, pero fijando el tamaño de los superbloques

a 2^{16} bits, el tamaño de los bloques a 256 bits y sin usar una respuesta precalculada en los bloques, ya que como tiene tamaño fijo solo se van a necesitar como máximo 4 accesos para realizar el conteo de sus unos, usando *popcount* en las cuatro palabras del bloque.

Notar que el *select* mencionado en la sección 2.5.2 y el cual es implementado en el adaptive dynamic bitvector, no es de tiempo constante como lo son las implementaciones más eficientes. Esto se debe a que si se usara una de esas implementaciones, los tiempos de construcción de la hoja estática al realizar *flattening* provocaría un aumento mayor al tiempo extra que supone no ser de tiempo constante, quedándose finalmente con la implementación mencionada en esa sección. Una mejora que se podría realizar sería implementar el *select* propuesto por Pandey, Bender y Johnson [30], donde realizan una implementación más eficiente, pero que usa instrucciones que dependen de tener una arquitectura relativamente moderna en el procesador; en particular, las instrucciones nuevas requeridas son *PDEP* y *TZCNT*.

En cuanto a las hojas dinámicas, se usa un tamaño máximo de 2048 bits, o equivalentemente un tamaño máximo de 32 palabras de 64 bits cada una. Notar que a diferencia de la teoría, cada vez que se inicia una hoja dinámica se le reserva la cantidad máxima de palabras, esto con el fin de evitar hacer re-asignaciones frecuentes en el tamaño de la hoja dinámica, ya que es una operación demasiado costosa. Notar que cuando se hace un *rank*, *select* o alguna actualización se recorre palabra por palabra, con un máximo de 32 palabras.

En cuanto a los valores de constantes usados para la implementación, a continuación se hace un listado con una pequeña explicación de la constante junto a su valor usado:

- ϵ: Es la constante que define cuál es la proporción máxima para realizar *flattening* en un nodo interno, en particular, se fija esta constante en 0.1 significando que si un nodo interno v tiene un $v.size > \epsilon \cdot n$, no se puede realizar *flattening* sobre ese nodo.

- α: Es la constante que define el nivel de balance que se quiere en el árbol, implicando un valor más cercano a 0.5 un árbol más balanceado, y un valor más cercano a 1 un árbol más desbalanceado, en este caso se escoge un valor de 0.65, lo cual da un árbol relativamente bien balanceado, pero no tan estricto como para estar constantemente balanceándose.

- γ: Es la constante que define el tamaño mínimo de una hoja estática, aunque también se utiliza para saber cuándo fusionar dos hojas dinámicas, en particular, esto se hace cuando la cantidad de bits entre ambas hojas es menor que el tamaño mínimo de una hoja estática. Para este caso se fija esta constante en 0.75, siendo por ende el tamaño mínimo de una hoja estática 1536.

- θ: Es la constante que regula la frecuencia con la que se realiza la operación de *flattening*, en particular, dado un nodo v se realiza *flattening* si $v.queries \geq \theta \cdot v.size$, notando que si θ posee un valor muy alto, entonces el árbol va a permanecer más tiempo en forma dinámica, afectando el tiempo de las consultas que se realizaron durante ese intervalo extra que permaneció en forma dinámica; por el contrario, si el valor es muy bajo, podría darse el caso de estar gastando tiempo haciendo *flattening* en sitios donde se va a tener que realizar *splitting* momentos después. Estos costes que se producen en ambos extremos hacen que a priori sea difícil ajustar un valor concreto para el caso general, sobre todo cuando los resultados se ven afectados de la proporción de consultas

realizadas por actualización, por lo que se va a dejar como una variable a decidir cuando se ejecuta una serie de consultas/actualizaciones sobre el *ring* dinámico mejorado.

- $TrFactor$: Una forma de evitar desbordamientos al insertar es cuando la hermana de la hoja dinámica actual también es una hoja dinámica, entonces se pueden transferir bits de la hoja que se está desbordando a su hermana evitando el desbordamiento (y ayudando a ser un árbol más balanceado), pero para aplicar esto correctamente, se define una constante de transferencia, en donde solo se hace la transferencia de bits si es que esta transferencia de bits es mayor a $TrFactor \cdot 2048$, siendo 2048 el tamaño máximo de una hoja dinámica. Fijando en este caso esa constante como 0.125.

4.3. Diccionario compacto

Como ya se mencionó en la sección 2.6.1, el *ring* requiere la utilización de números en su estructura interna, por lo que para el *ring* dinámico implementado por Yuval Linker se utilizó un diccionario compacto, en particular, fue un árbol PFC que además usaba como estructuras auxiliares tanto a un arreglo como a dos punteros, pudiendo encontrar más detalles en la sección 2.6.2, o en el trabajo realizado del *ring* dinámico realizado por Yuval Linker [6].

Sin embargo, uno de los problemas que todavía posee este diccionario compacto, es que el árbol utilizado en su implementación no garantiza ser balanceado, en particular, si se van insertando las palabras en un orden alfabético, se termina con una lista enlazada que tarda un tiempo $O\left(\frac{U}{P_{max}}\right)$, siendo U la cantidad de palabras y P_{max} la cantidad máxima de palabras que pueden tener los PFC de las hojas.

4.3.1. Árbol AVL

Para solucionar esto se propone reemplazar el uso del árbol actual, por un árbol que se pueda balancear mientras se van insertando y eliminando elementos del diccionario. En particular, se decide reemplazar por el árbol AVL, que posee un balanceo más estricto en comparación a otros árboles similares (como el árbol rojo-negro), teniendo una altura máxima muy cercana a $\log\left(\frac{U}{P_{max}}\right)$, y garantizando por ende tiempos $O\left(\log\left(\frac{U}{P_{max}}\right)\right)$ para todos los casos. Notar que balancear el árbol va a hacer que este se demore más tiempo en las actualizaciones, al tener que verificar las reglas de balanceo, así como también en el espacio que utiliza, ya que a cada nodo se le va a agregar un nuevo valor interno que guarda la altura del subárbol generado desde ese nodo.

En cuanto al funcionamiento de los árboles AVL, estos se basan en agregar tres operaciones a los nodos, las cuales permiten mantener el árbol balanceado. En particular, dado un nodo v, se pueden definir estas nuevas operaciones como:

- $v.height$: Esta operación devuelve la altura del subárbol formado por el nodo v, o equivalentemente, la cantidad de nodos por los que se pasa al recorrer el camino más largo hacia una hoja. Notar que para resolver esta operación en tiempo constante, se agrega una variable que guarda la altura del subárbol nacido en v.

- *rotacion_izquierda*: Esta operación realiza un reordenamiento del subárbol que nace del nodo v, manteniendo el orden de los elementos guardados en el árbol, pero entregándole más elementos a la rama izquierda del subárbol. En particular, el nuevo nodo raíz del subárbol es el antiguo hijo derecho de v, quien tendrá como hijo derecho su antiguo hijo derecho, y como hijo izquierdo al nodo v, el cual a su vez tiene como hijo izquierdo su antiguo hijo izquierdo y como hijo derecho al antiguo hijo izquierdo de su antiguo hijo derecho. Un ejemplo de esto se puede observar en la figura 4.1.

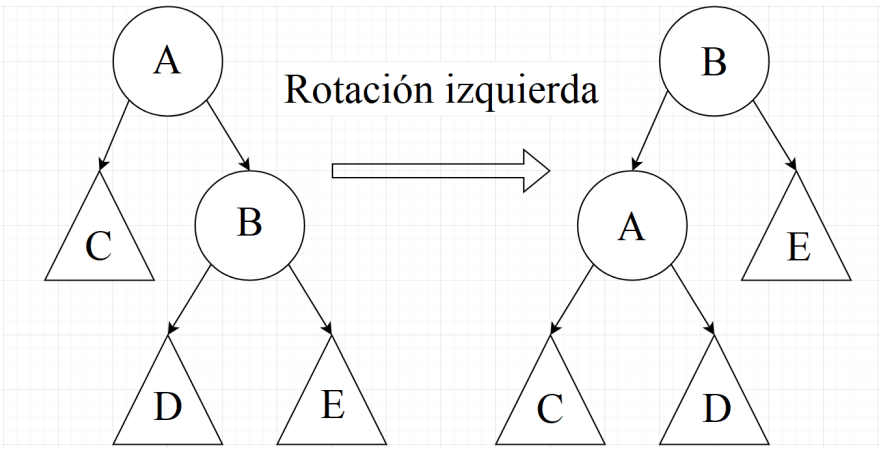

Figura 4.1: Ejemplo de realizar una rotación izquierda en el nodo A del árbol.

- *rotacion_derecha*: Esta operación realiza un reordenamiento del subárbol que nace del nodo v, manteniendo el orden de los elementos guardados en el árbol, pero entregándole más elementos a la rama derecha del subárbol. En particular, el nuevo nodo raíz del subárbol es el antiguo hijo izquierdo de v, quien tendrá como hijo izquierdo su antiguo hijo izquierdo, y como hijo derecho al nodo v, el cual a su vez tiene como hijo derecho su antiguo hijo derecho y como hijo izquierdo al antiguo hijo derecho de su antiguo hijo izquierdo. Un ejemplo de esto se puede observar en la figura 4.2.

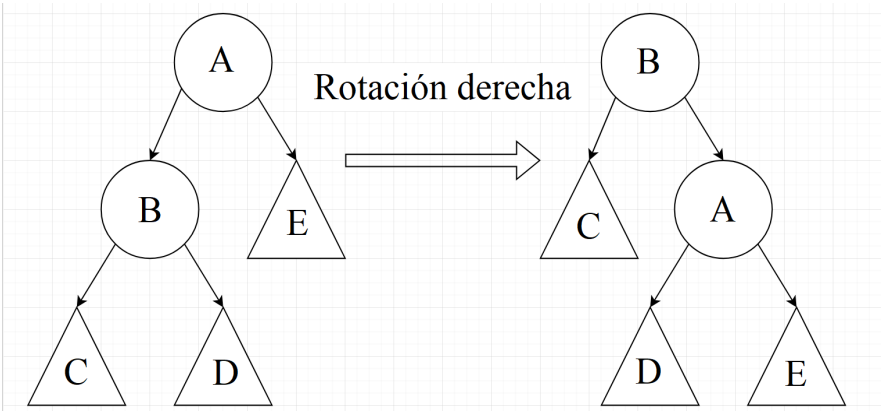

Figura 4.2: Ejemplo de realizar una rotación derecha en el nodo A del árbol.

Algo a mencionar es que la restricción en donde el árbol PFC guarda en sus nodos internos un puntero al PFC de la hoja más a la izquierda de su hijo derecho se sigue manteniendo

después de realizar estas operaciones, pudiéndose ver un ejemplo de estos punteros en la figura 2.10. Esto se debe a que dichas operaciones preservan el orden de los elementos guardados en el árbol.

Una demostración más práctica consistiría primero en notar que en ambas rotaciones los subárboles C, D y E no sufren modificaciones internas, por lo que sus punteros siguen apuntando al PFC correcto. Por lo tanto, basta con mostrar que los nodos A y B apuntan al PFC correcto en ambas operaciones. Para la operación de *rotacion_izquierda*, se puede observar en la figura 4.1 que el PFC más a la izquierda del hijo derecho del nodo A es el PFC más a la izquierda del subárbol D, tanto antes como después de la rotación. Dado que el puntero guardado en A no sufre modificaciones con la rotación, entonces este puntero apunta correctamente. Notar que un razonamiento análogo se puede aplicar para verificar la validez de los punteros en el resto de los nodos involucrados en ambas rotaciones. Por lo tanto, se concluye que estas operaciones pueden ser utilizadas en los nodos del árbol PFC sin afectar el correcto funcionamiento de la estructura.

Para poder mantener el árbol balanceado, cada vez que se quiera insertar o eliminar un elemento en el árbol, se procederá de forma similar a como se realiza en un árbol binario de búsqueda convencional. Específicamente, se efectúa una búsqueda recursiva en el árbol hasta encontrar la posición donde se debe insertar o eliminar el elemento. Luego, se realiza la inserción o eliminación correspondiente.

Posteriormente, se incorpora el comportamiento adicional propio de los árboles AVL: se recorre recursivamente hacia atrás (desde la hoja hacia la raíz), realizando el balanceo de cada nodo en el camino de regreso. Esto se debe a que los únicos nodos que podrían haber quedado desbalanceados son aquellos que se encuentran en el camino seguido durante la búsqueda.

Finalmente, para realizar el balanceo de un nodo, se pueden distinguir cuatro casos a la hora de intentar balancear un nodo v del árbol:

- *Izquierda-Izquierda:* Este caso corresponde cuando $v.left.height - v.right.height > 1$ y $v.left.left.height - v.left.right.height \geq 0$, o equivalentemente, el subárbol izquierdo del hijo izquierdo de v tiene más altura que el subárbol del hijo derecho de v. Para balancear el árbol se realiza $v.rotacion\_derecha$.

- *Izquierda-Derecha:* Este caso corresponde cuando $v.left.height - v.right.height > 1$ y $v.left.left.height - v.left.right.height < 0$, o equivalentemente, el subárbol derecho del hijo izquierdo de v tiene más altura que los demás. Para balancear el árbol se realiza primero $v.left.rotacion\_izquierda$ y luego $v.rotacion\_derecha$.

- *Derecha-Izquierda:* Este caso corresponde cuando $v.left.height - v.right.height < -1$ y $v.right.left.height - v.right.right.height > 0$, o equivalentemente, el subárbol izquierdo del hijo derecho de v tiene más altura que los demás. Para balancear el árbol se realiza primero $v.right.rotacion\_derecha$ y luego $v.rotacion\_izquierda$.

- *Derecha-Derecha:* Este caso corresponde cuando $v.left.height - v.right.height < -1$ y $v.right.left.height - v.right.right.height \leq 0$, o equivalentemente, el subárbol derecho del hijo derecho de v tiene más altura que el subárbol del hijo izquierdo. Para balancear el árbol se realiza $v.rotacion\_izquierda$.

4.4. Ring dinámico mejorado

Al igual que la implementación del *wavelet matrix*, la estructura utilizada en el *ring* dinámico hecho por Yuval Linker realiza una abstracción del funcionamiento del *ring*, permitiendo cambiar fácilmente el *wavelet matrix* utilizado. Por lo tanto, se decide utilizar esta implementación del *ring*, realizando un cambio en el *wavelet matrix*, pasando de uno que utiliza un *bitvector* basado en SPSI, a uno que utiliza como *bitvector* el *adaptive dynamic bitvector*. Un ejemplo de lo descrito anteriormente se puede ver en la figura 4.3.

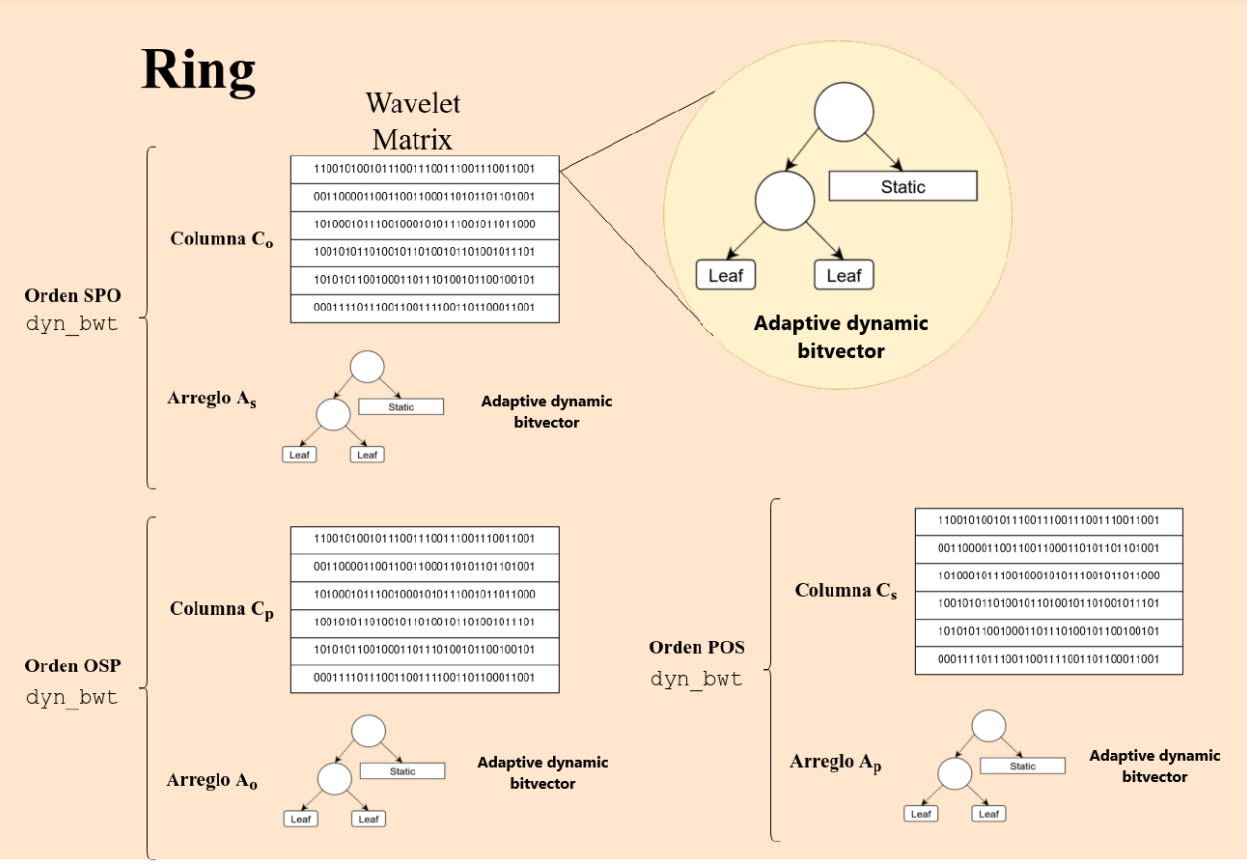

Figura 4.3: Diagrama del *ring* dinámico mejorado con aridad 3.

Notar que esta implementación del *ring* dinámico mejorado está limitada a tener una aridad 3, pues solo se va a utilizar para realizar operaciones sobre la representación del grafo que guarda. En particular, se busca poder realizar intercaladamente cuatro operaciones principales:

- *Consulta:* Corresponde a realizar una consulta BGP sobre la representación del grafo que se guarda en el *ring* dinámico mejorado. Para esto se hace uso del algoritmo Leapfrog Triejoin visto en la sección 2.3, pero usando la estructura del *ring* para recorrer las tripletas en vez de los *tries*.
- *Insertar arista:* Corresponde a agregar una nueva arista a la representación del grafo

que se guarda en el *ring* dinámico mejorado. Notar que si se inserta una arista con un nodo que no existe, se crea un nuevo nodo.

- *Eliminar arista:* Corresponde a eliminar una arista de la representación del grafo que se guarda en el *ring* dinámico mejorado.
- *Eliminar nodo:* Corresponde a eliminar un nodo de la representación del grafo que se guarda en el *ring* dinámico mejorado.

Notar que las últimas tres operaciones son operaciones de actualización, en donde se tienen que realizar modificaciones sobre el *ring* y el diccionario compacto asociado, por lo que, a continuación, se van a describir a grandes rasgos cómo se hace cada una de estas operaciones.

4.4.1. Inserción de arista

Para el caso de la inserción de una arista, lo primero a realizar es la traducción de las palabras introducidas por el usuario (los 3 elementos de la tripleta) a los identificadores usados en el *ring*. Para ello, se utiliza la operación *get_or_insert* del diccionario, buscando el identificador de la palabra si se encuentra y, en caso contrario, creando un nuevo elemento en el diccionario para esa palabra (para más detalles, ver la sección 2.6.3).

Una vez teniendo los identificadores para los 3 elementos que identifican a la arista mediante una tripleta de sujeto, predicado y objeto, se procede a realizar la inserción en el *ring*. Para ello, primero se debe encontrar la ubicación en donde se tiene que insertar la tripleta en cada uno de los 3 órdenes.

Dada una tripleta $\{x_S, x_P, x_O\}$, para encontrar la ubicación en donde se tiene que insertar, se comienza por escoger uno de los tres elementos del triple. A priori da igual cuál se escoja, pero en este caso se decide partir por el elemento del predicado, por lo cual se acota el espacio de inserción a aquellas ubicaciones que tienen en el predicado a x_P. En particular, este intervalo se calcula como $[A_P(x_P),\ A_P(x_P+1)-1]$, siendo $A_P(x_P)$ la posición de la primera tripleta que tiene a x_P como valor al ordenar por el predicado, y $A_P(x_P+1)-1$ la posición del último triple que tiene a x_P como valor al ordenar por el predicado. Notar que si no hay triples en el intervalo, entonces $A_P(x_P) = A_P(x_P+1)$.

Una vez acotado, si es que no existen tripletas en el intervalo, se procede a insertar la tripleta en la ubicación encontrada; por lo tanto, se insertaría x_S en $C_S[A_P(x_P)]$, ya que C_S está ordenado por p, además de incrementar en uno los contadores correspondientes en A_S. Notar que, como se usa un *gap-encoded bitvector*, basta con agregarle un 0 al intervalo que representa la cantidad de x_S. Luego, se iría moviendo a los siguientes órdenes usando la función F definida en la ecuación 2.2, insertando los valores de la tripleta de forma equivalente a como se hizo en el orden POS.

En caso de que existan tripletas en el intervalo, se utiliza nuevamente la función F, pero esta vez para mover cada uno de los límites del intervalo, quedando entonces el nuevo intervalo como:

$$\Big[F_S(A_P(x_P)),\ F_S(A_P(x_P+1)-1)\Big]. \tag{4.1}$$

Una vez acotado por el nuevo intervalo, nuevamente se verifica si hay tripletas en este intervalo; si es que no existen, de forma equivalente a como se hizo anteriormente, se inserta la tripleta, solo que en vez de partir insertando en C_S, se tendría que partir desde C_O. En caso de existir, se acota nuevamente por el intervalo:

$$\Big[F_O(F_S(A_P(x_P))),\ F_O(F_S(A_P(x_P+1)-1))\Big]. \tag{4.2}$$

Una vez acotado por este último intervalo, se verifica por última vez si existen tripletas en el intervalo; si es que no existen, de forma equivalente a las anteriores veces, se inserta la tripleta, solo que en vez de partir insertando en C_S o C_O, se parte insertando desde C_P. En caso de ya existir la tripleta, indicaría que la tripleta que se quiere insertar ya existe en el *ring*, y ya que actualmente no se aceptan elementos repetidos, se omitiría la inserción.

4.4.2. Eliminación de arista

La eliminación de una arista en el *ring* dinámico mejorado se realiza de forma análoga a la inserción, es decir, se escoge un elemento del triple, a partir del cual se empieza a acotar los triples que son candidatos para eliminar hasta encontrar el triple exacto que se desea eliminar. Una diferencia respecto a la inserción es que necesariamente se tienen que acotar por todos los elementos del triple antes de poder eliminar una arista, ya que de lo contrario se podría eliminar una tripleta incorrecta.

Notar que con la implementación actual no deberían existir tripletas repetidas, por lo que si al final de la búsqueda del triple no se encuentra un triple único — es decir, si no se encuentra ningún triple o se encuentra más de uno — no se realizará la eliminación de la arista, aunque es importante mencionar que este comportamiento se podría cambiar manejando cada uno de esos casos por separado.

Una vez realizada la búsqueda del triple y verificado que efectivamente solo hay un triple, se procede a eliminarlo. Por lo tanto, es importante notar que al final de la búsqueda se tendría un rango $[i, i]$, siendo i la posición que tiene el triple bajo uno de los tres órdenes. Supongamos que se partió acotando primero por el predicado (orden POS), entonces al final se estaría acotando por el objeto (orden OSP), por lo cual la posición i sería la posición del triple bajo el orden OSP (i_P). Teniendo esto, se calcula la posición del triple en el orden SPO (i_O) y en el orden POS (i_S).

Pudiendo con esto hacer finalmente la eliminación de los elementos del triple. En particular, se borra el predicado eliminando el i_P-ésimo elemento del arreglo C_P y disminuyendo en uno los contadores correspondientes en A_P. Notar que, como se usa un gap-encoded-bitvector, basta con quitarle un 0 al intervalo que representa la cantidad de x_P, siendo x_P el identificador del predicado que se quiere borrar. De forma equivalente, se hace lo mismo para los otros dos órdenes del *ring*, usando i_O para el orden SPO e i_S para el orden POS.

Después de hacer la eliminación del triple como tal, hay que comprobar si los distintos elementos que se borraron de la tripleta se siguen utilizando. En caso de que no se sigan utilizando, se elimina ese valor del diccionario como se explicó en la sección 2.6.3.

Para comprobar si un elemento se sigue utilizando en el *ring*, existen dos casos distintos. Por un lado, tenemos a los elementos que guardan los nodos (sujeto y objeto), y por otro lado, al elemento que guarda la relación (predicado). Para el caso de los nodos, se tiene que verificar si todavía existe un triple que use ese identificador, ya sea como sujeto o como objeto, pues ambos comparten el mismo diccionario. Para el caso de la relación, basta con verificar si existe algún triple que contenga dicho predicado.

Para verificar si existe algún triple que tenga como sujeto al identificador x_S, se tiene que calcular el valor $A_S[x_S + 1] - A_S[x_S]$; si este valor es 0, significa que no existen triples que usen ese identificador como sujeto. De lo contrario, el identificador sigue siendo utilizado. Notar que algo totalmente equivalente se puede hacer para verificar si un identificador se sigue utilizando como objeto o como predicado.

4.4.3. Eliminación de nodo

Para la eliminación de nodos, se realiza algo similar a la eliminación de aristas, pero en vez de buscar una tripleta en específico, se encuentra un rango de tripletas y se van borrando una a una, de forma equivalente a como se hacía con las aristas individuales.

Entonces, para buscar el rango de tripletas a eliminar, se calculan dos rangos distintos: uno para cuando el nodo a eliminar actúa como objeto y otro para cuando actúa como sujeto. Dado un nodo con identificador x que se desea eliminar, el primer rango se calcula como $[A_O(x), A_O(x+1) - 1]$ y corresponde a las aristas donde x aparece como objeto. El segundo rango se calcula como $[A_S(x), A_S(x + 1) - 1]$ y corresponde a las aristas donde x aparece como sujeto.

Una vez obtenido el conjunto de aristas a eliminar, se procede con su eliminación de forma casi idéntica al caso de una única arista. Sin embargo, una optimización importante es que no es necesario verificar si el nodo debe eliminarse del diccionario en cada arista eliminada, ya que al final del proceso el nodo completo será eliminado. Esto permite ahorrar una verificación por cada arista procesada.

4.5. Código e implementación

En esta sección se describen algunos cambios menores realizados en la implementación del *ring* dinámico, con el fin de corregir errores detectados y añadir nuevas funcionalidades, además de mencionar los elementos necesarios para ejecutar el código.

En primer lugar, se desarrolló un código que permite ejecutar un conjunto de *queries* que incluyen intercaladamente las cuatro operaciones descritas anteriormente. Cabe destacar que anteriormente solo se disponía de implementaciones que permitían ejecutar un conjunto de consultas de un único tipo, o bien ejecutar las operaciones en un orden fijo y controlado.

A partir de este nuevo código, fue posible detectar algunos errores menores que provocaban fallos en el uso de los *ring* dinámicos (tanto en la versión original como en la mejorada). El primer problema encontrado fue que el dataset de Wikidata contenía tripletas repetidas.

Para solucionarlo, se añadió un filtro en el constructor del *ring* que garantiza la ausencia de duplicados. El principal problema de permitir elementos repetidos es que el comportamiento frente a la eliminación de aristas repetidas no estaba correctamente definido (actualmente, tales eliminaciones se omiten).

Posteriormente, se detectó un error que ocurría al intentar eliminar una arista que no existía en el *ring*. En ciertos casos, esto podía resultar en la eliminación incorrecta de otra arista. El problema se originaba cuando, al acotar por un solo elemento del triple, el intervalo resultante contenía una única tripleta. Este caso era erróneamente interpretado como una coincidencia exacta, cuando en realidad no se había verificado la presencia de todos los elementos del triple. Es importante notar que este error no ocurre cuando se eliminan tripletas cuya existencia en el *ring* está previamente asegurada.

Otro error encontrado se producía al realizar la eliminación de un nodo. En particular, cuando se eliminaba una tripleta asociada al nodo, no se realizaba correctamente la verificación para determinar si ese nodo seguía siendo utilizado. Esta verificación es clave para decidir si un elemento debe ser eliminado del diccionario. Como consecuencia, se producían casos en que existían nodos con identificadores que ya no estaban presentes en el diccionario. Este error se debía a que solo se verificaba si existían triples con ese identificador como sujeto o como objeto, cuando se deberían haber verificado ambos. Cabe señalar que se trató de un error de tipeo, ya que en la memoria de Yuval Linker la verificación correcta está descrita adecuadamente.

Finalmente, fue necesario modificar la estructura *EmptyOrPFC*, que originalmente se creaba a partir de un *UNION*, para reemplazarla por una estructura que incluye un *bool* adicional para distinguir los casos. Esta modificación se debió a un error que provocaba que, al eliminar una palabra del PFC, su identificador no se eliminara correctamente, lo que podía provocar sobrescrituras incorrectas al realizar *split* (operación del diccionario). Si bien este cambio soluciona el problema y permite un comportamiento correcto, implica un costo adicional en espacio, ya que ahora cada elemento del arreglo ocupa más memoria.

En cuanto a los detalles técnicos, el código fue escrito en *C++17* para poder hacer uso de *std::variant*, que permite manejar una variable que alterna entre distintos tipos de clase de manera más segura que un *UNION*. No obstante, se utilizan también librerías escritas en *C++11*, por lo que pueden generarse advertencias (*warnings*) dependiendo del compilador y configuración utilizados.

El código del *ring* dinámico mejorado se encuentra disponible en: https://github.com/overexpOG/Ring-amo, el cual es un *fork* del *ring* dinámico original implementado por Yuval Linker: https://github.com/yuval-linker/Ring.

Para su correcto funcionamiento, es necesario instalar previamente:

- *sdsl-lite*, utilizado para las estructuras compactas estáticas.
- El submódulo *DYNAMIC*, que es un *fork* del repositorio *DYNAMIC* de Yuval Linker, a su vez basado en el repositorio original de Nicola Prezza *DYNAMIC*. Este módulo es necesario para manejar estructuras dinámicas, en particular, la *wavelet matrix*.

Capítulo 5

Validación

En esta sección se presentan los resultados obtenidos a partir de los distintos experimentos diseñados durante el desarrollo de esta memoria. En primer lugar, se llevaron a cabo experimentos para determinar el valor más adecuado del hiperparámetro θ en el *ring* dinámico mejorado. A diferencia de otros hiperparámetros, el valor óptimo de θ puede variar considerablemente dependiendo del contexto de uso [7].

Posteriormente, se realizaron experimentos destinados a analizar los efectos que tiene la proporción entre consultas y actualizaciones en el comportamiento general del *ring* dinámico mejorado. Finalmente, se llevaron a cabo experimentos comparativos con el objetivo de evaluar el rendimiento del *ring* dinámico mejorado frente a otras implementaciones del *ring*.

Todos estos experimentos se realizaron sobre un subgrafo de Wikidata [31], específicamente sobre el subgrafo que contiene aproximadamente mil millones de tripletas, utilizando tanto el subgrafo como el conjunto de consultas proporcionados por https://zenodo.org/records/4035223. De este modo, se busca evaluar el comportamiento de la estructura en un entorno que simula un escenario real con una gran cantidad de datos.

Es importante señalar que las consultas extraídas se clasifican en 17 tipos distintos, según el patrón que presentan en el grafo, lo que puede impactar directamente en el rendimiento de la estructura. Los nombres y patrones correspondientes a cada tipo de consulta pueden verse en la figura 5.1.

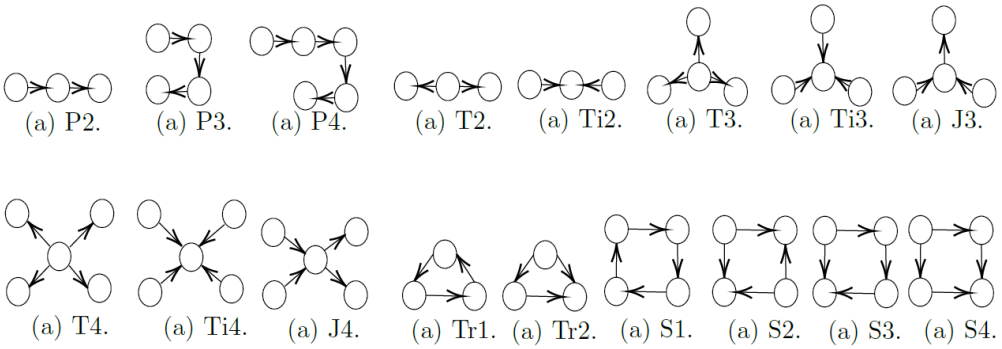

Figura 5.1: Tipos de consultas usadas para probar el subgrafo de la wikidata, extraido de [6].

Para la realización de estos experimentos, se utilizó un servidor con procesador Intel(R) Xeon(R) E5-2609 a 2.4 GHz, con 4 núcleos, 10 MB de caché y 128 GB de RAM. El uso de un servidor se debe a la gran cantidad de memoria RAM requerida para los experimentos con el *ring* dinámico mejorado, necesitando alrededor de 30–35 GB de RAM para las consultas y aproximadamente 50–60 GB de RAM para la construcción de la estructura. Este consumo se debe principalmente a usar un subgrafo con mil millones de tripletas.

Cabe notar que si los mismos experimentos se ejecutaran con el subgrafo de 80 millones de tripletas, se podría utilizar un computador con 8 GB de RAM, aunque solo se probo su funcionamiento utilizando un computador de 16 GB.

Cabe mencionar que las mejoras introducidas en el *ring* dinámico mejorado fueron dos: por un lado, la implementación del *ring* utilizando el adaptive dynamic bitvector, y por otro lado, la mejora del diccionario compacto mediante árboles AVL para balancear el árbol PFC. Por lo tanto, se pueden distinguir cuatro variaciones del *ring* dinámico, que son las siguientes:

- *RDMAVL*: *ring* dinámico mejorado con árboles AVL en el diccionario.
- *RDMAB*: *ring* dinámico mejorado con árboles binarios en el diccionario.
- *RDAVL*: *ring* dinámico con árboles AVL en el diccionario.
- *RDAB*: *ring* dinámico con árboles binarios en el diccionario.

Siguiendo esta misma lógica para los *ring* estáticos, se pueden definir también cuatro variaciones:

- *RAB*: *ring* estático con árboles binarios en el diccionario.
- *RAVL*: *ring* estático con árboles AVL en el diccionario.
- *CRAB*: c-ring con árboles binarios en el diccionario.
- *CRAVL*: c-ring con árboles AVL en el diccionario.

Aun que en las pruebas solo se llego a utilizar el *RAB* y el *RAVL*.

5.1. Datasets de queries

Para la construcción de los conjuntos de queries sobre los cuales se realizarán los experimentos, es importante notar que, además de las consultas extraídas del subgrafo de Wikidata, se utilizarán también queries que realicen actualizaciones sobre el *ring* dinámico mejorado. En particular, estas actualizaciones corresponden a la inserción de aristas, eliminación de aristas y eliminación de nodos, extrayendo gran parte de estas queries de actualización de las que fueron usadas para probar el *ring* dinámico [6]. Sin embargo, aquellas queries que correspondían a la inserción de aristas fueron ampliadas agregando variaciones al final, con el fin

de incluir inserciones que intentaran agregar aristas con nodos que no existían previamente, permitiendo así también la creación de nodos.

Un aspecto relevante a considerar al realizar experimentos sobre el *ring* dinámico mejorado es que el adaptive dynamic bitvector utilizado para su construcción modifica su estructura interna al recibir actualizaciones, o cuando tras una actualización se reciben una cantidad determinada de consultas. En particular, las operaciones que se realizan y el orden en que se ejecutan afectan los tiempos de respuesta de dichas operaciones, influyendo a su vez en los tiempos generales del *ring* dinámico mejorado. Por lo tanto, para medir un comportamiento realista de esta estructura, no es suficiente probar por separado las consultas y las actualizaciones, sino que es necesario evaluar el funcionamiento del *ring* dinámico mejorado utilizando todos los tipos de queries de manera simultánea, ordenando dichas queries de forma aleatoria, con el fin de emular un comportamiento lo más parecido posible al de consultas reales.

Notar que si se introdujeran todas las queries de forma simultánea y azarosa, se estaría utilizando una proporción fija de consultas por actualización, lo cual afecta el desempeño del adaptive dynamic bitvector y, por ende, del *ring* dinámico mejorado. Por esta razón, se decidió construir cuatro conjuntos de datos de queries, denominados *q-1000*, *q-100*, *q-10* y *q-1*, cada uno con una proporción distinta de consultas por actualización. En particular, *q-1000* contiene mil consultas por cada actualización, *q-100* cien consultas por actualización, *q-10* diez consultas por actualización, y finalmente *q-1* una consulta por actualización. Esto permite evaluar de manera adecuada el comportamiento del *ring* dinámico mejorado bajo distintas cargas de trabajo.

Es importante mencionar que, para el caso de Wikidata, se espera una proporción más cercana a la del conjunto *q-1000* que a los demás datasets. Por lo tanto, se considera al conjunto *q-1000* como el más relevante para el problema actual, utilizando los demás conjuntos principalmente para entender el comportamiento general del *ring* dinámico mejorado.

Para la construcción de cada conjunto de datos, se empleó una cantidad aproximada de 3000 queries. La cifra no es exacta debido a que se priorizó mantener una proporción precisa de consultas por actualización, lo que permitió que el total de queries excediera ligeramente las 3000 para alcanzar dicha proporción. Cabe destacar que, dado que solo se cuenta con 850 consultas diferentes, se permitió que estas se repitieran dentro de los datasets. Esto no representa un problema, pues las consultas se ejecutan completas cada vez que se invocan. Por el contrario, las queries de actualización, cuando se llaman consecutivamente, su segunda ejecución se realiza de forma incompleta. Por lo cual, se agrego una restricción para impedir que pudieran haber queries de actualización repetidas en los datasets.

Finalmente, cabe señalar que existe una desproporción entre la cantidad de eliminaciones (sumando tanto las de aristas como las de nodos) y la cantidad de inserciones tras el aumento. Por esta razón, se impuso la restricción de que cada dataset contenga la misma cantidad de queries de inserción y eliminación. Además, las eliminaciones se distribuyeron con una proporción aproximada de cuatro eliminaciones de arista por cada eliminación de nodo, aunque esta proporción no es estricta y su distribución es aleatoria. Cabe mencionar que, dado que el dataset *q-1000* solo contiene tres actualizaciones, se le agregaron una inserción de arista, una eliminación de arista y una eliminación de nodo para comprobar su comportamiento bajo las distintas operaciones de actualización.

5.2. Experimentos para encontrar el valor más adecuado de θ en el ring dinámico mejorado

En esta sección se presentan diversos experimentos con el objetivo de determinar el valor más adecuado de θ para utilizar en el *ring* dinámico mejorado. En particular, se evaluarán valores de θ cercanos a los recomendados para el uso del adaptive dynamic bitvector [7].

5.2.1. Efectos de θ en las queries de selección

Al comparar los tiempos promedio que se utilizan para resolver una consulta con distintos valores de θ, como se observa en la figura 5.2, se nota que al disminuir el valor de θ hasta 0.01, los tiempos promedio se mantienen o disminuyen. Sin embargo, al reducir θ más allá de 0.01, se observa un aumento continuo en los tiempos promedio, salvo en el dataset q-100, donde se registra una disminución en $\theta = 0.00001$.

A pesar de esta excepción, en todos los datasets utilizados el valor de θ que obtiene los mejores tiempos promedio sigue siendo 0.01.

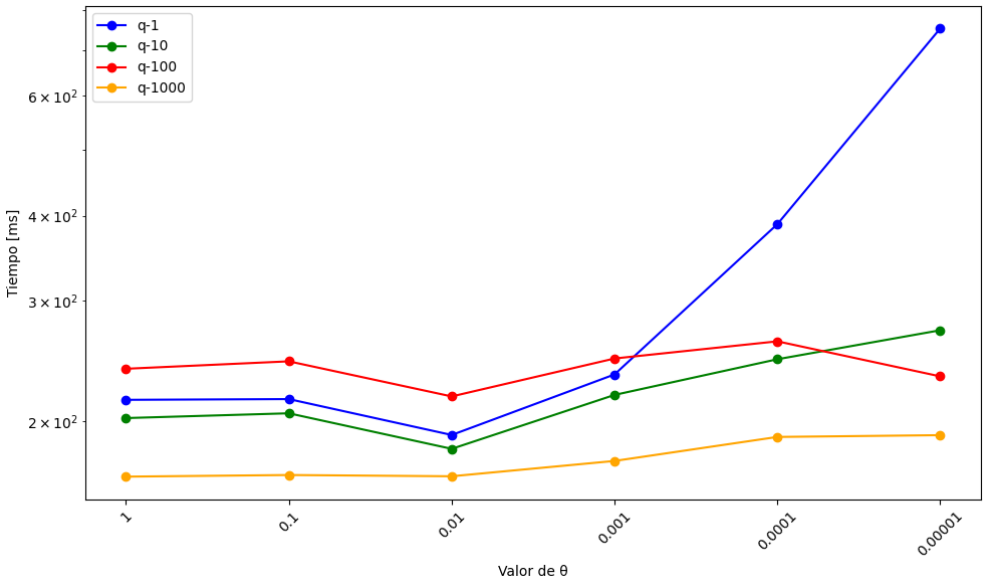

Figura 5.2: Tiempo promedio en milisegundos que tarda en realizar una consulta con distintos valores de θ usando los 4 datasets en el RDMAVL, con el tiempo en escala logarítmica.

5.2.2. Efectos de θ en las queries de actualización

Al comparar los tiempos promedio para realizar la eliminación de una arista al variar el valor de θ, como se muestra en la figura 5.3, se observa un comportamiento similar al de las queries de selección. Los menores tiempos promedio de eliminación se obtienen con $\theta = 0.01$. Sin embargo, a diferencia de las consultas, en todos los datasets se registra una disminución

en los tiempos promedio al usar $\theta = 0.00001$, y en aquellos datasets con una menor proporción de consultas por actualización, incluso se observa una ligera disminución en $\theta = 0.0001$.

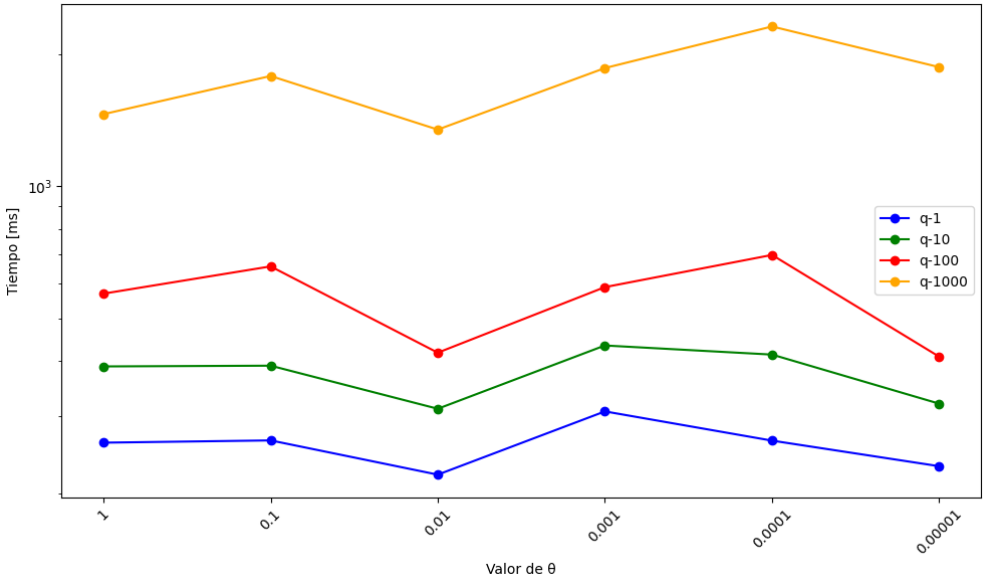

Figura 5.3: Tiempo promedio en milisegundos para realizar la eliminación de una arista con distintos valores de θ, usando los 4 datasets en el RDMAVL. El tiempo está en escala logarítmica.

Cabe destacar que un comportamiento muy similar se observa al comparar los tiempos promedio para la inserción de una arista al variar θ, cuyo gráfico se encuentra en el anexo A.

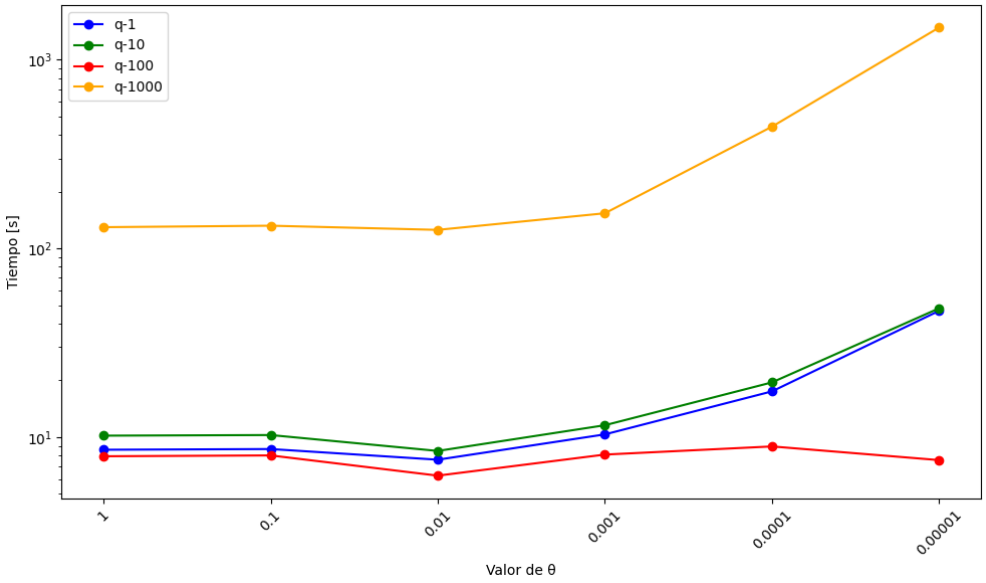

Figura 5.4: Tiempo promedio en segundos para realizar la eliminación de un nodo con distintos valores de θ, usando los 4 datasets en el RDMAVL. El tiempo está en escala logarítmica.

Al comparar los tiempos promedio para realizar la eliminación de un nodo al variar el valor de θ, como se muestra en la figura 5.4, se observa que a partir de $\theta = 0.001$, al disminuir θ los tiempos promedio aumentan de forma exponencial, excepto en el caso de q-100. Cabe destacar que la escala del tiempo está en formato logarítmico para facilitar la visualización del crecimiento exponencial.

5.2.3. Efectos de θ en el tiempo total para procesar todas las queries

Al comparar el tiempo total utilizado para procesar todas las queries al variar θ, como se muestra en la figura 5.5, se observa un comportamiento similar al presentado en las queries de consulta: una disminución del tiempo en $\theta = 0.01$, seguida de un aumento al disminuir aún más el valor de θ. Obteniendo el menor tiempo total para procesar todas las queries con $\theta = 0.01$.

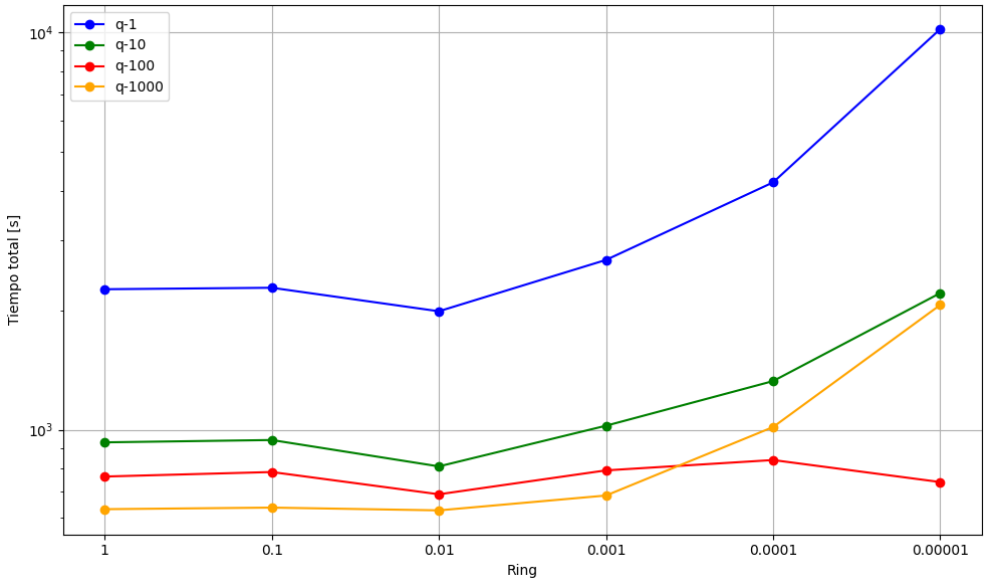

Figura 5.5: Tiempo total en segundos para procesar todas las queries de un dataset usando RDMAVL, variando el valor de θ. El tiempo está en escala logarítmica.

5.2.4. Elección de θ

A partir de los resultados obtenidos al variar θ en el rango de valores $[1, 0.00001]$, se concluye que el valor más adecuado para usar en el *ring* dinámico mejorado es $\theta = 0.01$. Esto se debe a que dicho valor logra consistentemente los mejores resultados en todos los experimentos realizados.

Cabe destacar que este valor coincide con la recomendación para el *adaptive dynamic bitvector*, considerando su tamaño y la proporción de consultas por actualización [7].

5.3. Experimentos para averiguar los efectos del valor q en el ring dinámico mejorado

Como se mencionó anteriormente, la cantidad de consultas que se realizan por cada actualización (denotada como q) afecta los tiempos de las operaciones en el *adaptive dynamic bitvector*. Por ello, se decide estudiar los efectos que tiene el valor de q sobre la estructura del *ring* dinámico mejorado.

Es importante aclarar que el q del *ring* dinámico mejorado no es idéntico al q del *adaptive dynamic bitvector*, aunque sí están directamente relacionados. Esto se debe a que una consulta en el *ring* dinámico mejorado implica cientos o miles de consultas en los *adaptive dynamic bitvectors* que se usan internamente. De manera similar, una actualización en el *ring* dinámico mejorado implica realizar actualizaciones en sus *adaptive dynamic bitvectors* (y también consultas, aunque en menor medida). Por lo tanto, aunque no sean exactamente lo mismo, el valor de q en el RDMAVL afecta directamente al q de los *adaptive dynamic bitvectors* usados en él.

5.3.1. Efectos del valor q en las queries de selección

Al analizar la distribución de los tiempos de consulta en el *ring* dinámico mejorado según el valor de q, como se muestra en la figura 5.6, se observa que dicha distribución tiende a disminuir a medida que aumenta q. Esto es esperable, ya que los tiempos de las operaciones en el *adaptive dynamic bitvector* disminuyen conforme q crece, y como se mencionó anteriormente, estos valores están directamente relacionados.

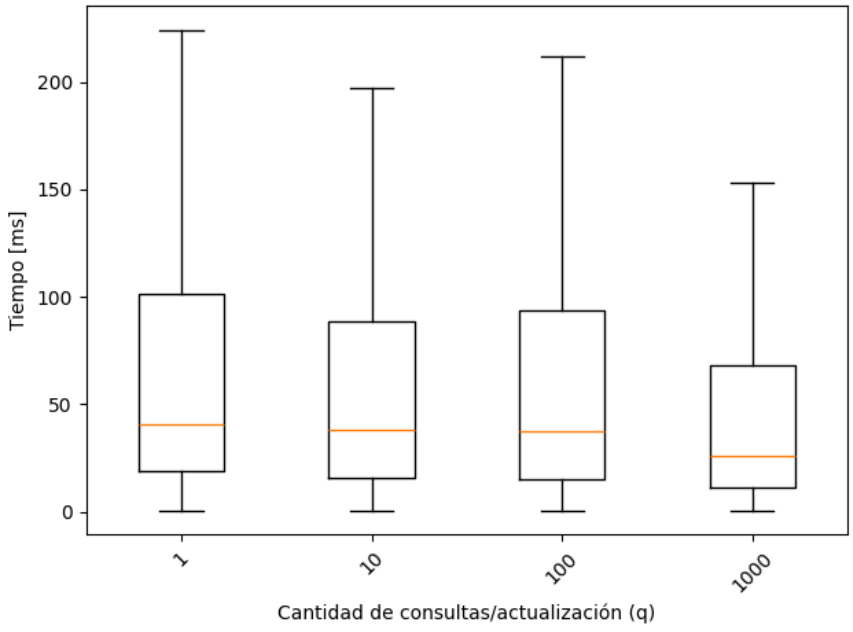

Figura 5.6: Distribución de la duración en milisegundos de una consulta en RDMAVL variando la proporción consulta/actualización (q).

Algo a notar es que para el dataset de queries q-100 se observa un pequeño aumento en los tiempos, cuando se esperaría una disminución. Una posible explicación es que los datasets presentan una distribución distinta en los tipos de consultas, siendo que cada tipo puede tener un tiempo esperado diferente.

Al analizar los tiempos promedio por tipo de consulta para cada dataset, mostrados en la figura 5.7, se puede ver que en la mayoría de los tipos (9 de 17), el dataset q-100 tiene un tiempo promedio menor que q-1 y q-10. Solo en 2 de 17 tipos, q-100 muestra un tiempo promedio mayor que el resto de los datasets. Por lo tanto, es probable que el ligero aumento observado para q-100 en la figura 5.6 se deba a que posee un mayor porcentaje de tipos de consulta con tiempos promedio más altos.

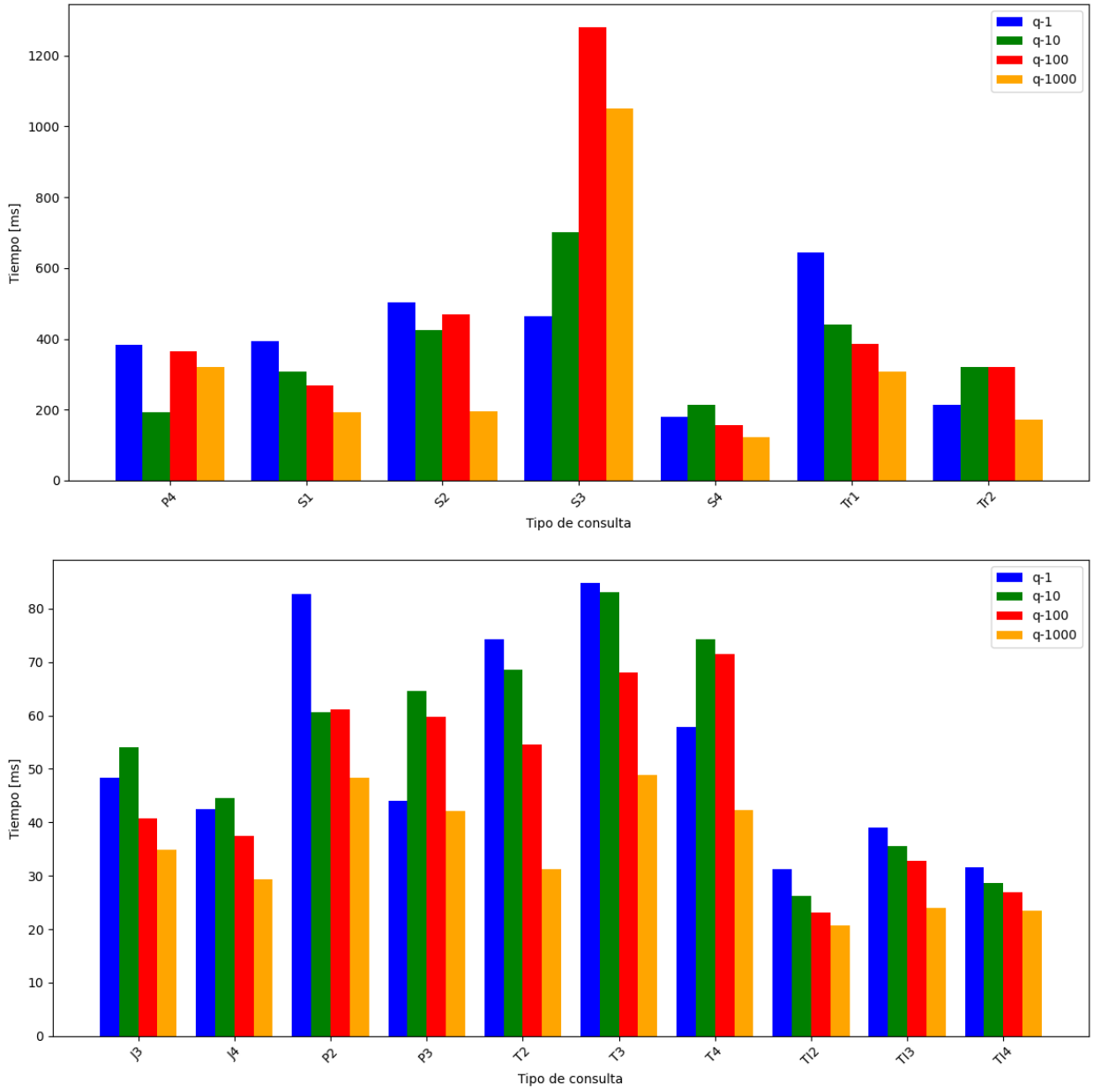

Figura 5.7: Tiempo promedio en milisegundos que toma realizar cada tipo de consulta en los distintos datasets en el RDMAVL.

Además de lo mencionado anteriormente, en la figura 5.7 se puede observar un comportamiento similar al descrito para la figura 5.6. En particular, se nota que en general los tiempos promedio que toman los distintos tipos de consulta tienden a disminuir al usar datasets con una mayor proporción de consultas por actualización. Una de las excepciones a este comportamiento es el tipo de consulta S3, donde se observa un comportamiento casi opuesto al esperado.

5.3.2. Efectos del valor q en las queries de actualización

En cuanto al efecto que tiene la proporción de consultas por actualización en las queries de actualización, se puede observar en la tabla 5.1 que, a diferencia de las consultas, las queries de actualización presentan un mayor tiempo promedio por operación al aumentar el valor de q. Esto es especialmente notable en el caso del dataset q-1000, donde el tiempo promedio es aproximadamente un orden de magnitud mayor.

dataset	Insertar arista [ms]	Eliminar nodo [ms]	Eliminar arista [ms]
q-1	199.793	7561.431	220.294
q-10	242.008	8315.974	306.047
q-100	440.592	6126.725	411.537
q-1000	2234.535	125526.078	1366.890

Tabla 5.1: Tiempos promedio en milisegundos que toma realizar cada operación de actualización usando distintos datasets en el RDMAVL.

5.3.3. Efectos del valor q en la memoria RAM utilizada durante las queries

Finalmente, además de los tiempos promedio que toma cada operación, en la figura 5.8 se observa que el espacio utilizado en memoria RAM por el RDMAVL disminuye a medida que aumenta el valor de q. Registrando diferencias cercanas al 30 % entre la proporción de consultas por actualización que utiliza más memoria RAM y aquella que utiliza menos. Este aumento en el uso de memoria para valores menores de q se atribuye a una mayor frecuencia de operaciones de *flattening* y *splitting* en el adaptive dynamic bitvector.

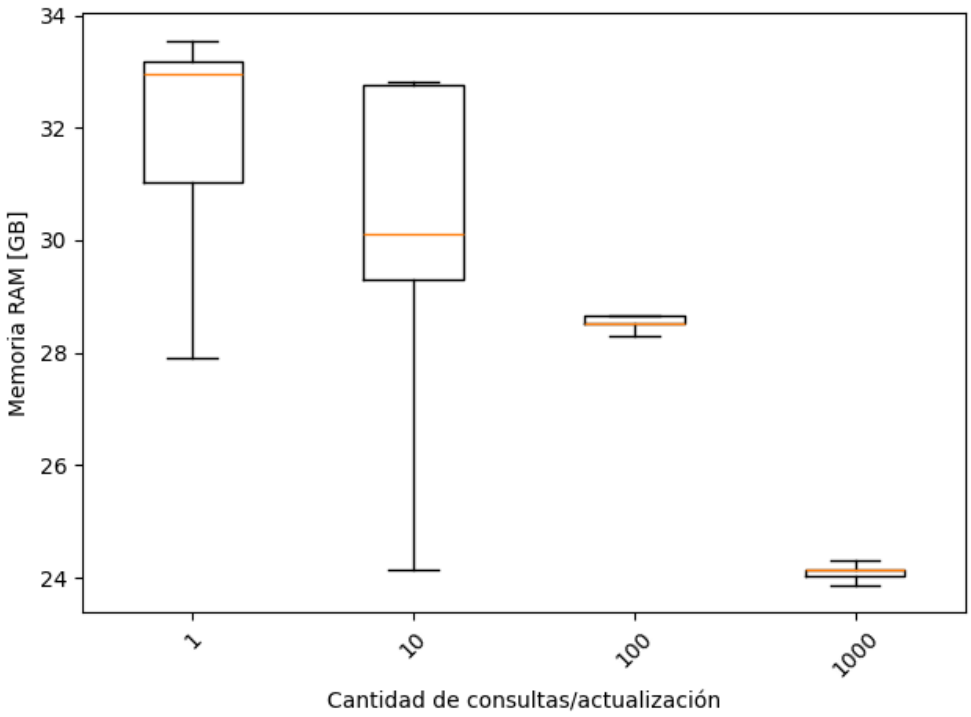

Figura 5.8: Distribución de la memoria RAM en GB utilizada al realizar las queries variando la proporción q en el RDMAVL.

5.4. Experimentos comparativos entre rings

Finalmente, en esta sección se compararán los resultados obtenidos por el *ring* dinámico mejorado con respecto al resto de las implementaciones del *ring*, analizando en particular los tiempos de ejecución y el espacio utilizado por cada tipo de *ring*.

5.4.1. Comparación del tiempo usado para realizar queries de consulta

Para las queries de consulta, se realizará la comparación utilizando tanto los *rings* estáticos como los dinámicos, dado que ambos tipos de *rings* pueden procesar este tipo de consultas. Es importante señalar que los *rings* estáticos no soportan queries de actualización, por lo que se generaron cuatro nuevos datasets que corresponden a subconjuntos de los originales, conservando únicamente las queries de consulta.

Estos nuevos datasets, denominados qs-1000, qs-100, qs-10 y qs-1, serán utilizados en los *rings* estáticos en lugar de los datasets originales q-1000, q-100, q-10 y q-1. Al contener las mismas queries de consulta, incluso en el mismo orden, se espera que, para tiempos similares en los *rings*, el desempeño al responder estas queries sea comparable. Por lo tanto, cuando se haga referencia a pruebas con algún dataset q-x en un *ring* estático, en realidad se estará utilizando el dataset qs-x.

Al comparar los tiempos que toma cada *ring* para ejecutar las queries de consulta, como se muestra en la figura 5.9, se observa que el *ring* dinámico mejorado logra reducir aproximadamente a la mitad los tiempos respecto al *ring* dinámico tradicional, aunque aún requiere alrededor de cuatro veces más tiempo que el *ring* estático. Cabe destacar que, en la mayoría de los casos, el RDMAVL obtiene resultados iguales o peores al RDMAB.

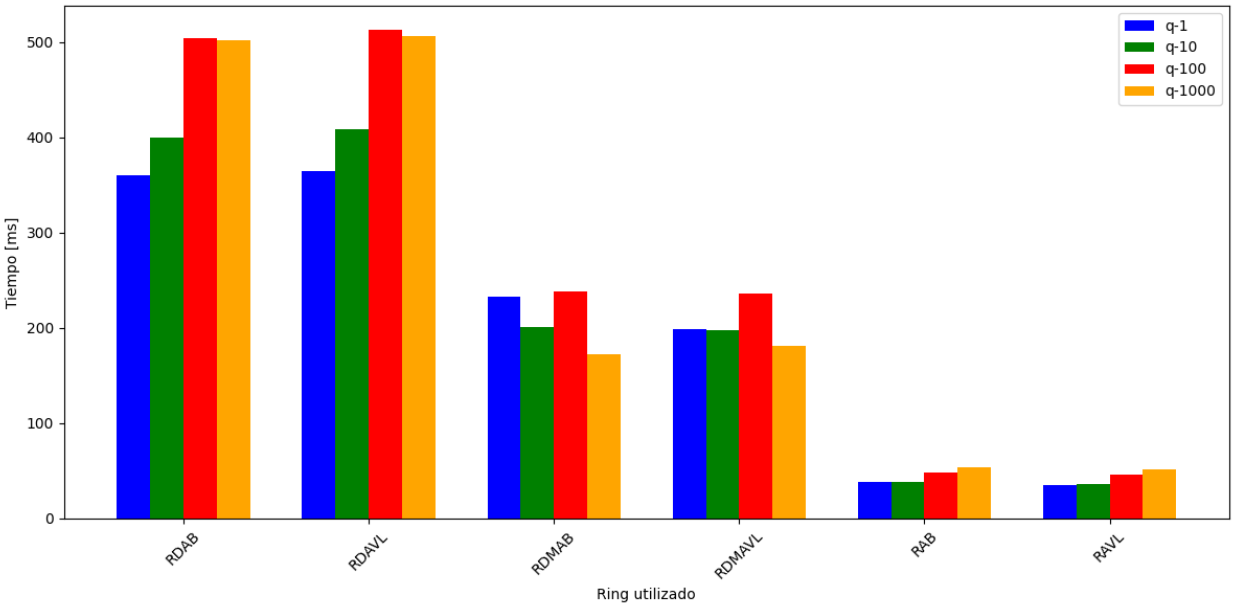

Figura 5.9: Tiempo promedio en milisegundos que toma realizar una consulta en los distintos tipos de *rings* usando los 4 datasets.

Al analizar la comparación de los tiempos para los distintos tipos de consulta utilizando el dataset q-1000, como se muestra en la figura 5.10, se observan resultados consistentes con lo mencionado anteriormente. En general, el *ring* dinámico presenta tiempos entre 2 y 3 veces mayores que el *ring* dinámico mejorado, mientras que este último, a su vez, tiene tiempos entre 2 y 3 veces mayores que el *ring* estático, aunque existen algunas excepciones donde la diferencia es aún mayor.

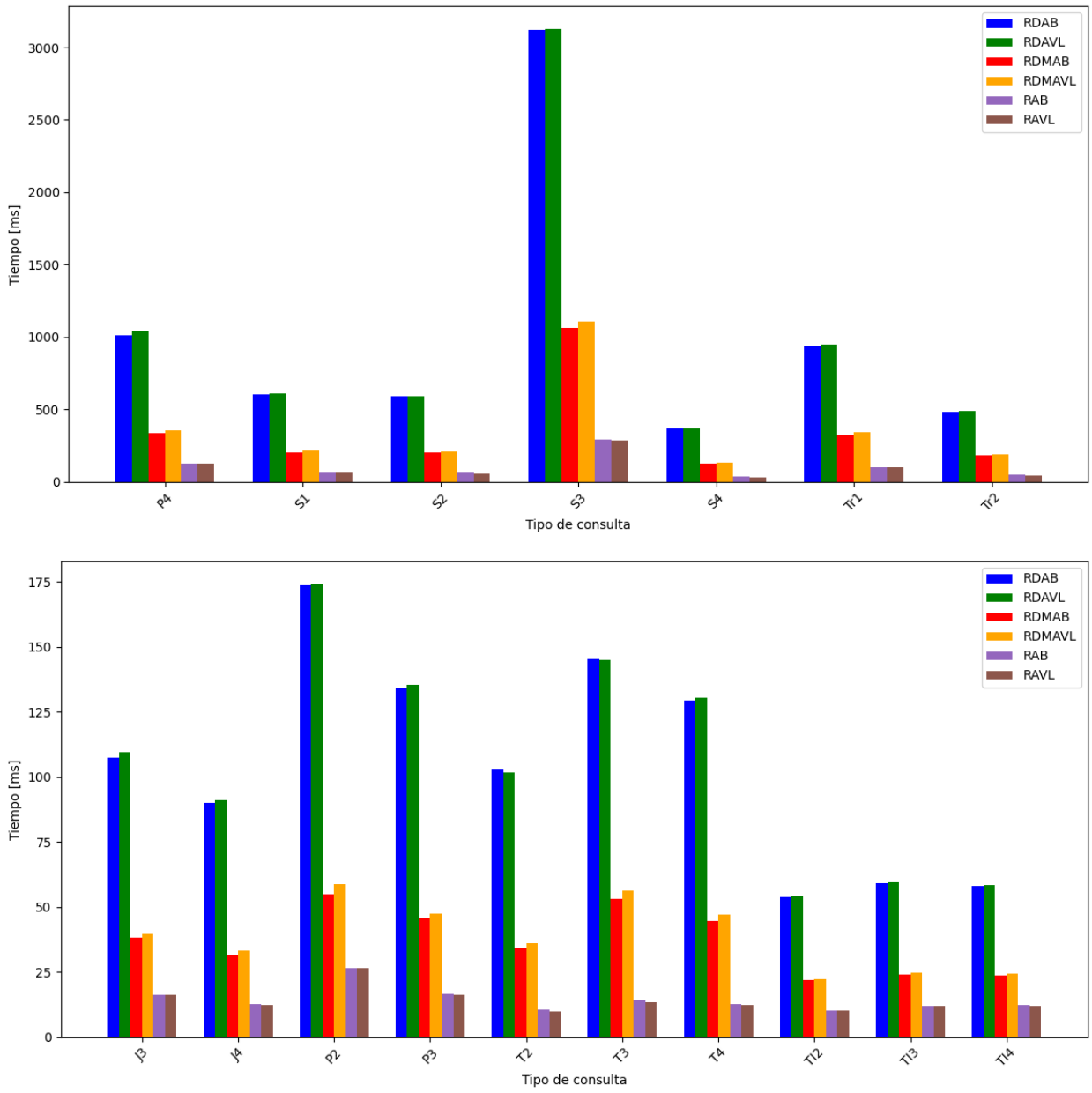

Figura 5.10: Tiempo promedio en milisegundos que toma realizar cada tipo de consulta en los distintos *rings* usando el dataset q-1000.

5.4.2. Comparación del tiempo usado para realizar queries de actualización

Para las queries de actualización, se compararán únicamente los *rings* dinámicos, ya que los *rings* estáticos no permiten este tipo de operación.

Al comparar los tiempos promedio que toma cada *ring* dinámico en realizar la inserción de una arista (figura 5.11), se puede observar que el *ring* dinámico mejorado presenta un tiempo aproximadamente tres órdenes de magnitud mayor al del *ring* dinámico tradicional. Es decir, mientras que los *rings* dinámicos realizan esta operación en menos de un milisegundo, el

ring dinámico mejorado requiere tiempos cercanos al segundo. También se observa que, en la mayoría de los casos, el RDMAVL obtiene resultados similares o incluso mejores que el RDMAB (a excepción de q-1000).

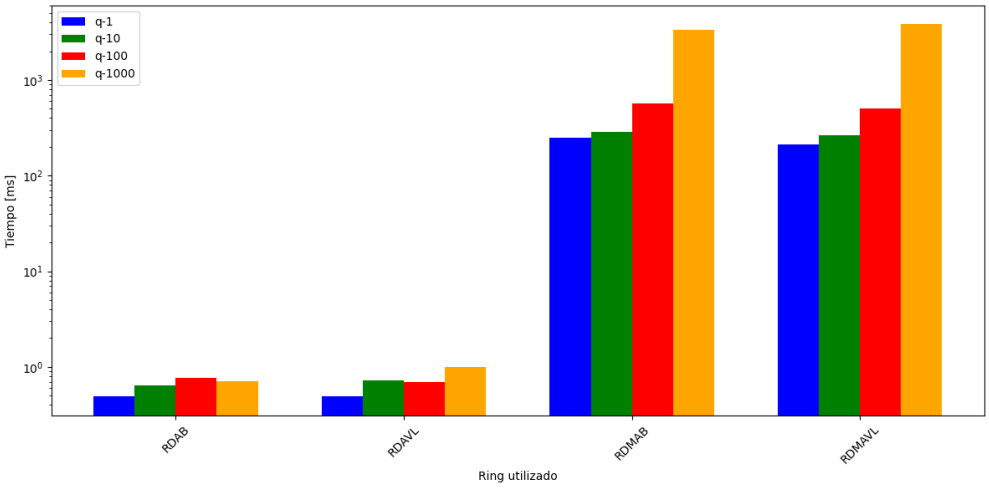

Figura 5.11: Tiempo promedio en milisegundos que toma realizar una inserción de arista en los distintos tipos de *rings* usando los 4 datasets, con el tiempo en escala logarítmica.

Un comportamiento similar se observa al analizar los tiempos promedio que toma cada *ring* dinámico al realizar tanto la eliminación de una arista como la eliminación de un nodo. En particular, en la eliminación de una arista se evidencia un aumento casi equivalente de tres órdenes de magnitud, mientras que en la eliminación de un nodo el aumento es menor, cercano a un orden de magnitud. Las figuras correspondientes a estos experimentos se encuentran en el anexo B.

5.4.3. Comparación del tiempo total usado para procesar todas las queries

Para la comparación del tiempo total utilizado en el procesamiento de todas las queries, se considerarán únicamente los *rings* dinámicos, ya que los *rings* estáticos no ejecutan el conjunto completo de queries, ya que no pueden hacer queries de actualización. El objetivo principal es observar el comportamiento al ejecutar todos los tipos de queries en conjunto.

Al analizar la figura 5.12, se observa que en aquellos datasets con una mayor proporción de consultas por actualización, el *ring* dinámico mejorado obtiene tiempos totales menores en comparación con el *ring* dinámico tradicional. En particular, mientras mayor sea la proporción de consultas respecto a las actualizaciones, mayor es la mejora observada en los tiempos.

Sin embargo, en el dataset que tiene igual proporción de consultas y actualizaciones (q-1), el tiempo requerido para procesar todas las queries se duplica en el *ring* dinámico mejorado, evidenciando su sensibilidad al valor de q. También se puede notar que, en la mayoría de los casos, el RDMAVL presenta resultados similares o incluso mejores que el RDMAB.

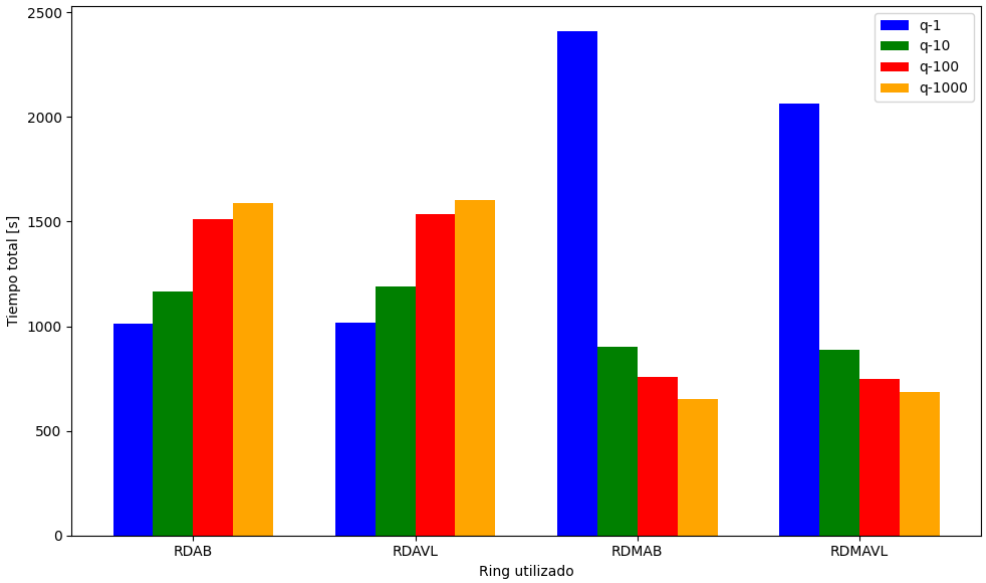

Figura 5.12: Tiempo total, en segundos, que tarda en procesarse la totalidad de las queries de un dataset utilizando los distintos *rings* dinámicos.

5.4.4. Comparación de la memoria RAM utilizada durante las queries

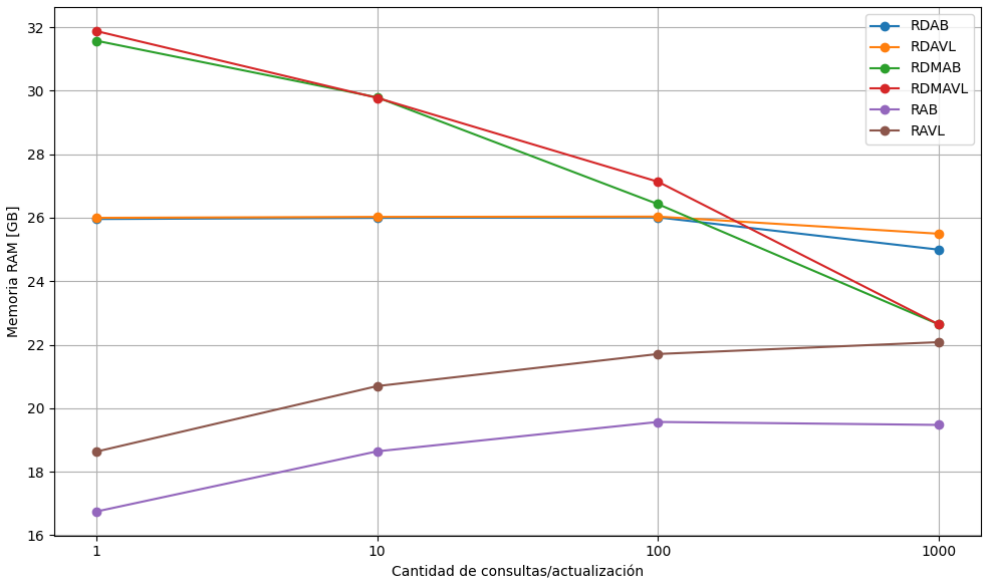

Figura 5.13: Memoria RAM promedio, en GB, utilizada durante la ejecución de cada dataset de queries usando los distintos *rings*.

Al comparar la memoria RAM utilizada durante la ejecución de las queries con distintos *rings*, como se muestra en la figura 5.13, se observa que los *rings* dinámicos consumen más memoria RAM que los *rings* estáticos, siendo el *ring* dinámico mejorado el que presenta

un mayor consumo cuando la proporción de consultas por actualización es baja. Sin embargo, a medida que esta proporción aumenta, la cantidad de memoria RAM utilizada por el *ring* dinámico mejorado se vuelve menor que la usada por el *ring* dinámico, aproximándose considerablemente a la del *ring* estático.

5.4.5. Comparación del espacio utilizado al guardarse

Finalmente, al comparar el espacio promedio utilizado para guardar un triple en cada *ring*, como se muestra en la figura 5.14, se observa que el *ring* dinámico mejorado es el que ocupa menos espacio al guardarse, seguido por el *ring* dinámico, y luego por el *ring* estático. También se puede apreciar que aquellos *rings* que utilizan el diccionario con árbol AVL emplean un espacio ligeramente mayor que los que utilizan árboles binarios normales, aunque esta diferencia es casi imperceptible.

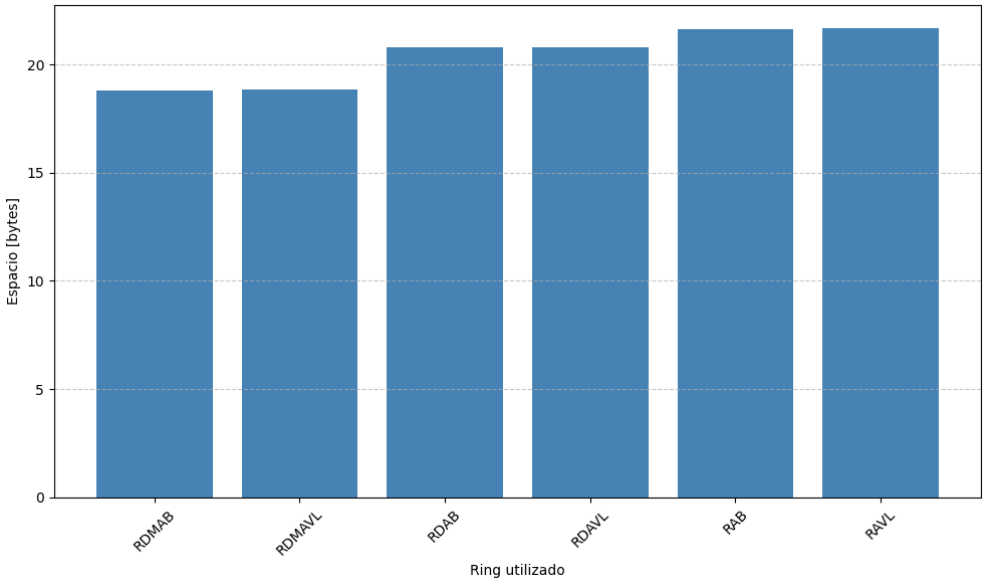

Figura 5.14: Espacio promedio, en bytes, utilizado para guardar un triple al variar el tipo de *ring*.

Es importante mencionar que, al momento de guardar el *ring* dinámico mejorado, se realiza sucesivamente la operación de *flattening* con el fin de almacenarlo de forma más comprimida. Esta operación genera un aumento considerable en los tiempos de guardado del *ring* dinámico mejorado en comparación con el resto. Cabe destacar que, durante el proceso de guardado, se siguen respetando las restricciones previamente explicadas del *ring* dinámico mejorado. No obstante, si se ignorara el valor de ϵ al guardar, se podrían obtener mejores tiempos en las operaciones de consulta de forma temporal al cargar el *ring* dinámico mejorado.

Capítulo 6

Conclusión

El trabajo desarrollado en esta memoria consistió en modificar la implementación del *ring* dinámico, reemplazando el bitvector utilizado con el objetivo de reducir los tiempos de ejecución en las queries, acercándose así al rendimiento del *ring* estático. Para ello, fue necesario traducir la implementación del *adaptive dynamic bitvector* al lenguaje C++, y posteriormente adaptar dicho bitvector a la estructura del *ring* dinámico. Adicionalmente, se modificó el diccionario compacto para garantizar que el árbol utilizado se mantuviera siempre balanceado.

Como resultado, se logró implementar una versión mejorada del *ring* dinámico, en la que, manteniendo la capacidad de actualización propia del diseño dinámico, se reducen los tiempos de ejecución en escenarios donde predominan las queries de consulta por sobre las de actualización, lo cual corresponde al caso de uso esperado en conjuntos de datos como Wikidata. No obstante, en contextos donde la proporción entre consultas y actualizaciones es más equilibrada, se observa un deterioro en los tiempos de respuesta.

Estos resultados se explican principalmente por dos factores. En primer lugar, el *ring* dinámico mejorado logra, de manera consistente, mejores tiempos en la ejecución de consultas, independientemente de la proporción entre consultas y actualizaciones. No obstante, este rendimiento se ve contrarrestado por un aumento considerable en los tiempos de las operaciones de actualización. De esta forma, en escenarios donde predominan las consultas, el beneficio en tiempo supera el costo de las actualizaciones; sin embargo, cuando la proporción entre ambos tipos de operaciones es similar, el impacto negativo de las actualizaciones tiende a dominar.

Respecto al uso de memoria, el *ring* dinámico mejorado presenta un consumo comparable al de las versiones anteriores. En particular, el uso de memoria RAM varía de forma significativa según la proporción de consultas por actualización: se alcanzan valores cercanos a los del *ring* estático cuando predominan las consultas, pero se duplica aproximadamente en escenarios con proporciones equilibradas. A pesar de ello, el espacio requerido para almacenar el *ring* dinámico mejorado resulta ligeramente inferior al del resto de implementaciones, manteniendo así su carácter de estructura altamente compacta.

Finalmente, respecto a la mejora implementada en el diccionario —que consistió en re-

emplazar el árbol binario del árbol PFC por un árbol AVL, con el objetivo de mantenerlo balanceado y reducir así los tiempos de consulta—, si bien no se logró obtener una mejora en el rendimiento, sí se consiguió mantener la estructura balanceada. Esto hace que el diccionario sea más robusto frente a situaciones adversas, como la inserción consecutiva de elementos en orden lexicográfico.

El desarrollo de esta memoria requirió una profundización en el estudio de estructuras de datos compactas y sus algoritmos, con el fin de comprenderlas y aplicarlas en el contexto de una estructura compleja como el *ring*. A lo largo del trabajo se aprendió a utilizar y modificar librerías existentes, muchas de ellas desarrolladas en el marco de investigaciones científicas, como es el caso del *ring*. Para ello fue necesario adquirir un dominio detallado de C++17, permitiendo implementar y adaptar soluciones complejas de forma eficiente. Además, se aplicaron conocimientos en diseño experimental y análisis de resultados, lo que permitió consolidar aprendizajes teóricos previos en un entorno aplicado.

Por lo tanto, se puede concluir que el objetivo general de esta memoria fue cumplido. Se logró mantener tanto la competitividad en el uso de espacio como las capacidades de inserción y eliminación del *ring* dinámico, mejorando al mismo tiempo los tiempos de procesamiento en los casos de uso esperados. Estas mejoras, sin embargo, dependen del contexto: en escenarios con predominancia de consultas —como es el caso esperado de uso de la Wikidata— el *ring* dinámico mejorado muestra un rendimiento superior al *ring* dinámico original. No obstante, en contextos con una alta proporción de actualizaciones, el rendimiento puede verse afectado. Cabe destacar que, en los escenarios donde sí se observa una mejora, los tiempos de consulta del *ring* dinámico mejorado se sitúan entre los del *ring* dinámico y el *ring* estático, cuando se esperaba acercarse más al rendimiento de este último.

A partir de los resultados obtenidos, surgen diversas líneas de trabajo futuro. Una de ellas consiste en optimizar la operación *select* del *adaptive dynamic bitvector*, con el objetivo de mejorar su eficiencia. Una posible estrategia sería implementar la versión propuesta por Pandey et al., que utiliza una instrucción a nivel de hardware capaz de acelerar dicha operación entre 2 y 4 veces; aunque esto requeriría una CPU que soporte dicha instrucción [30]. Otra posible mejora sería incorporar un intérprete SPARQL, otorgando así al *ring* capacidades comparables a sistemas de bases de datos como Jena o Virtuoso.

En conclusión, el *ring* dinámico mejorado representa una extensión significativa de los casos de uso del *ring*, al ofrecer, bajo determinados contextos, mejoras sustanciales en los tiempos de procesamiento de consultas. Esto permite considerar su uso en escenarios donde predominan las consultas por sobre las actualizaciones, mientras que el *ring* dinámico original seguiría siendo más adecuado en entornos con un mayor volumen de operaciones de actualización.

Bibliografía

[1] D. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra, "Join processing for graph patterns: An old dog with new tricks," in *Proceedings of the GRADES'15*, GRADES'15, (New York, NY, USA), Association for Computing Machinery, 2015.

[2] A. Atserias, M. Grohe, and D. Marx, "Size bounds and query plans for relational joins," *SIAM Journal on Computing*, vol. 42, no. 4, pp. 1737–1767, 2013.

[3] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, "Worst-case optimal join algorithms," *J. ACM*, vol. 65, mar 2018.

[4] D. Arroyuelo, A. Gómez-Brandón, A. Hogan, G. Navarro, J. L. Reutter, J. Rojas-Ledesma, and A. Soto, "The Ring: Worst-case optimal joins in graph databases using (almost) no extra space," *ACM Transactions on Database Systems*, vol. 29, no. 2, p. article 5, 2024.

[5] T. L. Veldhuizen, "Leapfrog triejoin: a worst-case optimal join algorithm," 2013.

[6] Y. Linker, "Implementación dinámica del ring," tesis de grado en ingeniería, Universidad de chile, Santiago, Chile, 2023. https://users.dcc.uchile.cl/~gnavarro/algoritmos/mem/tesisYuval.pdf.

[7] G. Navarro, "Adaptive dynamic bitvectors," in *Proc. 31st International Symposium on String Processing and Information Retrieval (SPIRE)*, 2024. To appear.

[8] A. Bigerl, F. Conrads, C. Behning, M. A. Sherif, M. Saleem, and A.-C. Ngonga Ngomo, "Tentris – A Tensor-Based Triple Store," in *The Semantic Web – ISWC 2020*, pp. 56–73, Springer International Publishing, 2020.

[9] D. Arroyuelo, G. Navarro, J. L. Reutter, and J. Rojas-Ledesma, "Optimal joins using compressed quadtrees," *ACM Trans. Database Syst.*, vol. 47, May 2022.

[10] A. Hogan, C. Riveros, C. Rojas, and A. Soto, "A worst-case optimal join algorithm for sparql," in *The Semantic Web – ISWC 2019* (C. Ghidini, O. Hartig, M. Maleshkova, V. Svátek, I. Cruz, A. Hogan, J. Song, M. Lefrançois, and F. Gandon, eds.), (Cham), pp. 258–275, Springer International Publishing, 2019.

[11] G. Navarro, *Bitvectors*, p. 64–102. Cambridge University Press, 2016.

[12] G. J. Jacobson, *Succinct static data structures*. PhD thesis, Carnegie Mellon University, USA, 1988. AAI8918056.

[13] S. Denver, "Jacobson's rank," 2023. From Denver Smith's blog.

[14] B. Langmead, "Jacobson's rank," 2020. Teaching material of langmead-lab.

[15] R. Patro, "Bitvector rank & select: Primitives of succinct data structures," 2019. Presentation used in the course 'Algorithms, Data Structures and Inference for High Throughput Genomics' at the University of Maryland.

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

[17] R. González and V. Mäkinen, "Practical implementation of rank and select queries," 2005.

[18] B. Langmead, "Clark's select," 2021. Teaching material of langmead-lab.

[19] N. Prezza, "A framework of dynamic data structures for string processing," in *16th International Symposium on Experimental Algorithms (SEA 2017)*, vol. 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 11:1–11:15, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017.

[20] M. Fredman and M. Saks, "The cell probe complexity of dynamic data structures," in *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, STOC '89, (New York, NY, USA), p. 345–354, Association for Computing Machinery, 1989.

[21] G. Navarro, "Wavelet trees for all," *Journal of Discrete Algorithms*, vol. 25, pp. 2–20, 2014. 23rd Annual Symposium on Combinatorial Pattern Matching.

[22] F. Claude, G. Navarro, and A. Ordóñez, "The wavelet matrix: An efficient wavelet tree for large alphabets," *Information Systems*, vol. 47, pp. 15–32, 2015.

[23] F. Claude and G. Navarro, "The wavelet matrix," in *Proceedings of the 19th International Conference on String Processing and Information Retrieval*, SPIRE'12, (Berlin, Heidelberg), p. 167–179, Springer-Verlag, 2012.

[24] T. Gagie, G. Navarro, and S. J. Puglisi, "New algorithms on wavelet trees and applications to information retrieval," *Theoretical Computer Science*, vol. 426-427, pp. 25–41, 2012.

[25] M. A. Martínez-Prieto, N. Brisaboa, R. Cánovas, F. Claude, and G. Navarro, "Practical compressed string dictionaries," *Information Systems*, vol. 56, pp. 73–108, 2016.

[26] H. Williams and J. Zobel, "Compressing integers for fast file access," *The Computer Journal*, vol. 42, 08 2002.

[27] A. A. and, "Maintaining α-balanced trees by partial rebuilding," *International Journal of Computer Mathematics*, vol. 38, no. 1-2, pp. 37–48, 1991.

[28] J. Nievergelt and E. M. Reingold, "Binary search trees of bounded balance," *SIAM Journal on Computing*, vol. 2, no. 1, pp. 33–43, 1973.

[29] V. Mäkinen and G. Navarro, "Dynamic entropy-compressed sequences and full-text indexes," *ACM Trans. Algorithms*, vol. 4, July 2008.

[30] P. Pandey, M. A. Bender, and R. Johnson, "A fast x86 implementation of select," *CoRR*, vol. abs/1706.00990, 2017.

[31] D. Vrandečić and M. Krötzsch, "Wikidata: a free collaborative knowledgebase," *Commun. ACM*, vol. 57, no. 10, pp. 78–85, 2014.

Anexo A

Efectos de theta en las queries de actualización

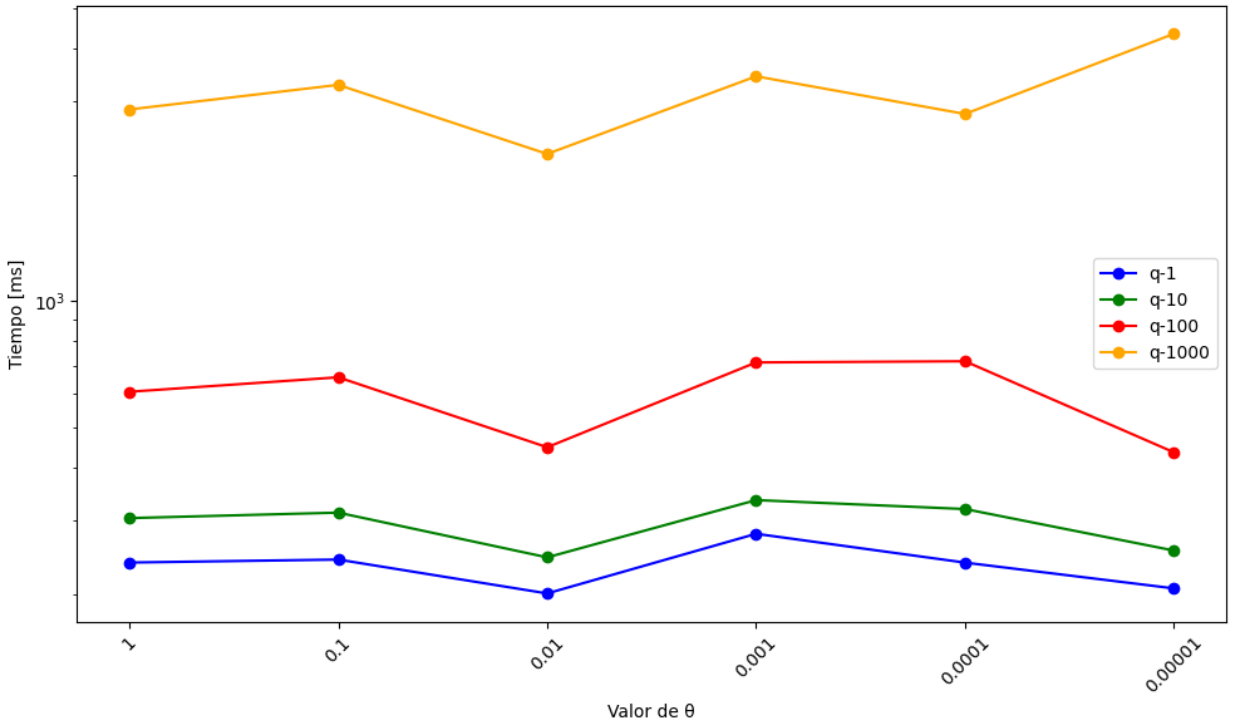

Figura A.1: Tiempo promedio en milisegundos que tarda en realizar la inserción de una arista con distintos valores de θ usando los 4 datasets en el RDMAVL, con el tiempo en escala logaritmica.

Anexo B

Comparación del tiempo usado para realizar queries de actualización

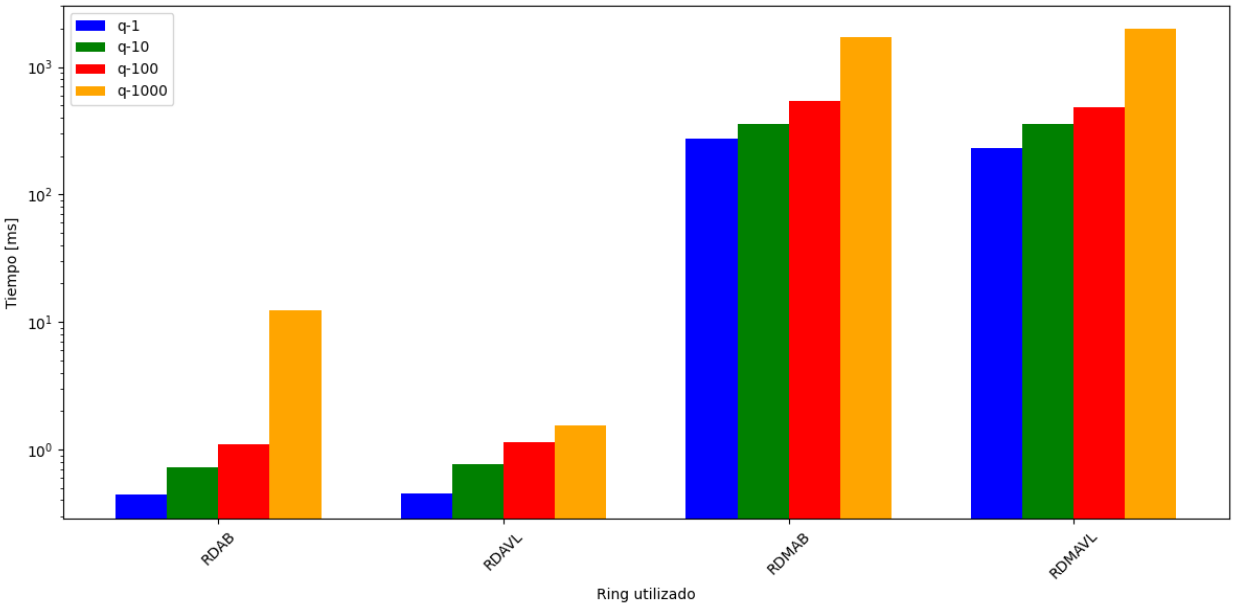

Figura B.1: Tiempo promedio en milisegundos que tarda en realizar la eliminación de una arista con distintos rings usando los 4 datasets, con el tiempo en escala logaritmica.

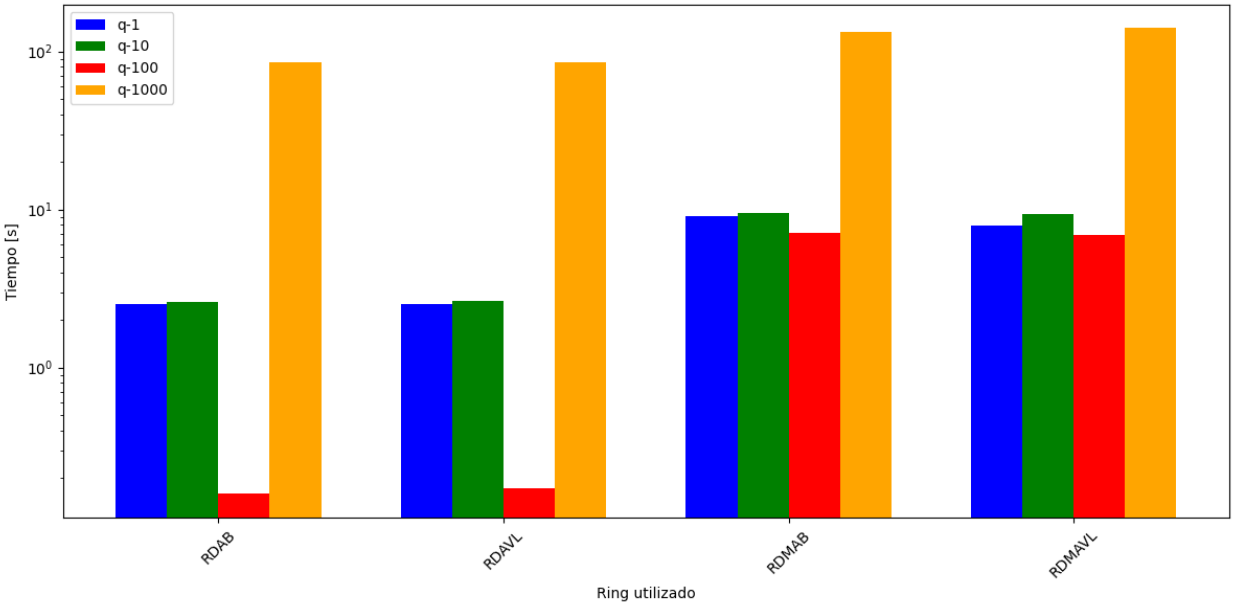

Figura B.2: Tiempo promedio en milisegundos que tarda en realizar la eliminación de un nodo con distintos rings usando los 4 datasets, con el tiempo en escala logaritmica.