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

IMPLEMENTACIÓN DINÁMICA DEL RING

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

YUVAL ARIE LINKER GROISMAN

PROFESOR GUÍA:
GONZALO NAVARRO BADINO

MIEMBROS DE LA COMISIÓN:
JUAN ÁLVAREZ RUBIO
CLAUDIO GUTIÉRREZ GALLARDO

SANTIAGO DE CHILE
2023

Resumen

Las bases de datos de grafos permiten almacenar información utilizando representaciones de nodos y aristas. En este contexto, es posible realizar consultas utilizando *basic graph patterns*, equivalentes en términos generales a *joins* naturales. No obstante, esta consulta requiere ser resuelta en tiempos accesibles. Considerando esto, el Ring es una base de datos compacta que logra tiempos bajos para consultas *join* implementando una variante del algoritmo Leapfrog Triejoin, la que cumple con ser Worst Case Optimal.

Sin embargo, las estructuras sobre las que se basa el Ring no soportan operaciones de actualización, impidiendo la modificación del grafo. Adicionalmente, debido a que los grafos ocupan valores en los nodos y etiquetas, representados como strings, en este índice se debe traducir previamente el alfabeto del grafo al universo de identificadores utilizado por la base de datos. Esto excluye al Ring de varios casos de uso.

Por este motivo, esta memoria plantea y valida el diseño de un nuevo Ring dinámico, el cual permite inserciones y eliminaciones de elementos. Para hacer esto posible, se modificaron y adaptaron estructuras existentes como B-Trees para formar *bitvectors* y *wavelet matrices*, con el fin de tener un formato dinámico y que soporte las operaciones necesarias para el funcionamiento del Ring. Además, se creó un diccionario compacto a base de Plain Front Coding y árboles binarios, cuyo propósito es servir de traductor entre el alfabeto del usuario y los identificadores del índice. La solución fue implementada en C++ y logró ser funcional, siendo validada con el dataset de Wikidata Graph Pattern Benchmark frente a distintos índices del estado del arte, para el cual se añadieron experimentos de actualización.

El Ring dinámico, incluyendo su diccionario, obtuvo buenos resultados en cuanto a compresión, logrando reducir en un 79 % el espacio ocupado frente a los datos en texto plano. En cuanto a los tiempos de consultas, el índice creado fue mucho más lento que su versión estática, sin embargo, resultó ser superior que algunos sistemas, en particular Blazegraph, motor que soporta Wikidata en la actualidad. Adicionalmente, los resultados obtenidos son, en términos generales, comparables con el resto de bases de datos, a excepción del caso particular en el que los patrones presentan ciclos. Para las operaciones de actualización, el Ring dinámico obtuvo resultados mejores que las bases de datos alternativas, siendo únicamente superado por Virtuoso para el caso específico del borrado de nodos.

Se concluye que la solución planteada permite lograr dinamismo, preservando un espacio ocupado cercano al que necesita el Ring estático y siendo superado en tiempo únicamente por índices que requieren de mucho más espacio para funcionar. Incluso se obtienen mejores resultados que algunas implementaciones populares para bases de datos para grafos.

Por los perritos y la música.

Agradecimientos

En primer lugar, agradecer a mi papá y mamá que siempre me han apoyado en todo lo que he hecho, especialmente en mi educación. Me criaron y educaron para siempre salir adelante y, si no era por eso, no terminaba mi memoria. También, agradecer a mis hermanos que me inculcaron sus gustos lo que me trajo hasta donde estoy.

Agradecer a Gonzalo, mi profe guía, que me dio la oportunidad de trabajar en algo tan desafiante como lo es esta memoria. Siempre estuvo atento a mis preguntas y me respondió rápidamente a mis correos lo que es sumamente apreciado. Claramente sin él no hubiera podido realizar este trabajo.

Agradecer a mis panas de la universidad que me acompañaron en toda esta etapa y que sin ellos no habría sobrevivido. En particular darle las gracias al Xavi y al Raúl que me *carrearon* la carrera, siendo siempre juntos en los grupos para los proyectos y estudiando juntos para los controles. Que bueno que estuvieron ahí porque sino, no hubiera tenido vida universitaria. Una mención especial para lxs cabrxs de la sección 1 que hicieron Plan Común más entretenido y al grupo Michil que fuimos el mejor grupo de proyecto de software que ha pasado por el DCC.

También, agradecer a mis amigos de fuera de la universidad que me distraían de los estudios. Destacar al Ilan, Goldstein y Pupkin con los que siempre intentamos hacer cosas y nunca nos salían, pero las risas no faltaban. Fueron vitales los cabros del discord con sus sesiones largas de juego y charlas a las 2AM (y otras cosas que no puedo nombrar). Por último, agradecer a mis amigos músicos con los que he tenido varias aventuras tocando y yendo a conciertos.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

Las bases de datos para grafos permiten almacenar información representada con nodos y aristas. Bajo este esquema, la consulta de datos se lleva a cabo utilizando Basic Graph Patterns (BGP) definido como un conjunto finito de triples sujeto-predicado-objeto que definen un patrón. El triple se conforma por constantes y variables. Evaluar un BGP sobre un grafo es equivalente a realizar un *join* natural, con cada triple representando una *query* atómica, aunque los BGPs además incluyen selecciones simples donde se iguala un atributo a cierta constante.

En consecuencia, una consulta ampliamente utilizada en la representación de estructuras de grafos es el *join* [33]. Este tipo de consulta permite combinar relaciones diversas en un único resultado. Sin embargo, debido a la voluminosa cantidad de información que se requiere procesar, esta operación se vuelve costosa en términos de tiempo, lo que resalta la importancia de optimizarla. Un desafío clásico asociado a esta problemática radica en el hecho de que un algoritmo, para llevar a cabo un *join* natural, puede llegar a procesar más información de la necesaria, lo que impacta negativamente en su eficiencia.

Cuando se cumple el caso anterior, se puede afirmar que un algoritmo no es óptimo en el peor caso, o bien, no cumple con el criterio de optimalidad en el peor escenario (conocido como Worst Case Optimal, WCO por sus siglas en inglés) [6, 32]. Por ejemplo, el acercamiento tradicional a los *joins*, procesando de a dos relaciones a la vez, no cumple con ser óptimo para el peor caso. Formalmente, los algoritmos WCO para *joins* naturales son capaces de procesar estas consultas en un tiempo proporcional al límite AGM [6]. Este límite se define como el tamaño máximo que puede alcanzar el resultado de un join en una base de datos relacional de determinada magnitud. Entonces, a pesar de que un algoritmo WCO puede procesar más información de la necesaria para un input específico, se garantiza que existe alguna entrada donde habría tenido que procesar esa cantidad de información. Un ejemplo de un algoritmo WCO es el Leapfrog Triejoin (LTJ), el cual adopta un enfoque iterativo eliminando atributos en lugar de relaciones [38].

Una limitación común de las implementaciones de algoritmos Worst Case Optimal radica en su tendencia a requerir una cantidad significativa de espacio, debido a la necesidad de indexar múltiples órdenes de las relaciones almacenadas.

Por esta causa, Navarro et al. propusieron una estructura compacta denominada Ring, la cual implementa una variación del algoritmo Leapfrog Triejoin [4]. La principal ventaja de esta estructura reside en su capacidad para utilizar un mínimo espacio adicional en comparación con el requerido para almacenar las relaciones. De esta manera, es posible representar una gran cantidad de datos mientras se realizan consultas en un escenario Worst Case Optimal. En experimentos comparativos realizados sobre grafos RDF, esta estructura demostró obtener resultados superiores, tanto en términos de espacio utilizado como de tiempo requerido, en comparación con otras alternativas similares [4].

Aún cuando el Ring demuestra ser altamente eficiente en términos de tiempo y espacio, la única implementación existente es estática. Esto implica que cualquier actualización requiere reconstruir la estructura desde cero, lo que limita su capacidad para admitir inserciones y eliminaciones de tuplas en las relaciones. Además, un grafo tiene strings para representar los valores de sus nodos y etiquetas, pero el Ring utiliza enteros, suponiendo que se realizó externamente un mapeo para convertir los strings en identificadores numéricos.

En numerosos casos, es necesario trabajar con datos actualizados, es decir, se requiere un enfoque dinámico en la estructura. La implementación de inserciones y eliminaciones acercaría al Ring a más casos de uso y, en consecuencia, la transformaría en una estructura más completa. Sumado a esto, se espera que la ventaja en espacio que proporciona almacenar solamente un índice con los datos se traduzca en una ventaja en tiempos de actualización.

Asimismo, permitir consultas utilizando los nombres originales de nodos y aristas, en vez de identificadores almacenados por el Ring, facilitan la usabilidad de la estructura para los usuarios.

Por estas razones, el presente trabajo propone un nueva versión del Ring que modifica las estructuras subyacentes con el fin de permitir tanto inserciones como eliminaciones de nodos y aristas. Adicionalmente, se introduce una nueva estructura compacta y dinámica para la traducción de un alfabeto del usuario a IDs utilizados por el Ring, y viceversa.

1.1 Objetivos

1.1.1 Objetivo General

El objetivo general del trabajo es modificar el Ring permitiendo inserciones y eliminaciones de tuplas. Se busca que lo anterior se consiga de modo que la estructura siga siendo competitiva en cuanto a espacio ocupado y tiempo de consultas, con respecto a soluciones similares pertenecientes al estado del arte.

1.1.2 Objetivos Específicos

1. Diseñar e implementar estructuras compactas que permitan dinamismo ocupando espacio competitivo con alternativas similares del estado del arte.
2. Implementar inserción y eliminación de aristas en tiempo competitivo con otras alter-

nativas similares del estado del arte.

3. Implementar inserción y eliminación de nodos en tiempo competitivo con otras alternativas similares del estado del arte.

4. Diseñar e implementar un diccionario compacto y dinámico capaz de transformar el alfabeto del grafo a identificadores utilizados por el Ring.

5. Diseñar experimentos para obtener datos precisos que permitan comparar, en tiempo y espacio, distintos motores dinámicos para bases de datos orientadas a grafos.

Capítulo 2

Estado del Arte

En el siguiente apartado, se abordarán conceptos y estructuras de datos fundamentales para comprender el contenido de esta memoria. En primer lugar, se presentará una breve introducción al marco teórico de los joins, así como una explicación concisa del algoritmo de Leapfrog Triejoin. A continuación, se describirán las estructuras de datos comprimidas que conforman los distintos esquemas abordados y modificados en este estudio. Se procederá a explicar en detalle la arquitectura del Ring estático, para finalizar con la presentación de alternativas a las estructuras modificadas en este trabajo.

2.1 Algoritmos Worst Case Optimal

Los algoritmos Worst Case Optimal logran resolver una consulta join en tiempo proporcional al límite AGM [6], es decir, en $O(Q^*)$ con Q^* la cota AGM de la consulta. Este límite se refiere al tamaño máximo que puede tomar la respuesta de la *query join*. Por ejemplo, consideremos el join natural

$$Q(a, b, c) := R(a, b) \bowtie S(b, c) \bowtie T(c, a)$$

En este caso, el límite AGM se establece como $\sqrt{|R| \cdot |S| \cdot |T|}$. Un acercamiento tradicional al procesamiento de joins sería una estrategia *pair-wise join*, donde se computan resultados intermedios. Para ilustrar este punto, si cada relación tuviera el mismo tamaño, es decir $|R| = |S| = |T| = n$, se comenzaría realizando $R(a, b) \bowtie S(b, c)$. No obstante, este cálculo intermedio puede tener un tamaño máximo de n^2, mayor al límite AGM de $n^{\frac{3}{2}}$. Por lo tanto, esta estrategia no es WCO.

Si bien existen múltiples algoritmos para join que cumplen con ser WCO [32], el Ring utiliza una versión adaptada de Leapfrog Triejoin para procesar las consultas.

2.2 Leapfrog Triejoin

El algoritmo Leapfrog-Triejoin, abreviado como LTJ, aborda la resolución de consultas de joins utilizando un enfoque distinto al convencional. En lugar de iterar a través de las relaciones, LTJ procesa la consulta iterando a través de los atributos. Para ello, instancia una variable a la vez, desplazándose a todas las relaciones que tienen dicha variable.

Aunque LTJ es eficiente en términos de tiempo, para que el algoritmo funcione de manera óptima, se requiere indexar las relaciones a través de una cantidad factorial de *tries*. La causa de esto es que cada *trie* indexado almacena una copia de la relación sobre un orden específico para poder leerlo rápidamente. En otras palabras, si una relación tiene d atributos y n filas, entonces se deben crear $d!$ *tries*, cada uno almacenando n strings. Para ejemplificar, una relación `sujeto-objeto-predicado` necesitaría de 6 *tries*.

Leapfrog Triejoin es *worst case optimal* hasta un factor logarítmico, lo que significa que tiene un costo de $O(Q^* \log(n))$ con Q^* el límite AGM para la consulta y n la cardinalidad más grande entre las relaciones de la consulta join [38].

LTJ se basa en una estructura de datos abstracta llamada *Trie Iterator* que implementa una operación principal denominada `leap`. Dado una variable x, un patrón de triples t y una constante c, `leap`(x, t, c) retorna la menor constante $c_x \geq c$, tal que t tenga soluciones en la relación evaluada al reemplazar x por c_x. Si no existe tal valor c_x, retorna un símbolo especial $\perp$ representando vacío. Para lograr que Leapfrog Triejoin sea Worst Case Optimal, basta que los *Trie Iterators* soporten la operación `leap` en $O(\log n)$.

Se recuerda que las consultas se conforman por triples cuyos valores pueden ser constantes o variables. Antes de comenzar el algoritmo, se define un Global Attribute Order (GAO), es decir, un vector que contiene las variables encontradas en la consulta ordenadas de acuerdo a su prioridad. Luego, LTJ mantiene un nodo actual, en algún *trie*, por cada triple que conforma la *query*. Al comenzar el procesamiento, se resuelven aquellos valores constantes descendiendo por el *trie* adecuado, dado por el atributo de la constante. El resto de triples quedan en la raíz de su *trie*. A continuación, cuando se debe instanciar una variable x, se intersectan los hijos de todos los nodos que corresponden a x. Para aquello se debe haber elegido, para cada triple, un *trie* tal que el atributo de x esté justo a continuación en el orden establecido, lo que es posible debido a que se definió un GAO previamente. Iterando sobre este proceso se obtiene la intersección de las relaciones al evaluar cada variable según el *Trie Iterator*.

Leapfrog Triejoin ha sido exitosamente implementado para el lenguaje SPARQL [20]. Además, se han propuesto algunas optimizaciones para este algoritmo, como el uso de caché de consultas para acelerar las respuestas [23], lo cual se podría traducir en una futura optimización del Ring.

2.3 Bases de datos de grafos

Una base de datos de grafos se refiere a aquellos modelos donde los datos o esquemas son representados como grafos o mediante estructuras de datos que generalizan la noción

de grafos [2]. Su aplicación también ha sido evaluada en diversos escenarios prácticos como relaciones entre documentos y data biológica [8, 36].

Para consultarlos se utiliza como elemento básico los *Basic Graph Patterns* (abreviado BGP) [4]. Sea $\mathcal{V}$ el conjunto infinito de variables y $\mathcal{U}$ las constantes que conforman el grafo. Entonces, un *triple pattern* es una tupla sujeto-predicado-objeto, cuyos valores $(s, p, o) \in (\mathcal{U} \cup \mathcal{V})^3$. Así, un BGP es un conjunto finito de *triple patterns*. Para evaluar el BGP sobre un grafo, las variables de los triples se reemplazan por constantes pertenecientes a $\mathcal{U}$, de modo que el patrón exista en el grafo. Entonces, cada *triple pattern* es una query atómica y, por lo tanto, evaluar el BGP sobre un grafo es equivalente a una consulta *join* o, en el caso más simple, a la selección de un atributo del triple con un valor específico.

En la actualidad, existen distintos motores de bases de datos diseñados para este tipo de modelos, tales como Apache Jena, Blazegraph, Neo4j, Virtuoso y RDF-3X, entre otros [20, 30, 31, 1]. Cada opción presenta distintas propiedades sobre el almacenamiento, indexado, *joins* y *queries*, lo que las diferencia entre sí. Cabe destacar que, a pesar de que varias de estas opciones funcionan exclusivamente para grafos RDF, existen formas de hacer un mapeo de RDF a grafos con propiedades y viceversa [3].

Sin embargo, tanto en el mercado como en la academia, hay escasas alternativas que logran realizar joins WCO. Una de las opciones que cumple con ser Worst Case Optimal es Tentris, una tienda de triples RDF basada en tensores [7]. Esta solución utiliza una estructura denominada *hypertrie* para almacenar los triples y luego emplea álgebra de tensores para llevar a cabo consultas SPARQL. No obstante, es importante destacar que esta base de datos se considera estática, es decir, no permite actualizaciones de datos.

La estructura más cercana al Ring, en cuanto a rendimiento en espacio y tiempo, es el Qdag [5], el cual utiliza una representación del grafo basada en Quadtrees que corre en memoria principal. Este índice comprimido también cumple con ser WCO.

Otro motor para grafos que cumple con la propiedad de joins WCO es una modificación sobre Apache Jena para utilizar Leapfrog-Triejoin, llamada Jena LTJ [20]. Lo anterior se implementó indexando B+-trees. Este caso sí logra ser dinámico siendo, por lo tanto, la mejor alternativa para comparar la nueva implementación que busca crear este trabajo guiado.

Existen variados estudios que cuantifican y comparan el rendimiento de modelos y motores para datos de grafos [13, 1, 22, 27]. Entre ellos hay estudios sobre motores dinámicos [13]. Sin embargo, en este se prioriza la medición de tiempos en las consultas en lugar del espacio utilizado. Si bien se obtienen buenos resultados en la optimización de consultas, al realizar los experimentos con una cantidad masiva de datos, varios motores no son capaces de representar el grafo. Esto evidencia la escasez de alternativas óptimas en espacio que existen en el mercado.

2.4 Estructuras de Datos Compactas

Las estructuras de datos compactas desempeñan un papel fundamental en los algoritmos implementados en las bases de datos mencionadas en este trabajo. No solo deben cumplir

con buenos tiempos en las operaciones que realizan, sino que también deben ser capaces de comprimir la información almacenada de manera eficiente.

Estas estructuras son las bases de la arquitectura del Ring y otros esquemas presentados en esta memoria. Muchas de estas estructuras se encuentran disponibles en la librería SDSL (disponible en `https://github.com/simongog/sdsl-lite`), implementadas en C++11. Cabe destacar que estas implementaciones son estáticas y no permiten las operaciones de actualización.

2.4.1 Bitvectors

Un *bitvector*[29] se define como una secuencia de bits $B[1..n]$ que soporta las siguientes operaciones:

- $access(B, i)$: retorna el bit $B[i]$ para cualquier $0 < i \leq n$.
- $rank_v(B, i)$: retorna el número de ocurrencias del bit v en el rango $[1, i)$
- $select_v(B, j)$: retorna la posición de la j-ésima ocurrencia del bit v.

Estos vectores se pueden representar de forma plana utilizando n bits, para lo cual se han planteado estructuras que permiten realizar las tres operaciones mencionadas requiriendo un espacio de $o(n)$ bits extra [9, 21].

Los *bitvectors* son parte esencial de las estructuras compactas, pues variados esquemas de compresión almacenan los bits en estos vectores y, utilizando las funciones *rank* y *select*, son capaces de recuperar la información original [12, 34]. Estas operaciones suelen tener optimizaciones a bajo nivel, lo que les permite calcular rápidamente recuentos de bits.

2.4.2 Gap-Encoded Bitvectors

Si se tiene una secuencia creciente de m enteros pertenecientes al intervalo $[1, n]$, se puede representar la secuencia utilizando *gap-encoded bitvectors*. En este esquema, cada elemento $x_1, x_2, ..., x_m$ de la secuencia se almacena en un *bitvector* como $\mathtt{0}^{x_1}\mathtt{1}\mathtt{0}^{x_2-x_1}\mathtt{1}...\mathtt{0}^{x_m-x_{m-1}}\mathtt{1}$, donde $\mathtt{0}^k$ son k ceros consecutivos [34]. Es decir, para cada $i \in [1, m]$, se almacenan $x_i - x_{i-1}$ ceros consecutivos y luego un 1, tomando $x_0 = 0$. De esta manera, se puede obtener cada x_i realizando $select_1(B, i)$ en tiempo constante. Sin embargo, si m es muy pequeño comparado con n, el vector queda con muy pocos unos. En este caso, tomando $s_i = x_i - x_{i-1}$, el espacio utilizado es $\Sigma_{i=1}^{m} \log s_i + O(m)$ bits.

2.4.3 Wavelet Trees y Wavelet Matrix

Los *wavelet trees* [28] se definen como un árbol binario que representa un string $S[1..n]$ de un alfabeto $[1,\tau]$ usando $n \log_2 \tau + o(n \log \tau)$ bits de espacio.

La estructura es de la siguiente forma: cada nodo representa un rango de símbolos del alfabeto. Las hojas representan un único símbolo. Si un nodo representa $[a, b]$, entonces el hijo izquierdo representa $[a, \lfloor (a+b)/2 \rfloor]$ y el hijo derecho $[\lfloor (a+b)/2 \rfloor + 1, b]$. El nodo solo almacena un string de bits donde los que pertenecen al hijo izquierdo se representan con un 0 y los del derecho con un 1. En la Figura 2.1 se muestra un ejemplo de un wavelet tree.

Para minimizar aún más el espacio ocupado por la estructura se puede notar que toda la información está en los bits, por lo que los punteros y nodos pueden ser omitidos en la implementación. Usando esta idea se crea la *wavelet matrix* [10]. Esta estructura es la misma representación de la idea abstracta de los *wavelet trees* pero, en vez de utilizar una forma de árbol binario, utiliza una matriz.

En esta matriz cada fila representa un nivel del árbol binario. Para pasar de un nivel a otro, todos los bits marcados con 0 van a la izquierda y todos los 1 van a la derecha. Formalmente, si se tiene una *wavelet matrix* B, se denota un nivel de la matriz como B_ℓ. Entonces, si $B_\ell[i] = 0$, su posición correspondiente en el siguiente nivel es $rank_0(B_\ell, i)$. Mientras que si $B_\ell[i] = 1$, su posición en el siguiente nivel es $z_\ell + rank_1(B_\ell, i)$, con z_ℓ la cantidad de ceros en el nivel B_ℓ [10].

Así, se tienen $\log_2 \tau$ niveles, cada uno con n bits. Para poder bajar por la matriz se pueden usar las operaciones de *rank* y *select*, siguiendo la definición de la construcción de cada nivel de la estructura.

Es preciso señalar que cada nivel de la matriz es un *bitvector* por lo que se tienen disponibles las operaciones de *access*, *rank* y *select*. Además, tanto *wavelet trees* como *wavelet matrix*, deben implementar las operaciones *access*, *rank* y *select* pero a nivel de símbolos de un string S, es decir:

- $access(S, i)$: retorna $S[i]$.
- $rank_a(S, i)$: retorna el número de ocurrencias del símbolo a en S dentro del rango $[1, i)$
- $select_a(S, j)$: retorna la posición de la j-ésima ocurrencia del símbolo a en S.

Además, estas estructuras permiten operaciones esenciales para llevar a cabo la versión del algoritmo del Leapfrog-Triejoin utilizada en el Ring, como lo son `range-next-value` y `range-intersection-queries` [18]. Estos métodos permiten simular el movimiento entre tries que necesita LTJ para obtener una respuesta. La implementación se resuelve usando las operaciones de *access*, *rank* y *select* sobre los bits de la *wavelet matrix*.

2.4.4 Plain Front Coding

El *Plain Front Coding* es una estructura comprimida utilizada para almacenar diccionarios ordenados lexicográficamente [26]. Este esquema es especialmente útil para conjuntos de strings que comparten prefijos largos. Se aprovecha del hecho de que, si existe un orden lexicográfico, entradas contiguas tenderán a compartir un prefijo largo.

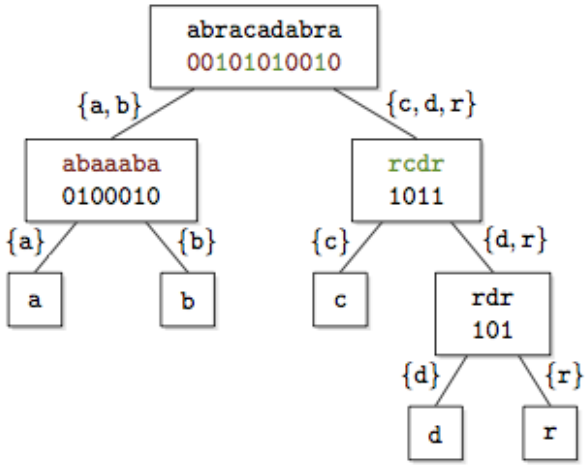


Figura 2.1: Ejemplo de un wavelet tree de la palabra *abracadabra*. Figura sacada de `https://en.wikipedia.org/wiki/File:Wavelet_tree.png`

Un Plain Front Coding, abreviado PFC, está compuesto por una cabecera y entradas. La cabecera es el primer string S_0 almacenado y, por lo tanto, el menor lexicográficamente. Luego, cada string del diccionario S_i se codifica con respecto al anterior. La codificación es un número representando el máximo prefijo común ℓ_i entre S_{i-1} y S_i, seguido del string S_i empezado desde la posición $\ell_i + 1$.

Utilizando el ejemplo de la Figura 2.2, la cabecera sería el string `a`. Luego, la segunda palabra es `alabada` donde el prefijo en común con `a` es de largo 1. Por lo tanto, la entrada es un `1` seguido del string `labada`. De esta manera, se va almacenando la información sin la necesidad de representar directamente la palabra completamente. Cabe destacar que el PFC resulta en un único string donde cada entrada se separa por un símbolo especial del alfabeto \$.

La obtención del string S_i del diccionario, bajo el esquema del Plain Front Coding, requiere leer todos los strings $S_j, j \in [1, i-1]$ para poder obtener su prefijo.

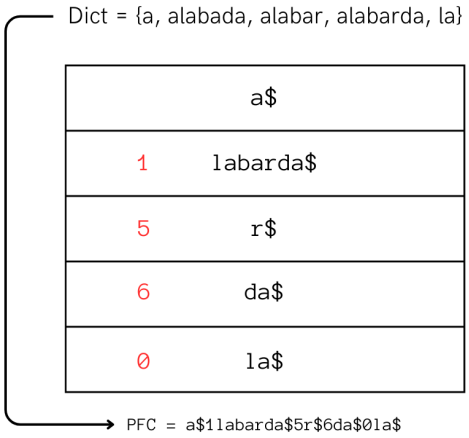


Figura 2.2: Ejemplo de compresión usando Plain Front Coding

2.5 Estructuras compactas dinámicas

Si bien existen variados estudios sobre estructuras compactas donde el objetivo es optimizar el espacio ocupado, la mayoría de estos ven las estructuras como estáticas. A pesar de ello, existen trabajos donde se modifican estructuras comprimidas para admitir el dinamismo, es decir, inserciones y eliminaciones [34, 12, 37, 25].

Un *framework* importante es la librería DYNAMIC (`https://github.com/xxsds/DYNAMIC`) de Nicola Prezza [34], la cual implementa varias estructuras comprimidas dinámicas, incluyendo *bitvectors* y *wavelet trees*. Además, este repositorio de código es público y open-source bajo una licencia MIT. Las estructuras utilizadas que se encuentran en la librería se explican con mayor detalle en la Sección 4.1.

2.6 Ring

El Ring busca almacenar un conjunto de tuplas que representan relaciones. Los atributos forman parte de un universo de símbolos de tamaño U. Para tener un orden se mapean todos los símbolos a un conjunto de enteros positivos consecutivos $[1..U]$. Entonces, una relación es vista como un conjunto de n tuplas pertenecientes a U^d, con d la cantidad de atributos.

Esta estructura compacta logra correr consultas join en tiempo *worst case optimal*, utilizando poco espacio extra al necesario para almacenar todas las tuplas de las relaciones. Al ser comparado con otras alternativas similares logra resultados superiores en tanto tiempo como espacio utilizado [4].

A pesar de que el Ring puede ser adaptado para soportar relaciones de cualquier aridad, en este trabajo se hablará del Ring especializado para almacenar grafos. Esto significa que se almacenaran triples de la forma `spo`, ya que las aristas de los grafos se pueden representar como una relación de `sujeto-predicado-objeto`. Por ello, se utilizará una aridad de $d = 3$. De este modo, cualquier relación almacenada en el Ring se representará como `s` (Sujeto), `p` (Predicado) y `o` (Objeto).

2.6.1 Elementos del Ring

El Ring se puede pensar como una tabla T de elementos pertenecientes a $[1, U]$ y un orden Π. El Ring tiene tantas columnas como atributos hayan, por lo tanto, para almacenar grafos se tendrían únicamente 3 columnas. Por otra parte, cada fila de T representa una tupla almacenada. De esta manera, se define la celda $T[i][j]$ como el atributo $\Pi(j)$ de la tupla i. En otros términos, el Ring es un conjunto de columnas C_j, donde cada columna es un re-ordenamiento de la tabla. El orden de la columna está dado por el primer atributo del orden de la relación.

Para grafos se tienen tres atributos, `S,P` y `O`, por lo tanto, se necesitarían tres columnas:

- C_o representando `SPO` y, por lo tanto, ordenado por `S`, desempatando por `P`, y si aún

quedan empates, resolviendo por `O`.

- C_p representando `OSP` y, por lo tanto, ordenado por `O`, desempatando por `S`, y si aún quedan empates, resolviendo por `P`.
- C_s representando `POS` y, por lo tanto, ordenado por `P`, desempatando por `O`, y si aún quedan empates, resolviendo por `S`.

Un ejemplo se puede ver en la Figura 2.3. Cada columna se almacena en el Ring como una *wavelet matrix*. Sin embargo, para poder ser capaces de identificar una tupla de una columna a otra es necesaria una estructura adicional, perteneciente al Ring. Cada columna C_j tiene un arreglo A_j definido como

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

Es decir, por cada elemento $c \in C_j$ el arreglo A_j, representado como *bitvector*, almacena la cantidad de ocurrencias de símbolos menores que c. Cada arreglo se almacena junto a la columna que ordena, en otras palabras, A_s se almacena junto a C_o ya que el orden es `SPO`.

Es importante destacar que las columnas del Ring representan la información de los *tries* usados en Leapfrog Triejoin. En alguna de las tres columnas C_j existe, para cada nodo posible de cada unos de los 6 *tries*, un rango que corresponde a todos los hijos del nodo. Es decir, para el string s construido a partir del camino desde la raíz al nodo en el *trie*, el rango contiene todos los strings que comienzan con s.

En el paper original del Ring se presentan dos formas de indexación, con los *bitvectors* de las *wavelet matrices* en forma plana o comprimida. Esta compresión está dada por la librería SDSL y ocupa un término b, donde a mayor b mayor compresión pero peores tiempos de operaciones. La versión plana es llamada Ring, mientras que la versión comprimida se denomina C-Ring, la que utiliza $b := 16$.

Figura 2.3: Ejemplo de las columnas que componen a un Ring.

2.6.2 Movilización entre columnas

Dadas estas estructuras, lo que permite al Ring utilizar sólo 3 ordenes, es la posibilidad de traducir un índice desde una tabla a otra. Esto es posible debido a la función biyectiva $F_j : [1..n] \to [1..n]$ definida como

$$F_j(i) := A_j[c] + rank_c(C_j, i) \tag{2.1}$$

Con $c := C_j[i]$. Entonces, si T' es un re-ordenamiento de la tabla T por una columna j e $i' := F_j(i)$, entonces las filas $T[i]$ y $T'[i']$ corresponden a la misma tupla [4]. Por ejemplo, utilizando $F_\mathtt{o}$ se podría hacer una traducción `spo` $\rightarrow$ `osp`, ya que este último es un re-ordenamiento por la columna `o` representada por `spo`. Esto permite acceder a las tuplas almacenando únicamente la última columna de cada tabla.

La inversa de F_j es

$$F_j^{-1}(i') := select_c(C_j, i' - A_j[c])$$

Con c tal que $A_j[c] < i' \leq A_j[c+1]$. Esto se puede verificar sabiendo que `select` es la inversa de `rank`. La transformación de índices se expande a intervalos donde se puede transformar un intervalo contiguo $[s, e]$ a un intervalo $[s', e']$ en otra columna. Esto permite al Ring simular el descenso por los nodos de los *tries* utilizados por LTJ, puesto que la columna almacena los hijos del nodo en el que se está y el mapeo del intervalo, restringido por una constante, a la columna anterior (en el orden cíclico definido en el Ring) lleva al rango del hijo del nodo actual por el cual se desciende.

Adicionalmente, al evaluar una variable x en Leapfrog Triejoin se deben poder intersectar los hijos de los nodos de x. Esta operación equivale a encontrar los valores comunes en todos los rangos de las columnas C_j correspondientes, lo que se lleva a cabo con la operación `range-next-value`. Por lo tanto, es posible implementar LTJ utilizando las estructuras y funciones definidas sobre el Ring.

2.6.3 Representación Dinámica de grafos

Una posible representación de grafos que permite dinamismo es el planteado por Coimbra et al. [11] para Quadtrees, la que está basada en una estructura llamada k^2-tree. Estos árboles consisten en una representación compacta de la matriz de adyacencia del grafo que soporta eficientemente consultas sobre vecinos, búsqueda de rangos y navegación hacia adelante y hacia atrás.

Entonces, la solución que plantean es tener un conjunto $C = \{E_0, ..., E_r\}$ de aristas donde cada set E_i es un k^2-tree estático excepto por E_0 que es representado por una lista de adyacencia dinámica. Si bien esta solución era una posible alternativa a la implementación del Ring dinámico, la solución actual se vio más promisoria por lo que no fue utilizada en este trabajo. Esto debido a que lo planteado por Coimbra et. al. tiene buenos tiempos amortizados [11], sin embargo, de-amortizarlos requiere que la estructura utilice mayor espacio. No obstante, los resultados que se presentan en el trabajo planteado permiten predecir de buena manera cómo le iría a esta solución aplicada al Ring.

2.7 Diccionarios comprimidos

Los Diccionarios comprimidos son estructuras definidas sobre un alfabeto y un conjunto de strings. Almacenan un mapeo de strings a identificadores, lo que es utilizado en índices de bases de datos, debido a que es más fácil operar con números que con strings. Estos

diccionarios deben permitir dos operaciones principalmente:

- `locate`(s): Esta función traduce el string s a un identificador i
- `extract`(i): Esta función retorna el string s correspondiente al identificador i

Con esto puede existir una traducción de string a identificador y viceversa. Se han planteado varias estructuras para diccionarios comprimidos, sin embargo, en su mayoría son estáticos. Entre estos se encuentran tablas de hashing, Tries comprimidos y Front Coding. Estos diccionarios pueden almacenar los strings en forma plana o utilizando alguna compresión como, por ejemplo, codificación de Hu-Tucker [26].

Si bien existen adaptaciones dinámicas para compresores como Huffman, Hu-Tucker y de gramática (por ejemplo, Re-Pair), estos esquemas no funcionan bien para el problema del Ring dinámico donde las frecuencias de caracteres cambian constantemente y, por lo tanto, se deberían codificar nuevamente todas las palabras y no únicamente los strings que se van añadiendo. En esta memoria se decidió utilizar Plain Front Coding debido a que muestra el mayor impacto en la compresión del diccionario y resulta ser la única estructura que se puede mantener razonablemente en un entorno dinámico [26].

Capítulo 3

Problema

Como se ha mencionado anteriormente, el Ring se ha demostrado experimentalmente como una estructura altamente eficiente en términos de espacio y tiempo. Al utilizar el algoritmo de Leapfrog Triejoin y estar construido a base de estructuras compactas, se convierte en una opción muy sólida en comparación con otras bases de datos de grafos disponibles en el mercado y en la comunidad científica.

Sin embargo, su caso de uso se encuentra limitado debido a que es una estructura estática. Esto implica que cualquier actualización a los datos representados requiere de una nueva construcción del Ring. Además, el Ring sólo admite identificadores en sus consultas, lo que implica la necesidad de realizar una traducción previa del conjunto de datos completo antes de poder construir la estructura o extraer información.

La incapacidad de realizar modificaciones en el Ring se debe a las características de las estructuras compactas que lo componen, las cuales no permiten inserciones o eliminaciones de bits. Adicionalmente, carece de un diccionario que pueda servir como interfaz entre el alfabeto de identificadores usado por el Ring y el alfabeto de símbolos usado por el usuario. No resulta difícil combinar el Ring con un diccionario estático [26], sin embargo, para el caso donde se requieren actualizaciones, no existen implementaciones de diccionarios comprimidos dinámicos. Sería, entonces, valioso implementar tanto las operaciones de actualización, como un diccionario dinámico y compacto propio del Ring.

Esta propuesta sería beneficiosa, ya que permitiría que el Ring se adapte a una mayor variedad de casos de uso y pueda ser utilizado en distintos proyectos que requieran una solución eficiente para almacenar grafos en memoria principal. Además, muchas de las opciones con las que se compara el Ring sí admiten actualizaciones y utilizan el lenguaje SPARQL para realizar sus consultas, por lo que se podría comparar con mayor fidelidad el rendimiento de la estructura.

La solución planteada se consideraría suficientemente buena si, al permitir las nuevas operaciones de inserción y eliminación, así como la implementación del diccionario, el espacio ocupado sigue siendo mucho menor que las alternativas y el tiempo de consultas join se mantiene competitivo para utilizar un algoritmo WCO.

Capítulo 4

Solución

En este capítulo, se abordará el diseño e implementación de la arquitectura del Ring dinámico. Se proporcionará una descripción detallada de la propuesta, teniendo en cuenta las adaptaciones necesarias para permitir operaciones de inserción y eliminación. Además, se explicará la implementación del diccionario asociado al Ring, el cual actúa como una interfaz entre el alfabeto de identificadores, utilizado por el índice, y el alfabeto de símbolos perteneciente al usuario. Por último, se presentará la implementación de las operaciones de actualización en el modelo desarrollado, brindando una visión completa de todos los componentes claves del sistema.

4.1 Arquitectura del Ring dinámico

Como se mencionó en las secciones anteriores, el Ring se compone de dos estructuras fundamentales: *wavelet matrices* y *bitvectors*. Para asegurar la preservación de los algoritmos presentados en el trabajo original del Ring y mantener joins WCO, es crucial que estas estructuras sean capaces de llevar a cabo eficientemente dos operaciones principales:

- $\texttt{rank}_b(B, i)$: el número de bits iguales a $b \in \{0, 1\}$ en $B[1..i]$.
- $\texttt{select}_b(B, j)$: la posición de la ocurrencia número j del bit b en B.

Con el fin de lograr este objetivo y considerando las distintas implementaciones disponibles, se modificaron las estructuras presentes en la librería DYNAMIC de Nicola Prezza[1] [34]. Esta decisión de diseño se basó en los buenos resultados de compresión y rendimiento que tienen los *bitvectors* y *wavelet matrix* dinámicos disponibles en dicha librería. Además, el hecho de que la implementación de DYNAMIC utilice C++11 facilitó la integración de estas estructuras en el código existente del Ring.

[1] Código disponible en `https://github.com/xxsds/DYNAMIC`

4.1.1 Bitvectors

En el Ring estático, los *bitvectors* son arreglos de memoria contigua de gran tamaño donde se almacenan bits. Esto implica que cualquier inserción de un bit en una posición que no sea el final requiere la reconstrucción completa del *bitvector*. Por esta razón, esta estructura no es modificable para permitir actualizaciones.

Por el contrario, el diseño propuesto por Nicola Prezza en DYNAMIC para *bitvectors* se basa en árboles B [34]. Esta estructura está diseñada para resolver el problema de Sumas Parciales Buscables con Inserción, o SPSI por sus siglas en inglés. Este problema consiste en encontrar una estructura de datos `P` que admita una secuencia de enteros no negativos $s_1, s_2, ..., s_m$ y soporte las siguientes operaciones:

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

Además, es necesario que la estructura admita tanto inserciones como actualizaciones; esto es, cambiar los valores de s_i o aumentar el tamaño de la secuencia agregando un nuevo entero s_{m+1}. Se puede notar que si la secuencia está compuesta únicamente de bits, entonces `P.sum`(i) es equivalente a `rank`$_1(i)$ y `P.search`(x) corresponde a `select`$_1(x)$. De esta manera, al solucionar el problema SPSI, se tiene una estructura de datos que funciona como *bitvector* con actualizaciones.

Nicola Prezza propone una solución utilizando un B-Tree, donde las hojas son *bitvectors* representados como arreglos de enteros de 64 bits. Esta estructura de datos se denomina `spsi`, en referencia al problema que resuelve. La implementación en DYNAMIC utiliza dos parámetros: `B`, definido como la cantidad de hijos que puede tener un nodo, y `B`$_{leaf}$, que establece la cantidad de bits $\mathcal{L}$ almacenados en una hoja.

En la implementación original de Nicola Prezza se debía cumplir la condición `B`$_{leaf} < \mathcal{L} <$ 2`B`$_{leaf}$. Sin embargo, esto significa que al dividir una hoja en 2 (operación `split`) ésta quedaba con un tamaño `B`$_{leaf}$. Lo anterior la hace muy susceptible a una fusión al momento de remover un bit de la estructura, impactando los tiempos de eliminación. Es por esto que se modificó la condición a `B`$_{leaf} < \mathcal{L} <$ 3`B`$_{leaf}$ con el fin de que al dividir una hoja se tengan 1,5`B`$_{leaf}$ bits.

Asimismo, cada nodo almacena valores pre-calculados con el propósito de agilizar las operaciones de `sum` y `search`. Cada nodo contiene dos vectores de tamaño 2`B` + 2, uno llamado `subtree_sizes` para almacenar el tamaño de los sub-árboles de los hijos; y otro, `subtree_psum` que guarda la suma parcial acumulada a la izquierda de cada hijo.

Cabe destacar que existen ciertas garantías teóricas de `spsi`. Sea $M = m + \Sigma_{i=1}^{m} s_i$, luego la estructura de sumas parciales implementada en DYNAMIC toma a lo más

$$2m(\log(M/m) + \log\log m + O(\log M/\log m))$$

bits, y es capaz de realizar las operaciones de `sum, search` y actualizaciones en tiempo $O(\log m)$ [34]. En términos teóricos, los tiempos de estas funciones son $O(1)$ en las estructuras

estáticas y $O(\log m)$ en las dinámicas, por lo que se puede esperar una diferencia de un orden de magnitud. Por lo tanto, es necesario mencionar que se espera un aumento en los tiempos de operaciones en comparación con la implementación basada en SDSL del Ring estático. En la práctica, lo anterior se debe a que, para realizar tanto `select` como `rank`, se requiere descender por el árbol para encontrar la hoja correcta y luego realizar los cálculos correspondientes. No obstante, esto no significa que DYNAMIC presenta una implementación poco óptima dado que existe una cota inferior $\Omega(\log n / \log \log n)$ para cualquier estructura que soporta inserciones y `sum` [17].

Dentro del Ring dinámico, los *bitvectors* se encuentran presentes como parte de las *wavelet matrix* y como los arreglos A_j a lo largo de la estructura, como se muestra en la Figura 4.1. Es preciso señalar que los valores de estos arreglos son sumas parciales y, por ende, `spsi` es una estructura perfecta para representarlos.

Se decidió utilizar la interfaz `succint_bitvector` de DYNAMIC como clase para los arreglos A_j, que ocupa un `spsi` con $\mathtt{B} = 16$ y $\mathtt{B}_{leaf} = 8192$. Estos valores se basan en resultados teóricos y experimentales obtenidos en el paper DYNAMIC donde, empleando estos parámetros, se obtiene un espacio usado de $n + o(n)$ bits, siendo n el largo del *bitvector* [34]. Estos vectores simplemente almacenan bits usando un `spsi`, sin embargo, el Ring se encarga que las inserciones y manipulación de datos sean consistentes con una codificación *gap-encoded*. Para los *bitvectors* de las *wavelet-matrix* se usaron otros valores para los parámetros `B` y $\mathtt{B}_{leaf}$.

Cabe considerar, por otra parte, que los grafos de gran tamaño tienden a tener un alfabeto de nodos `SO` y de aristas `P` más bien disperso, especialmente en el grafo de Wikidata [39]. En consecuencia, se decidió representar el arreglo como un *gap-encoded bitvector*, que almacena la secuencia $s_1, s_2, ..., s_n$ de frecuencias de elementos almacenados. Así, se define $A_j[i] = select_1(A_j, i) - i$, mientras que las operaciones de actualización quedan simplificadas a agregar o quitar un cero de la secuencia, según corresponda.

4.1.2 Wavelet Matrix

Para representar las columnas C_j, el Ring estático utiliza *wavelet matrices* como estructura compacta debido a que define un orden entre los elementos que almacena. Además, es posible implementar funciones necesarias para llevar a cabo el algoritmo de Leapfrog Triejoin. Dado que estas estructuras se basan en *bitvectors*, la versión disponible en la librería SDSL tiene los mismos problemas para implementar las operaciones de actualización.

La *wavelet matrix* soporta tanto `rank` como `select` en tiempos eficientes, junto con otras acciones que facilitan el recorrido de las columnas en el Ring. Adicionalmente, se pueden modificar para soportar inserciones y eliminaciones [28]. Esto se logra usando las clases disponibles en la librería DYNAMIC. Esta estructura se basa en un arreglo mutable de *bitvectors*, donde cada uno representa un nivel. Cabe destacar que lo anterior, implica que la matriz también utiliza los parámetros `B` y $\mathtt{B}_{leaf}$. Los valores utilizados en la implementación final son $\mathtt{B} = 32$ y $\mathtt{B}_{leaf} = 2048$, lo que se discute en el Capítulo 5.

Paralelamente, se almacena un arreglo con el índice de cada nivel donde empiezan los *unos* o hijos derechos del árbol. Esto equivale a la cantidad de ceros del nivel anterior; lo que

resulta útil para ahorrar llamadas a $\texttt{rank}_0()$ cada vez que se quiere navegar hacia la derecha del árbol representado.

A pesar de incluir `rank` y `select`, la versión presente en DYNAMIC de las matrices, no incorporaba todos los métodos necesarios para poder soportar el algoritmo de Leapfrog Triejoin. Es por esto que parte del trabajo fue la adición de las siguientes funciones a la estructura:

- `select_next`(i, x): Retorna la posición del siguiente elemento x fuera del rango $[0, i]$.
- `range_next_value`(x, i, j): Retorna el menor valor c en el rango $[i, j]$, tal que $c \geq x$.
- `all_values_in_range`(i, j): Retorna un vector con todos los valores en el rango $[i, j]$.

La implementación de `select_next` consiste en bajar por la matriz, acotando el intervalo según el bit de x & $(1 \ll \mathit{depth})$, donde *depth* es la profundidad en la que se encuentra. Al llegar al último nivel, se vuelve a subir encontrando siempre el siguiente 1 o 0, según corresponda. Por otra parte, las funciones de `range_next_value` y `all_values_in_range` fueron programadas de forma recursiva. Ambas están basadas en descender por la matriz, modificando los intervalos y utilizando `rank` para moverse por los hijos del árbol representado. Estas tres funciones permiten simular con éxito el bajar por los *tries* de relaciones, estructura sobre la que se basa el algoritmo de Leapfrog Triejoin. Es preciso aclarar que el Ring estático ya había implementado estas funciones en SDSL, por lo que sólo fue necesario adaptarlas para ser usadas dentro de la *wavelet matrix* de DYNAMIC.

De esta manera, se adaptó la estructura para ser utilizada por el Ring dinámico. Finalmente, se incluyó el *bitvector* para A_j, junto con la *wavelet matrix* C_j, en una clase llamada `dyn_bwt` cuyo propósito es ser una interfaz para el Ring. Se presenta en la Figura 4.1 un esquema de la arquitectura completa del Ring dinámico. En ella, se puede observar que para relaciones `SPO` existen tres `bwt_dyn`, cada uno conformado por una *wavelet matrix*, donde cada nivel es un `spsi`, y un `succint_bitvector`.

4.2 Diccionario compacto

El Ring está definido sobre un universo $[1, U]$ de índices, los que simbolizan el alfabeto del usuario. Sin embargo, la traducción desde el lenguaje de los datos a identificadores del Ring se debe llevar a cabo antes de crear la estructura. Esto dificulta las operaciones de actualización y consulta ya que, para cada operación que se quiera realizar, se deben traducir los valores de la base de datos a índices del Ring de forma manual. Por esta razón se hace necesaria una estructura que permita traducir los alfabetos sin que ambas partes estén involucradas.

Para lograr lo anterior se deben cumplir ciertos objetivos. Sea $\mathcal{D}$ el conjunto de palabras utilizadas por el usuario, con $|\mathcal{D}| = U$. Es necesaria una estructura que sea capaz de realizar las siguientes operaciones:

- `locate` : $\mathcal{D} \rightarrow [1, U]$, donde `locate`$(s)$ retorna el identificador usado por el Ring para el string s.

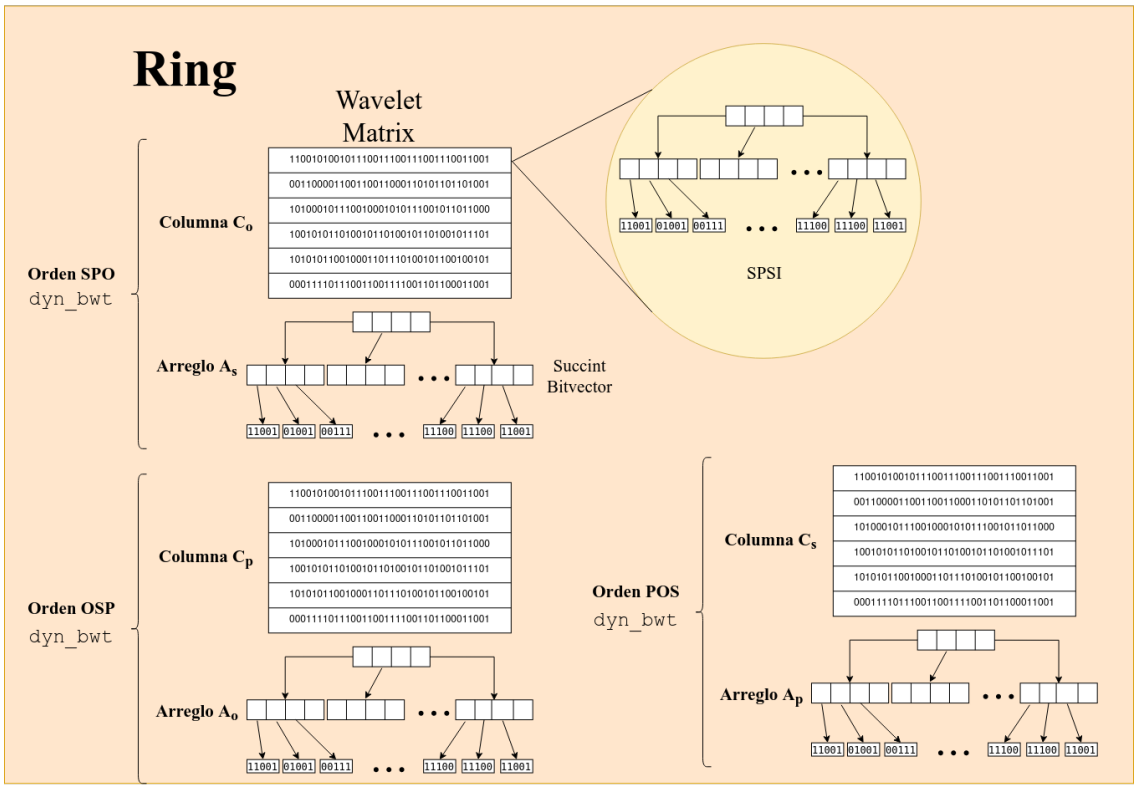


Figura 4.1: Diagrama del Ring dinámico para aridad 3.

- `extract` : $[1, U] \to \mathcal{D}$, donde `extract`(x) retorna el string representado por el ID x.

Sumado a esto, se requiere que la estructura sea dinámica para poder soportar las inserciones y eliminaciones en $\mathcal{D}$. Por último, se debe priorizar la eficiencia de `extract` por sobre `locate`. Lo anterior se debe a que las consultas a la base de datos requieren pocas palabras de parte del usuario, mientras que el resultado puede contener miles de valores. Es por ello que el rendimiento se verá afectado principalmente por la traducción de IDs a strings.

Es importante añadir que, a diferencia de los diccionarios estáticos, la implementación dinámica debe almacenar los identificadores junto a las palabras, puesto que se debe mantener una relación entre cada string del diccionario y su ID a lo largo de las actualizaciones. En consecuencia, no es posible utilizar su posición en el orden lexicográfico como identificador.

Martínez-Prieto et. al. estudian distintos esquemas de diccionarios comprimidos sobre variados tipos de data [26]. De sus resultados se pudo concluir que, para almacenar URLs y URIs, la mejor compresión es lograda por los Plain Front Coding. Debido a que en esta memoria la implementación se enfocó en los grafos Web [24], se decidió utilizar esta estructura para el diccionario dinámico. La ventaja del enfoque en este tipo de grafos es que la mayoría de valores de nodos y aristas son URLs. Esto cumple con que la mayoría de los strings del diccionario comparten prefijos comunes, puesto que los URLs comienzan con valores como `http://` o `https://` y tienden a compartir dominios.

En función de lo planteado, se decidió utilizar la estructura compacta Front Coding como almacenamiento de strings. Esto permite una compresión basada en la similitud de prefijos,

altamente apta para el problema planteado. Por otra parte, existen distintos codificadores compactos, como Huffman o Re-Pair, no obstante, al encontrarse en un contexto dinámico se dificulta la mantención de la codificación a lo largo de la estructura. Por lo tanto, se optó por utilizar la versión sin codificación, es decir, Plain Front Coding. Sumado a esto, dada la naturaleza de los diccionarios a utilizar, se espera que la compresión por prefijos sea suficientemente buena.

4.2.1 Traducción de String a ID

El Plain Front Coding [26], abreviado PFC, almacena el diccionario $\mathcal{D} = \{s_0, s_1, s_2, ..., s_U\}$, donde $\forall i \in [0, U-1], s_i < s_{i+1}$ lexicográficamente. Se definen las funciones

- $\texttt{lcp}(s_i, s_j)$: Calcula el largo del máximo prefijo en común entre s_i y s_j.
- $\texttt{suffix}(s, i)$: Retorna la porción desde el índice i del string s.

Entonces, tomando los strings como arreglos de caracteres indexados desde 0, se define la estructura recursivamente como

$$\texttt{PFC}[0] = s_0$$
$$\texttt{PFC}[i] = \texttt{lcp}(s_{i-1}, s_i)|\texttt{suffix}(s_i, \texttt{lcp}(s_{i-1}, s_i))$$

Con | la concatenación de strings.

Sin embargo, se debe adaptar la estructura para almacenar el identificador junto a su string correspondiente. Por lo tanto, sea x_i el identificador correspondiente a s_i. Entonces se redefine la estructura como

$$\texttt{PFC}[0] = x_0|s_0$$
$$\texttt{PFC}[i] = x_i|\texttt{lcp}(s_{i-1}, s_i)|\texttt{suffix}(s_i, \texttt{lcp}(s_{i-1}, s_i))$$

Se recuerda que `PFC` es, en la práctica, un arreglo de `char`. En consecuencia, es necesario agregar una terminación a cada entrada del Plain Front Coding para poder diferenciarlos. Por ello, se reserva el `char 0` como símbolo terminal. Con el motivo de apegarse a una notación estándar se representará la terminación usando $ dentro de los esquemas.

Otra dificultad presente en la definición del `PFC` es que, al ser arreglos de `char`, se dispone de un solo byte para almacenar los identificadores y el largo del prefijo común, limitándolos a un valor entre 0 y 255. Para evitar esto, se utilizó una codificación de enteros positivos para el diccionario llamada VByte [40].

Para un número x, se almacenan porciones de 7 bits, dejando el bit más significativo en 0. Al almacenar la última porción, el bit más significativo se cambia a 1. De esta manera,

para leer un número, se leen bytes hasta que el bit más significativo sea un 1, concatenando los primeros 7 bits para armar el número x. Esto hace que se necesiten $\lceil \frac{|x|}{7} \rceil$ bytes para codificar números, con $|x|$ el número de bits necesarios para representar x. Cabe destacar que, utilizando este método, no es necesario incluir un símbolo terminal después de un número almacenado, únicamente al terminar una cadena.

Para ejemplificar, si se tiene el número `1100101001`, en el diccionario se almacenaría como dos bloques. En primer lugar, los 7 bits menos significativos y agregando un `0` en el lugar más significativo: `00101001`. Luego, queda sólo la porción `110` y, como tiene un largo menor a 7, se pone el bit más significativo como `1`, por lo que el bloque siguiente es `10000110`. Así, queda almacenado el número como `00101001|10000110`. Representando esta codificación como $\texttt{encode\_number}(x)$, la definición se modifica a

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

Un ejemplo se muestra en la Figura 4.2. En esta se puede observar que existe una cabecera con un string inicial `alabada` donde sólo se almacena el identificador codificado y la cadena en forma plana. Luego, las siguientes entradas del `PFC` llevan el número codificado `lcp` con el que se omite el prefijo. En el caso de la segunda palabra, el máximo prefijo común entre las palabras `alabar` y `alabada` es `alaba`. Por lo tanto, se codifica el número 5 a `10000101` y se incluye como parte del PFC. Finalmente, se incluye la porción de la cadena que no es parte del prefijo junto con el símbolo terminal, es decir, $r\$$.

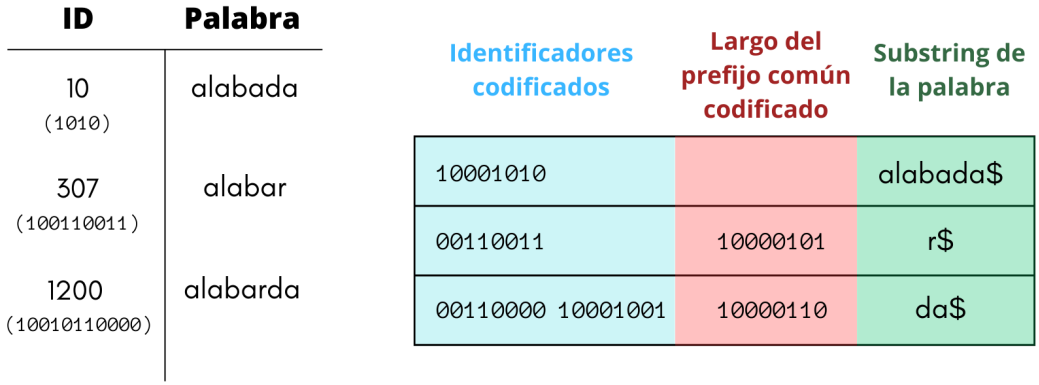


Figura 4.2: Ejemplo de un Plain Front Coding que almacena el diccionario {alabada, alabar, alabarda} con identificadores 10, 307 y 1200, respectivamente.

Es importante añadir que la obtención de una cadena $\texttt{PFC}[i]$ requiere de la lectura de todas las cadenas anteriores. Esto hace sumamente costoso la traducción de strings que se encuentran al final del Front Coding. Una alternativa planteada por Martínez-Prieto et. al. es la partición de esta estructura en *buckets*, almacenados en un arreglo [26]. Cada *bucket* es un PFC que tiene un máximo de P_{max} palabras almacenadas. Entonces, para buscar un string $s \in \mathcal{D}$ se debe hacer una búsqueda binaria por sobre las cadenas iniciales s_0 de los *bucket*.

Al encontrar el Plain Front Coding correcto, se decodifica secuencialmente hasta encontrar s. Esto resulta en un orden logarítmico para la traducción.

Con el objetivo de dinamizar el esquema, se decidió optar por un diseño de árbol binario cuyas hojas son Plain Front Codings. Se descartó el uso de un arreglo debido a que era necesario permitir que los *buckets* se dividieran y fusionaran manteniendo el orden lexicográfico. Los nodos del árbol tienen como llave una palabra del diccionario, tal que, el subárbol izquierdo tiene todas las palabras menores y el subárbol derecho todas las palabras mayores, siguiendo un orden lexicográfico. Así, la búsqueda toma un tiempo de $O(\log \lceil \frac{U}{P_{max}} \rceil)$. Sin embargo, como los PFC ya almacenan las palabras que se encuentran en las cabeceras, se añade un puntero desde el nodo a la hoja más izquierda del subárbol derecho. Así, el valor de un nodo se puede obtener leyendo la cabecera de su puntero al PFC. Un esquema se muestra en la Figura 4.3, donde se representan con línea punteada estos punteros. Entonces, al bajar por el árbol, si el valor buscado es igual a la llave del nodo se puede llegar instantáneamente a la hoja requerida.

Al plantear este esquema se introducen dos parámetros al diccionario: P_{min} y P_{max}, donde para toda hoja del árbol h_i, con una cantidad de palabras P_i, se debe cumplir que $P_{min} \leq P_i \leq P_{max}$. Tanto las operaciones de actualización, como el espacio utilizado se ven afectados por la elección de estos parámetros.

4.2.2 Traducción de ID a String

Es fácil notar que el almacenamiento compacto de palabras junto con sus identificadores no permite una eficiente implementación de `extract`. Si se quisiera encontrar un ID, la única forma sería revisar secuencialmente en todas las hojas. Esto resulta especialmente problemático, debido a que, como se planteó en un inicio, se debe priorizar la eficiencia de `extract` por sobre `locate`.

Para solucionar este problema se añade otra estructura al diccionario, un arreglo A_{ID}, con tamaño U, de punteros a las hojas del árbol. Esta estructura almacena un puntero por cada identificador donde, en la posición i, se encuentra la referencia al PFC en el que se almacena el identificador i y, por lo tanto, se puede obtener su palabra correspondiente. Se recalca que sólo se almacena una referencia a la hoja correcta, por lo que igualmente se requiere hacer una búsqueda secuencial para extraer la palabra asociada a i.

La estructura planteada no es compacta, a diferencia del diseño de los demás componentes que forman parte del Ring. Esta decisión se tomó debido a que, a pesar de que el alfabeto puede crecer indefinidamente, el universo $[1, U]$ de identificadores usado por el Ring tiende a estar acotado. Se nota que A_{ID} puede crecer indefinidamente, en consecuencia, cada vez que se agote el espacio en memoria se debe realocar al doble de su tamaño, lo que hace que el costo amortizado de agregar un nuevo ID sea constante. Además, el no tener esquemas de codificación y decodificación permiten una rápida implementación de `extract`. De este modo, es fácil concluir que $\texttt{extract}(x) = A_{ID}[x].\texttt{extract}(x)$ con lo que la complejidad queda acotada por el tamaño del PFC, es decir, P_{max}. Esto debido a que, dentro del Plain Front Coding, se hace una búsqueda secuencial del identificador dado como argumento y el acceso a la hoja correcta toma un tiempo constante. Se precisa señalar que, para recorrer secuencial-

mente el PFC, se deben reconstruir todas las palabras por las que se pasa, sobrescribiendo el sufijo del string actual s_i a partir de la posición de $\texttt{lcp}(s_{i-1}, s_i)$.

Por último, se debe notar que el universo de identificadores $[1, U]$ es contiguo. Sin embargo, las operaciones de eliminación pueden dejar con *agujeros* este intervalo. Esto ocurre cuando la palabra eliminada del diccionario no tiene un ID igual a U, lo que puede llevar a que U crezca rápidamente cuando, en realidad, pocos valores del intervalo $[1, U]$ están siendo utilizados.

Para solucionar este problema se añadió un puntero P_{first} al primer ID eliminado. De este modo, si se desea insertar una nueva palabra se debe revisar si P_{first} apunta a un espacio del arreglo A_{ID}. Si es así, se añade el nuevo string asociándolo al identificador designado por la posición en A_{ID}, sin modificar el tamaño del universo U. De lo contrario, se expande el intervalo de enteros designados a IDs y se le asigna a la nueva palabra el identificador $U + 1$. Para poder mantener actualizado el puntero P_{first}, se precisa saber dónde está el próximo identificador sin uso, por lo tanto, en todas las casillas de A_{ID} donde no se apunta a un PFC se almacena una referencia al próximo espacio del arreglo donde el ID se encuentre disponible (siguiendo el orden en que fueron liberados). Así, se busca simular una cola de IDs eliminados sin la necesidad de crear otra estructura.

Por otra parte, al retirar una palabra del diccionario se debe actualizar el puntero del último elemento en la cola representada. Considerando que la cantidad de identificadores eliminados puede crecer notoriamente, se adiciona un puntero P_{last} al último elemento de la cola, con el objetivo de evitar recorrer el conjunto de IDs disponibles. Entonces, liberar un identificador x se resume a anular su puntero al PFC que le correspondía, actualizar el arreglo de la forma $A_{ID}[P_{last}] = x$ y, finalmente, modificar el puntero al nuevo último elemento, es decir, $P_{last} = x$.

Tomando las dos estructuras diseñadas, se creó un Diccionario capaz de hacer un mapeo de Strings a IDs y viceversa. La Figura 4.3 muestra un esquema completo, donde se visualiza el árbol de PFCs y el arreglo de identificadores con punteros a las hojas donde se almacenan. Asimismo, se observan los punteros a identificadores eliminados. En el ejemplo de la figura, hay únicamente dos IDs sin ocupar, los números 5 y 7. El puntero P_{first} apunta a 5, cuya casilla en el arreglo almacena una referencia al espacio 7. Luego, el identificador 7 es el último en la cola, de ahí que P_{last} apunte a ese espacio.

4.3 Operaciones de actualización

En esta sección se detalla la implementación de las funciones de actualización usando el diseño del Ring dinámico, en conjunto con el Diccionario basado en Plain Front Coding. Se crearon tres funciones nuevas:

- `insert`(t): inserta el triple t, que representa una arista del grafo, en la base de datos.
- `delete_edge`(t): elimina del grafo la arista representada por el triple t.
- `delete_node`(n): elimina todas las aristas del grafo que tengan el valor n como sujeto u objeto.

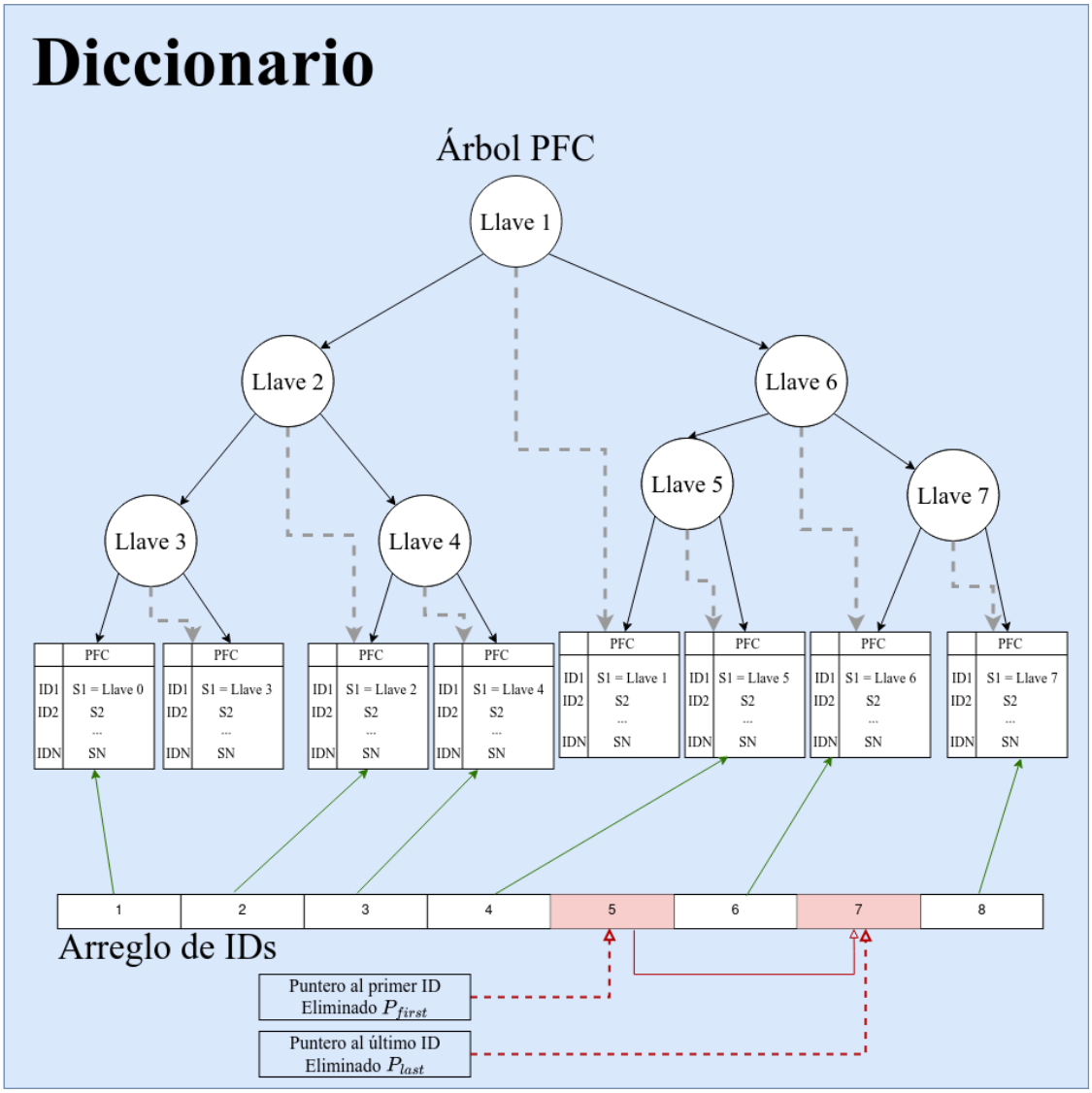


Figura 4.3: Esquema de la arquitectura del diccionario creado. Se aprecia el árbol de PFC para traducción de String a ID, el Arreglo de IDs con punteros a los PFC y el puntero al primer ID eliminado.

Estas funciones se crearon enfocadas en el almacenamiento de grafos y, por lo tanto, para relaciones `SPO` con aridad 3. Además, al ser un grafo, se utiliza un diccionario $\mathcal{D}_{\texttt{SO}}$ para las palabras que definen nodos; y otro diccionario $\mathcal{D}_{\texttt{P}}$ para las aristas. Esto debido a que se asume que los valores de los atributos de los nodos pertenecen a un conjunto distinto de aquellos que definen aristas.

Adicionalmente, se señala que la construcción de las estructuras ya mencionadas se hace a partir de múltiples inserciones, de modo que, al definir la inserción de triples, también se estará detallando el método de construcción.

4.3.1 Inserción de aristas

La incorporación de un triple al índice comienza con la traducción de las palabras del usuario a identificadores utilizados por el Ring. En esta etapa puede ser necesario incorporar nuevos strings al diccionario. Luego, con los IDs obtenidos, se procede a insertar el triple en las tres columnas del Ring y se actualizan los valores de los arreglos A_j.

Inserción en el diccionario

Al traducir un término que forma parte del triple que se está añadiendo, puede que existan nuevas palabras que expandan el conjunto $\mathcal{D}$. Para esto se crea una modificación de la función `locate`, llamada `get_or_insert`. Este método lleva a cabo la búsqueda del string a traducir de la misma forma que `locate`. Sin embargo, dentro del PFC tiene la posibilidad de encontrar o no la palabra buscada. Si la función encuentra almacenada la cadena que se busca traducir, entonces devuelve el identificador asociado a ésta, al igual que lo haría `locate`. De lo contrario, debe insertar la palabra. Para poder bajar por el árbol sólo una vez, siempre se debe descender con un nuevo identificador. En consecuencia, se hace necesario revisar los identificadores eliminados. Si no hay ninguno disponible se usa $U + 1$ como nuevo ID. De lo contrario, se asigna el elemento apuntado por P_{first}. Con el nuevo identificador x_{new}, se recorre el árbol hasta encontrar la hoja donde, por el orden lexicográfico, corresponde encontrar la palabra.

Dentro del Plain Front Coding, se decodifican las palabras secuencialmente hasta que se encuentre el string buscado s, la cadena actual sea mayor que s o se termine el PFC. En los últimos dos casos se debe insertar una nueva palabra en el string del PFC. Como se está en la posición donde debería ir la nueva palabra, no es necesario decodificar nuevamente la estructura para encontrar donde incorporarla.

Si se desea incluir el string s en la posición i, entonces se debe calcular

$$lcp_{i-1} = \texttt{lcp}(s, s_{i-1})$$
$$lcp_i = \texttt{lcp}(s, s_i)$$

Con estos valores se debe modificar el string s_i para incorporar el nuevo valor de lcp_i. Por lo tanto, en la posición siguiente al símbolo terminal de s_{i-1}, se inserta la cadena

$$lcp_{i-1}|x_{new}|\texttt{suffix}(s, lcp_{i-1})\$|lcp_i|x_i|\texttt{suffix}(s_i, lcp_i)\$$$

Donde lcp_{i-1}, x_{new} y lcp_i se encuentran en su forma codificada como se explicó en la sección anterior. Es necesario resaltar que existen casos borde donde no exista s_{i-1} (se añade al inicio del PFC) o no exista s_i (se inserta al final). En el primero de los casos, es fácil ver que la cadena que se debe incluir es la misma, pero sin lcp_{i-1}. En el segundo caso, simplemente se omite la segunda mitad de la cadena agregada ya que no existe s_i.

Luego de la inserción de la nueva palabra puede ocurrir que la cantidad de entradas almacenadas en el PFC supere P_{max}. En este caso, se debe dividir la hoja en dos mediante una operación de `split`. Para hacer esto se recorre el Plain Front Coding hasta haber leído la mitad de las palabras; luego, se divide el string que conforma el PFC en dos mitades: la primera mitad se queda en la misma hoja y la segunda mitad pasa a ser su hoja hermana. Paralelamente, se crea un nuevo nodo padre de las dos mitades. A continuación, se actualizan los valores de A_{ID} de los identificadores almacenados en la nueva hoja derecha para apuntar al PFC creado. Finalmente, se retorna el puntero de la hoja en la que ha sido insertada la nueva palabra, para así actualizar el arreglo de identificadores, asignando $A_{ID}[x_{new}]$ al PFC actualizado. Esto se ve ilustrado en la Figura 4.4.

Es preciso señalar que no se vuelven a calcular los `lcp` ni se cambian las codificaciones de los strings almacenados, únicamente se pasa la primera palabra del nuevo PFC derecho a su forma plana, ya que se transforma en la cabecera de esa hoja. También, resulta pertinente destacar que $P_{max} \geq 2P_{min}$ para que, al realizar la división de la hoja, se cumpla la restricción establecida para la cantidad de palabras en ambos PFCs creados. Luego de esta operación, podría balancearse el árbol para garantizar $O(\log n)$ en tiempos de búsqueda, pero por simplicidad se optó por omitir esta implementación en la presente memoria.

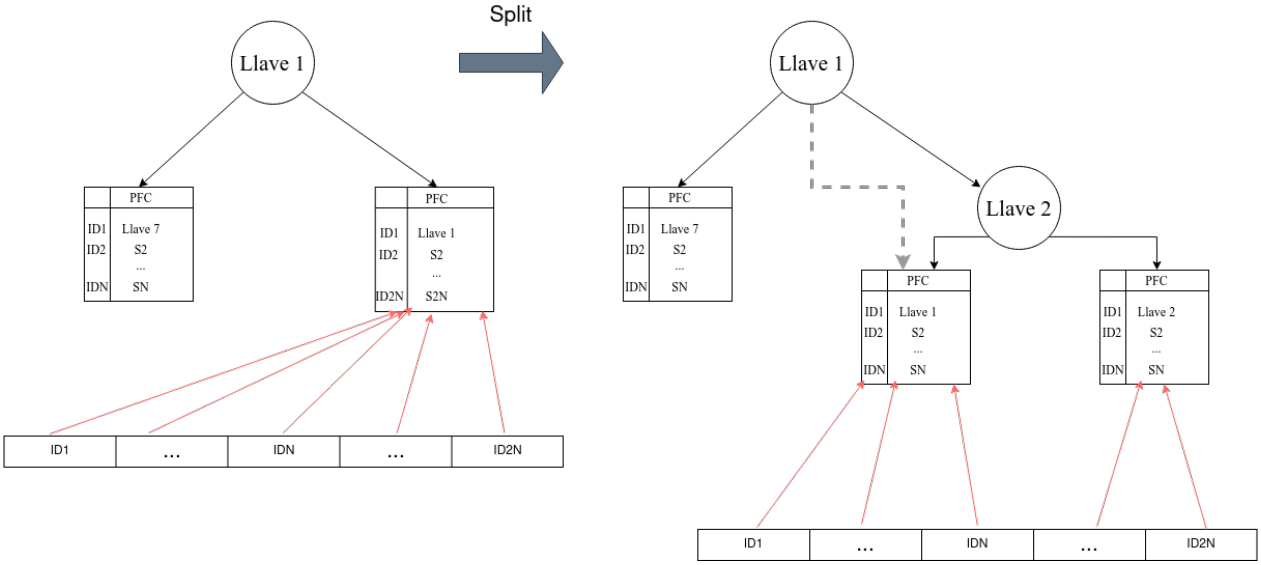


Figura 4.4: Representación de un Split para pasar de un PFC con $2N$ palabras a dos hojas con N palabras cada una.

Inserción en el Ring

Ya obtenidos los IDs (`s,p,o`) que identifican el triple, es necesario encontrar dónde se debe insertar. Para esto se comienza con un intervalo $[a, b]$, el cual se va acotando en cada relación hasta que su tamaño sea 0. Luego, se inserta en el índice a dentro de la columna C_j en la que se encuentra. A continuación, se debe movilizar el índice a otra columna. Esto se puede hacer utilizando la función F_j (definida en la ecuación 2.1). De este modo, se puede encontrar dónde introducir los nuevos valores para cada *wavelet matrix*.

En la implementación actual, se comienza con el orden `POS`, donde se busca cuántos triples tienen el valor `p`. Esto implica que se inicializa $[a, b] = [A_p[\texttt{p}], A_p[\texttt{p}+1]]$. Si el tamaño es mayor a 0, se traduce el intervalo usando F_s, haciendo que $[a, b] = [F_s(a), F_s(b)]$. Si el intervalo sigue teniendo un tamaño mayor a cero, se traduce una última vez usando F_o. Luego, si el triple no estaba en el grafo, el tamaño del intervalo debería ser cero, de lo contrario, no se inserta nada considerando que el Ring no acepta aristas repetidas.

Al mismo tiempo que se inserta cada valor en su columna C_j, se suma uno a su valor en A_j. Tomando el ejemplo de `s`, lo anterior equivale a insertar un 0 en la posición $\texttt{select}(A_s, \texttt{s})$.

La inserción en los *bitvectors* es equivalente a una inserción en un B-Tree convencional, donde al llegar a la hoja se inserta el bit correspondiente. Si la cantidad de bits almacenados en la hoja supera $2\texttt{B}_{leaf}$, entonces se debe dividir en dos, creando un nuevo nodo.

Por otra parte, se observa que el Ring almacena únicamente identificadores de un universo finito $[1, U]$. Por esta razón, las *wavelet matrices* almacenan enteros positivos no negativos donde cada nivel de la matriz tiene un bit del valor guardado, en orden de más a menos significativo. Para ejemplificar, si se baja por la estructura desde la raíz a una hoja y se leen los bits `1, 1, 0, 0` (en ese orden), entonces se tiene el número 12 ya que su representación en bits es `1100`.

De allí que, para insertar en la matriz un número x en la posición i, se debe leer el número desde el bit más significativo. En el primer nivel se inserta en i. Luego, utilizando `rank`, se mueve el índice i para ubicarse en la posición correcta del siguiente nivel de la matriz. También, si x necesita más bits para ser representado que la cantidad de niveles existentes en la matriz, se debe crear nu nuevo nivel con todos sus bits inicializados en 0. Luego, se puede proceder a insertar normalmente el nuevo número x. Lo anterior implica que el tiempo de inserción es amortizado, en vista de que se puede necesitar agregar un nuevo *bitmap*.

4.3.2 Eliminación de aristas

Al igual que en el caso anterior, existe una primera etapa de traducción en el diccionario, para luego eliminar el triple de las columnas y arreglos que componen al Ring. Sin embargo, a diferencia del caso anterior, el diccionario no sabe si los términos del triple se usarán nuevamente, por lo que el Ring debe realizar un chequeo de la cantidad de elementos que tiene cada identificador. Por consiguiente, se debe volver al diccionario para eliminar aquellos términos que ya no existen en la base de datos.

Eliminación en el diccionario

La eliminación en el esquema del diccionario es similar a la inserción. Para eliminar un par ID-string (x, s), el primer paso consiste en actualizar el arreglo de IDs, de modo que $A_{ID}[x] = \perp$, con $\perp$ el puntero nulo. A continuación, se actualiza P_{last} para apuntar a x. Además, si P_{first} no apuntaba a ninguna posición de A_{ID}, entonces se asigna $P_{first} = x$. Finalmente, se debe eliminar el string s del árbol de PFC.

Si se comienza únicamente con la palabra, se debe descender por el árbol hasta encontrar la hoja correcta. De lo contrario, se posee el ID x, por lo que no es necesario buscar s por el árbol, sino que se puede usar el arreglo A_{ID}, tal y como lo hace `extract`. Al encontrar s se debe eliminar del Plain Front Coding. Para realizar esta acción, es necesario obtener los largos de los prefijos asociados a $s_i = s$, es decir, $\texttt{lcp}_i = \texttt{lcp}(s_{i-1}, s_i)$ y $\texttt{lcp}_{i+1} = \texttt{lcp}(s_i, s_{i+1})$. Se nota que el nuevo largo del prefijo entre s_{i-1} y s_{i+1} es el mínimo entre $\texttt{lcp}_i$ y $\texttt{lcp}_{i+1}$. Por lo tanto, se define $lcp_{new} = \texttt{min}(\texttt{lcp}_i, \texttt{lcp}_{i+1})$. Así se puede reemplazar la entrada de s_{i+1} por

$$x_{i+1}|lcp_{new}|\texttt{suffix}(s_{i+1}, lcp_{new})\$$$

Nuevamente existen casos borde que se deben analizar. Si se elimina del final del PFC, simplemente se suprime la entrada de s ya que no afecta al resto de strings. Si, por el contrario, s se encuentra en el principio del Plain Front Coding, entonces se descarta la primera entrada y se decodifica la segunda. De este modo, se deja como cabecera lo que antes era la segunda entrada.

Por último, se revisa la cantidad de palabras P que tiene la hoja. Si se cumple que $P < P_{min}$ se deben fusionar ambas hojas hermanas. Esto se logra mediante una función `fuse`. En primer lugar, se actualizan todos los IDs de A_{ID} que apuntaban a la hoja derecha para que ahora apunten a la hoja izquierda. En segundo lugar, se llama al método `fuse` desde el hijo izquierdo. Esta función recibe el string del hijo derecho, sobre el cual se codifica la primera entrada y se unen los strings de ambas hojas. Después de fusionadas, se elimina la hoja derecha y el nodo padre se convierte en una hoja. Esto se ilustra en la Figura 4.5.

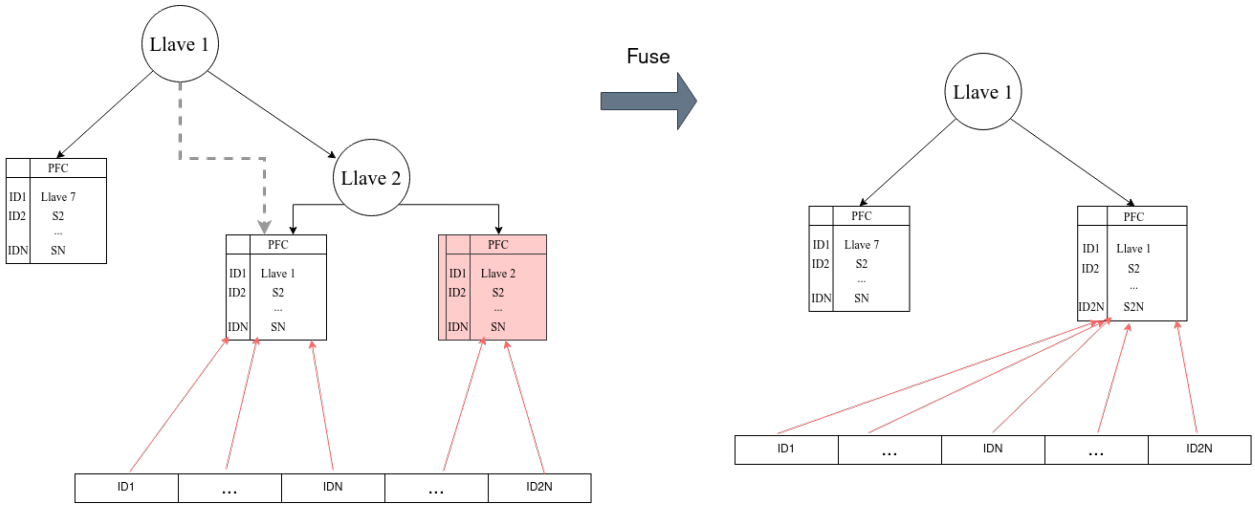


Figura 4.5: Representación de la fusión de dos hojas para pasar de dos PFCs con N palabras a sólo uno con N entradas. El PFC marcado en rojo es el que se elimina.

Eliminación de una arista en el Ring

La eliminación de un triple (`s, p, o`) en el Ring es análoga a la inserción. Se tiene un intervalo $[a, b]$, el cual se va restringiendo a medida que se cruzan las distintas columnas. Cuando el tamaño del intervalo es 1, significa que se encontró el triple. Por lo tanto, se lleva

a cabo la eliminación. Se reitera que no existe el caso en el que se atraviesan las tres columnas C_s, C_p, C_o y $|[a, b]| > 1$, puesto que significaría que se almacena un triple repetido, lo que no es soportado por el Ring.

Al igual que en la inserción, se comienza con C_s, de ahí que el intervalo se inicialice como $[a, b] = [A_p[\mathtt{p}], A_p[\mathtt{p}+1]]$. Luego, se traduce usando F_o y, finalmente, se usa F_p para llegar al último intervalo. Al encontrar el índice a eliminar, se quitan los valores de las *wavelet matrices* y se actualizan los *bitvectors* A_j.

Para actualizar los arreglos, basta con quitar un cero en la posición $\mathtt{select}_1(j+1)-1$. La eliminación del bit en el `succint_bitvector` funciona de la misma forma que en un B-Tree convencional, fusionando las hojas si la cantidad de bits almacenados es menor que $\mathtt{B}_{leaf}$.

Para eliminar en la columna un número x en la posición i, se debe leer el número desde el bit más significativo. En el primer nivel se elimina el bit en i. Luego, utilizando `rank`, se mueve el índice i para ubicarse en la posición correcta del siguiente nivel de la matriz. A medida que se baja por los niveles de la matriz, se actualizan los valores pre-calculados como el $\mathtt{select}_0$ del nivel anterior.

Luego de acabada la eliminación, se debe revisar si es que alguno de los valores `s, p` u `o` ya no tiene ocurrencias en el Ring. Esto se puede realizar rápidamente utilizando los arreglos A_j, ya que la cantidad de veces que se encuentra x en la columna C_j es $A_j[x+1] - A_j[x]$. Para ejemplificar, la cantidad de veces que se repite `p` es $A_p[\mathtt{p}+1] - A_p[\mathtt{p}]$. Sin embargo, para el caso de `s` y `o` se debe revisar tanto en A_s como en A_o. Esto se debe a que los sujetos y objetos comparten diccionario. En resumen, para saber si ya no hay ocurrencias del valor `s`, se debe calcular $A_s[\mathtt{s}+1] - A_s[\mathtt{s}]$ y $A_o[\mathtt{s}+1] - A_o[\mathtt{s}]$. Si ambos valores son 0, entonces se debe borrar del diccionario. De este modo, el Ring le retorna tres booleanos al diccionario para comunicar qué IDs debe borrar.

4.3.3 Eliminación de nodos

La eliminación de nodos corresponde a una fusión entre una consulta de *join* y la eliminación de aristas. En primer lugar, el usuario da como entrada el valor n del nodo a descartar. Sabemos que este valor se debe suprimir de $\mathcal{D}_{SO}$, por lo que se realiza una eliminación tal y como se explicó en la sección anterior. Luego se retorna el ID de n.

A continuación, el diccionario entrega el identificador x del nodo n al Ring. Ahora se buscan todos los triples (`s, p, o`), tales que $\mathtt{s} = n \vee \mathtt{o} = n$. Se comienza restringiendo por $\mathtt{s} = n$, tomando el intervalo $[A_s[\mathtt{s}], A_s[\mathtt{s}+1]]$. Luego, por cada triple en este rango, se elimina de la columna C_o y se propaga al resto de columnas. También se actualizan los arreglos A_j, según corresponda. Paralelamente, se revisan los valores que ya no tienen ocurrencias en el Ring. Si alguno de los identificadores ya no se utiliza, se añade el ID a un vector para ser entregado al diccionario correspondiente.

De la misma forma, se repite el proceso restringiendo por $\mathtt{o} = n$, tomando el intervalo $[A_o[\mathtt{o}], A_o[\mathtt{o}+1]]$. Al terminar de procesar y eliminar todos los triples, se entrega el vector con todos los valores de `SO` sin ocurrencias a $\mathcal{D}_{SO}$ para ser eliminados del diccionario. El proceso es análogo para $\mathcal{D}_P$. Es preciso señalar que, debido a que los vectores contienen identificadores,

no es necesario bajar por el árbol de PFCs para eliminar los valores. Esto se explica porque es equivalente a la operación `extract` y, por esa causa, se puede usar el arreglo de IDs para llegar al PFC correcto.

4.4 Código e implementación

Todo lo mencionado fue implementado usando C++11. El código del Ring dinámico se puede encontrar en `https://github.com/yuval-linker/Ring`. Para poder usar las estructuras compactas estáticas, es necesaria la instalación de SDSL. El repositorio usado para esto se encuentra en `https://github.com/yuval-linker/sdsl-lite`. También se hizo un *fork* del repositorio DYNAMIC de Nicola Prezza, donde se modificaron las estructuras dinámicas. Este código está disponible en `https://github.com/yuval-linker/DYNAMIC`.

Capítulo 5

Validación

En esta sección se presentan los resultados obtenidos para los distintos experimentos diseñados a lo largo de la memoria. En primer lugar, se diseñaron experimentos para comparar y analizar la forma en que afectan los parámetros `B` y $\texttt{B}_{leaf}$ al Ring dinámico. De esta manera, se pudo llegar a una elección informada de sus valores. Así, se llevaron a cabo experimentos que comparan la implementación del Ring dinámico con alternativas del estado del arte.

Todos los experimentos se realizaron sobre subgrafos de Wikidata [39]. Estos grafos Web presentan diversos patrones y su diccionario se acomoda al caso de uso para el que se diseñó el Ring. Junto con esto, se crearon conjuntos de consultas para la actualización de la base de datos, con el objetivo de hacer una evaluación comparativa de las nuevas operaciones disponibles.

Los experimentos sobre los parámetros del Ring dinámico se llevaron a cabo en un computador con una CPU Intel(R) Core(TM) i5-9300H a 2.40GHz, con 4 núcleos, 8MB de caché y 14GB de RAM. Por otro lado, los experimentos de comparación con otras bases de datos fueron realizados en un servidor con un procesador Intel(R) Xeon(R) Silver 4110 a 2.10GHz, 8 núcleos, 12MB de caché y 820GB de memoria RAM. Esto se debió a que el tamaño del dataset usado era demasiado grande para poder soportar la construcción de todos los índices en el computador original.

5.1 Experimentos para escoger `B` y $\texttt{B}_{leaf}$

Se realizaron experimentos para comparar el rendimiento del modelo diseñado en función de los parámetros `B` y $\texttt{B}_{leaf}$. Estos experimentos abarcaron todos los aspectos del Ring dinámico, incluyendo su tamaño, tiempo de consulta, inserción, eliminación de aristas y eliminación de nodos. El objetivo principal es analizar los resultados obtenidos para seleccionar de manera informada los mejores parámetros, los cuales se compararán con bases de datos del estado del arte.

En una primera etapa, se utilizó como *dataset* un subgrafo de wikidata con un tamaño de 2.868.176 triples. Así, se podía construir el índice de manera más expedita y no era

necesaria demasiada memoria RAM. Por consiguiente, se podía variar sobre un rango mayor de parámetros. Cabe destacar que sólo se variaron las *wavelet matrices* en el Ring, dejando siempre los arreglos A_j como `succint_bitvectors`. Se empleó `B` y $\texttt{B}_{leaf}$ como potencias de 2. Se varió `B` desde 2^4 hasta 2^{10}, mientras que se aplicaron valores de $\texttt{B}_{leaf}$ entre 2^8 y 2^{13}.

Cada versión del Ring dinámico se construyó a partir de estos datos. Los experimentos consistieron en medir el tamaño del índice, luego se realizaron 10 *queries joins* distintas. Para probar los tiempos de actualización se tomaron 100 triples aleatorios, los que se eliminaron y luego se volvieron a insertar. Finalmente, se escogieron 10 valores pertenecientes al conjunto de sujetos y se eliminaron los nodos asociados a estos valores. Los resultados presentados promedian los tiempos obtenidos para cada par de parámetros. La razón por la que no se hicieran grandes cantidades de operaciones durante estos experimentos proviene de que el objetivo consistía en únicamente observar el impacto de los parámetros para obtener un diseño inicial.

5.1.1 Resultados y análisis

En la Figura 5.1, se observan los resultados en cuanto a espacio usado por el índice. Se destaca que el principal factor del crecimiento o decrecimiento de espacio usado es $\texttt{B}_{leaf}$. Esto se explica porque, entre menor sea este parámetro, mayor será el número de hojas en el árbol y, por lo tanto, la cantidad de nodos también crecerá. Esto causa un mayor tamaño en la estructura. Por otro lado, `B` afecta únicamente a la altura, por esta razón no incide notablemente en este ámbito. En resumen, para una mejor compresión se requerirá aumentar $\texttt{B}_{leaf}$.

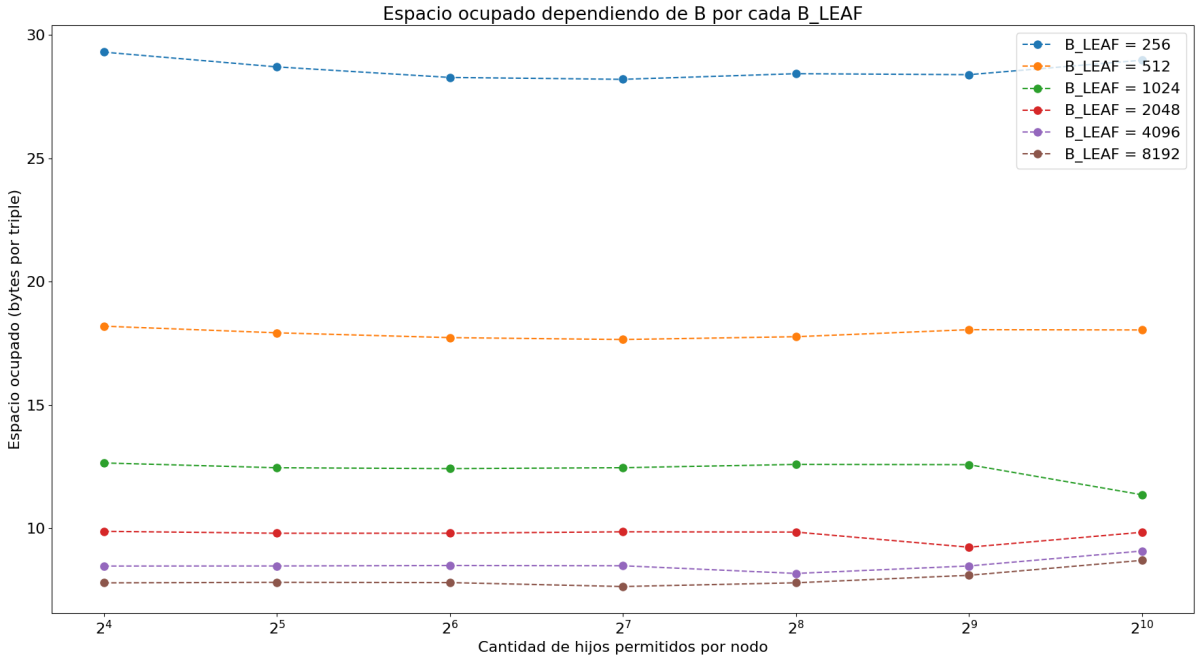


Figura 5.1: Espacio utilizado por el Ring dinámico dependiendo de `B` y $\texttt{B}_{leaf}$.

En cuanto a las consultas *join*, se obtuvieron los resultados presentados en la Figura 5.2. De los tiempos medidos se puede concluir que, en términos generales, el tiempo del *join* en

el Ring aumenta a medida que crece B, mientras que existe un óptimo alrededor de $\mathtt{B}_{leaf}$, el cual depende de la cantidad de hijos permitidos por nodo.

Los resultados se condicen con la estructura de árbol de `spsi`. En efecto, la operación más importante para el *join*, `rank` y `select`, requieren bajar el B-Tree. Entonces, permitir una menor cantidad de hijos por cada nodo implica que hay un menor costo en encontrar el hijo correcto por el cual bajar. Adicionalmente, *bitvectors* más grandes disminuyen el número de hojas disponibles y, por ello, hay una menor altura en el árbol. Sin embargo, se debe señalar que el tamaño de los *bitvectors* afecta a los tiempos de las operaciones `rank` y `select` que se apliquen sobre ellos. Por esta razón, no es conveniente aumentar sin restricción el valor de $\mathtt{B}_{leaf}$. Por lo tanto, existe un valor óptimo para $\mathtt{B}_{leaf}$ con el que se logra un equilibrio entre el tiempo que toma descender por el árbol y las operaciones realizadas en las hojas.

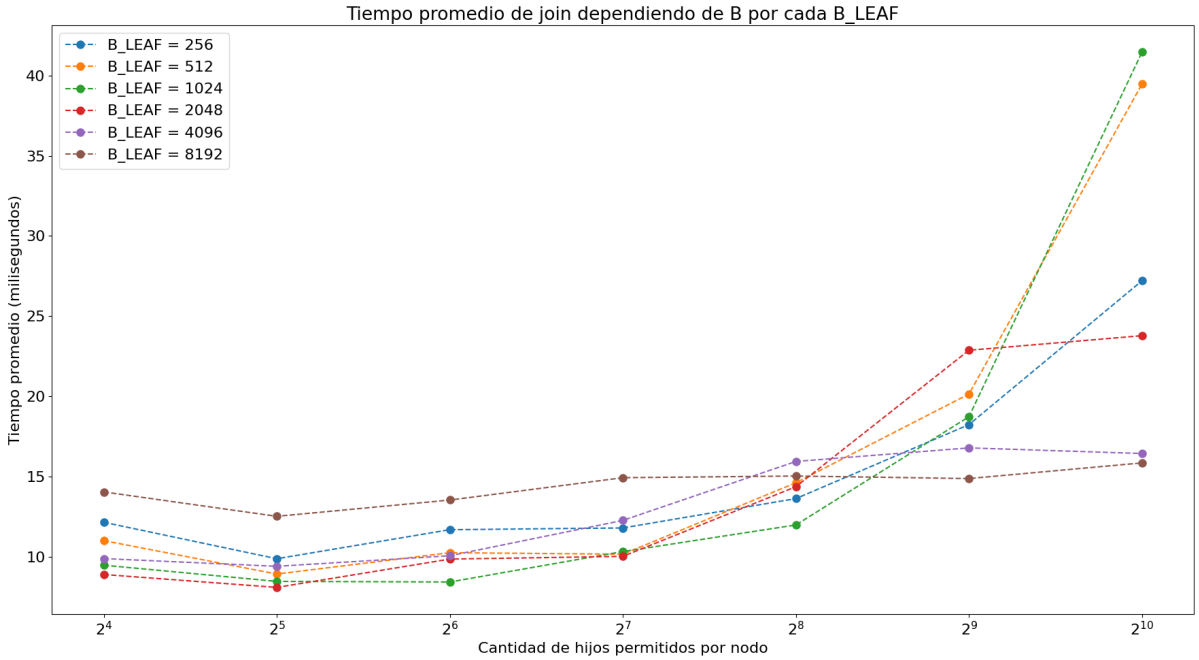


Figura 5.2: Tiempo promedio utilizado por el Ring dinámico dependiendo de B y $\mathtt{B}_{leaf}$ para resolver un join.

En relación a las operaciones de actualización, la inserción y eliminación son prácticamente simétricas. Esto es esperado debido a que ambas requieren de bajar por el B-Tree para finalmente actualizar una hoja. Adicionalmente, ambos procedimientos tienen un caso borde cuando se deja de cumplir la restricción $\mathtt{B}_{leaf} < \mathcal{L} < 3\mathtt{B}_{leaf}$. Este último caso implica la división o fusión de hijos del árbol, lo que añade un procesamiento extra a la actualización. Tomando en cuenta lo anterior y el hecho de que los parámetros elegidos inciden en la frecuencia con la que se debe dividir o fusionar una hoja, se pueden explicar los resultados de las Figuras 5.3 y 5.4.

En primer lugar, cabe mencionar que los tiempos presentados varían en gran manera según cuántas divisiones y fusiones de nodos y hojas fueron necesarias para completar la actualización. Entonces, al tener valores mayores para $\mathtt{B}_{leaf}$ se disminuirá la frecuencia con que se requiere volver a construir una hoja, aunque cuando se deba hacer será una operación más costosa. Por el contrario, al tener un valor pequeño para este parámetro se tendrá una

mayor facilidad para reconstruir un *bitvector* pero se deberá realizar más seguido. En la eliminación de nodos resulta más notorio el efecto de estos casos borde. Por lo tanto, existe un óptimo dependiente de B para la cantidad de bits permitidos en una hoja. Un valor que tiende a tener buenos resultados transversalmente es $\mathtt{B}_{leaf} = 2048$.

En segundo lugar, los resultados de actualizaciones empeoran en cuánto mayor es B, al igual que las consultas *join*. Sin embargo, para el caso $\mathtt{B} = 16$, se detectó una pequeña elevación en los tiempos. La causa de este fenómeno puede recaer en que, al tener una cantidad reducida de hijos permitidos por nodo, aumenta la altura del árbol afectando el costo de descender hacia la hoja correcta.

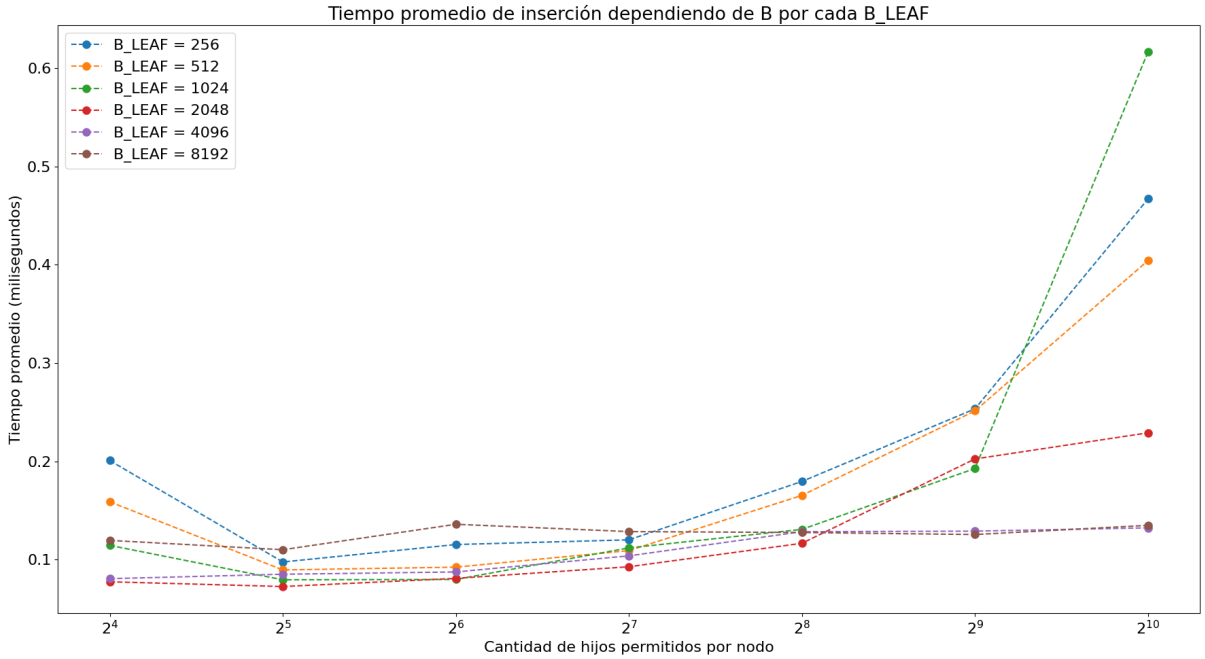
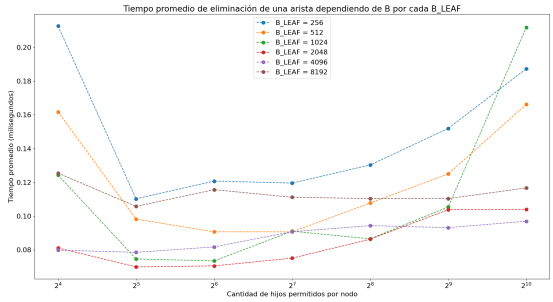


Figura 5.3: Tiempo promedio de inserción luego de añadir 100 aristas en el Ring dinámico dependiendo de B y $\mathtt{B}_{leaf}$.

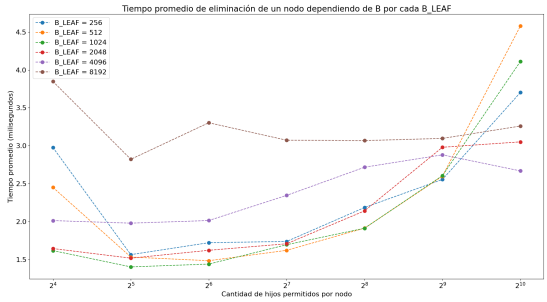
Por último, es importante añadir que los tiempos de inserción y eliminación se presentan en los mismo rangos, lo que es esperado por el hecho de que son operaciones simétricas. Sin embargo, al aumentar B, añadir un triple al Ring presenta promedios de tiempo superiores que el borrado. Nuevamente, se puede explicar con la cantidad de divisiones de hojas que se debieron realizar durante las inserciones para estos casos. Especialmente observando que los valores mayores de $\mathtt{B}_{leaf}$ obtuvieron menores tiempos, en consideración de que estos valores presentan menos casos en los que se rompe la restricción impuesta.

En función de los resultados obtenidos y el análisis reportado, se llegó a la conclusión de utilizar el menor B posible. La causa de esto es que permitir una pequeña cantidad de hijos por nodo no resulta en grandes repercusiones sobre el espacio utilizado, mientras que sí resulta beneficioso para todas las operaciones. Sin embargo, se pudo notar que existía un pequeño incremento de los tiempos tanto en consultas como actualizaciones cuando $\mathtt{B} = 16$, por lo tanto, se decidió utilizar $\mathtt{B} = 32$ ya que consigue los mejores tiempos.

En cambio, la elección de $\mathtt{B}_{leaf}$ requiere un poco más de análisis. Si bien, una gran cantidad



(a) Promedio de eliminación de 100 aristas



(b) Promedio de eliminación de 10 nodos

Figura 5.4: Tiempos de las operaciones de eliminación, variando $\mathtt{B}$ y $\mathtt{B}_{leaf}$.

de bits en las hojas del árbol resulta en una muy buena compresión, se decidió optimizar $\mathtt{B}_{leaf}$ en función del tiempo debido a los buenos resultados en espacio utilizado. En consecuencia, se decidió utilizar $\mathtt{B}_{leaf} = 2048$, ya que tiene tiempos de consultas y actualización competitivos, mientras que no afecta en gran medida el espacio utilizado. Por lo tanto, para los experimentos de comparación con el estado del arte, se optó por utilizar $\mathtt{B} = 32$ y $\mathtt{B}_{leaf} = 2048$.

5.1.2 Pruebas en el dataset completo

Dado que los experimentos anteriores construyeron una intuición inicial para el diseño del Ring dinámico, se decidió confirmar la decisión utilizando el dataset de Wikidata Graph Pattern Benchmark (WGBP) [20], el cual utiliza un subgrafo de Wikidata con $n = 82.923.234$ triples, 16.369.027 sujetos, 2.101 predicados y 37.883.206 objetos. Esta prueba incluye 850 *join queries*, separadas en subconjuntos de 50 para cada patrón presente en el grafo. Los patrones disponibles se ilustran en la Figura 5.5. Todas las consultas se ejecutaron con un *timeout* de 10 minutos y un límite de 1000 resultados.

Adicionalmente, se añadieron consultas de actualización divididas en tres tipos: Inserción, Eliminación de aristas o triples y Eliminación de nodos. Para la inserción y descartado de triples se eligió un subconjunto $\mathcal{E}$ compuesto de aristas al azar del grafo. Se utilizó una cantidad de $|\mathcal{E}| = 1500$ triples, donde se elimina por completo $\mathcal{E}$ del grafo y luego se vuelve a insertar.

Paralelamente, para la eliminación de nodos, se tomaron los 500 valores v con más triples en los cuales se cumpliera que $\mathtt{s} = v$. Dicho de otra manera, se tomaron los 500 nodos con mayor grado saliente. Esto resultó en una lista donde el primer valor tenía, al menos, 8309 triples y el último se conectaba con otros 285 nodos. De este modo fue posible combinar BGPs con actualizaciones en la misma operación.

Bajo estas condiciones, se analizó el espacio utilizado por el índice, tiempo promedio de 850 consultas *join* y los tiempos de las operaciones de actualización.

Para verificar que un $\mathtt{B}$ mayor implica un peor rendimiento se construyó un índice con parámetros $\mathtt{B} = 256$ y $\mathtt{B}_{leaf} = 2048$. Para verificar el incremento en tiempos para $\mathtt{B} = 16$ se construyó un índice con este valor como parámetro y $\mathtt{B}_{leaf} = 2048$. Por otra parte, con el objetivo de analizar el cambio en resultados dependiendo de $\mathtt{B}_{leaf}$, se construyeron 2 instancias

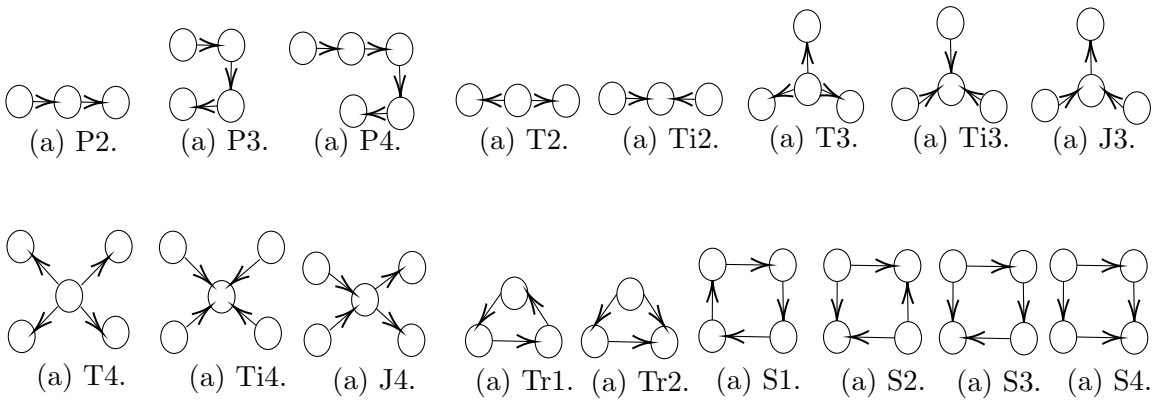


Figura 5.5: Patrones de consulta en la prueba de rendimiento de Wikidata [20].

más del Ring dinámico; una con un valor más pequeño, $\mathtt{B}_{leaf} = 1024$, y otra utilizando los valores de `succint_bitvector`, $\mathtt{B} = 16$ y $\mathtt{B}_{leaf} = 8192$, recomendado en el paper original de DYNAMIC [34].

Los resultados se muestran en la Figura 5.6, donde se presenta el espacio en bytes usados por cada triple y el tiempo promedio de las consultas *join*. Al analizar los valores presentados, se aprecia que al aumentar la cantidad de bits por hoja en el árbol de `spsi`, se obtiene una mejor compresión, sin embargo, hay un deterioro en los tiempos de consulta. Lo anterior proviene del costo extra que se añade al realizar `rank` y `select` en las hojas.

Paralelamente, tomar un valor bajo de $\mathtt{B}_{leaf}$, repercute negativamente en el espacio utilizado pero impactando positivamente el tiempo promedio de las consultas. También, se pudo observar la presencia de una disminución en el rendimiento para $\mathtt{B} = 16$ y se confirmó que el valor óptimo para B es 32. Finalmente, se pudo constatar que la mejora en compresión utilizando un B mayor es demasiado pequeña, en comparación al aumento del tiempo promedio. En conclusión, se pudo validar el análisis de la discusión presentada en la sección anterior.

B	$\mathtt{B}_{leaf}$	Espacio (bytes por triple)	Tiempo promedio (ms)
256	2048	11.78	321.87
32	2048	11.95	263.72
32	1024	15.30	276.99
16	8192	9.43	348.21
16	2048	11.95	296.97

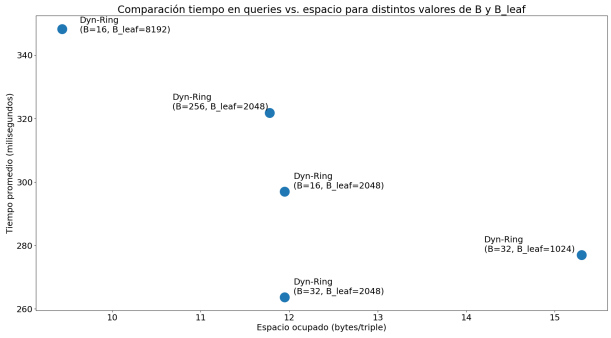


Figura 5.6: Resultados en espacio y tiempo del Ring dinámico para distintos parámetros. Resultados sobre el dataset filtrado de Wikidata.

A esto se añaden los experimentos de actualización del índice. Estos resultados se condicen con lo planteado en la sección anterior. En primer lugar, se pudo confirmar el impacto de B sobre la inserción, demostrado en que los tiempos para $\mathtt{B} = 16$ y $\mathtt{B} = 256$ son mayores a aquellos que presentan los índices que utilizan $\mathtt{B} = 32$. Por otro lado, se pudo observar la baja alteración que provoca $\mathtt{B}_{leaf}$ sobre los resultados, siendo la pareja óptima $\mathtt{B} = 32$ y

$\mathtt{B}_{leaf} = 2048$. Sumado a ello, se logró confirmar el bajo tiempo de inserción que tiene el Ring dinámico. En la Figura 5.7 se aprecian los resultados mencionados.

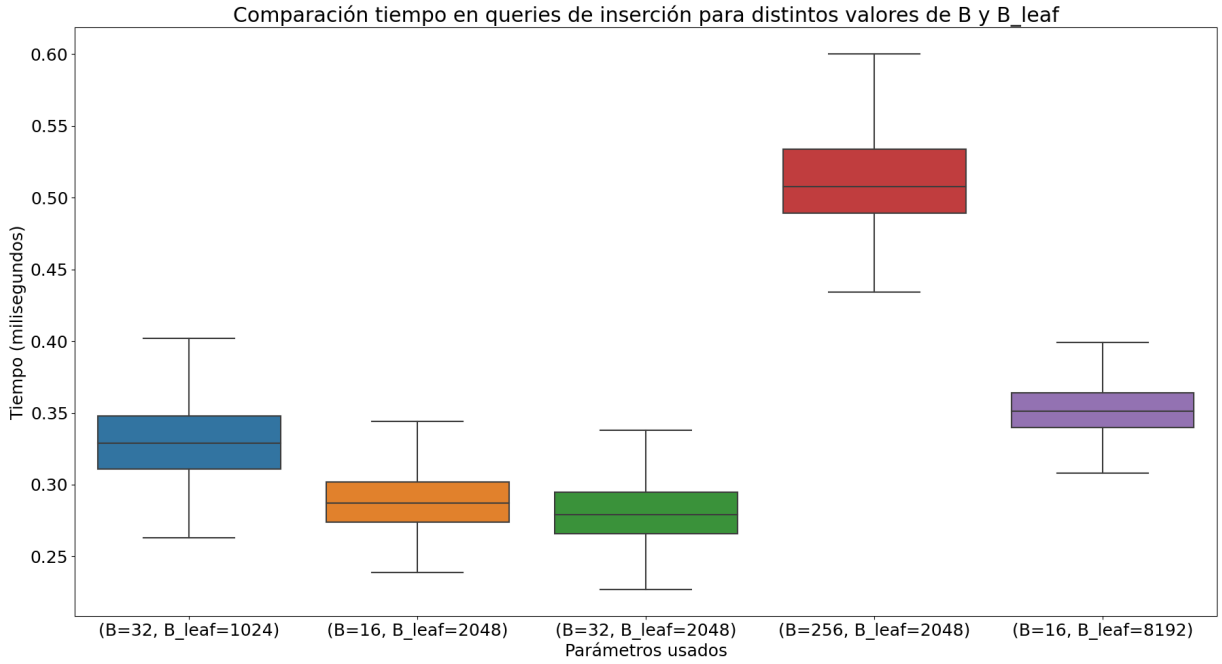


Figura 5.7: Tiempos de inserción, categorizados por $\mathtt{B}$ y $\mathtt{B}_{leaf}$ utilizados. Las cajas van del percentil 25 al 75, con la mediana marcada dentro con una línea. Se removieron los valores atípicos o *outliers*.

En cuanto a las eliminaciones de aristas en el índice, se logró validar el par de parámetros elegido como óptimo para esta operación. Los resultados se pueden observar en la Figura 5.8. Se destaca que $\mathtt{B}_{leaf}$ afecta en la varianza de las actualizaciones. Se observa como los cuartiles se separan en mayor medida en cuanto más pequeño es el tamaño permitido de la hoja. Esto se puede deber a la frecuencia de divisiones necesarias que ocurren en el árbol.

Finalmente, los resultados para la eliminación de nodos, presentados en la Figura 5.9, revelan que un menor valor para $\mathtt{B}_{leaf}$ implica mejores tiempos. Es preciso señalar que, al borrar una cantidad considerable de triples, se está forzando a que existan casos donde se deban fusionar hojas. Esta operación se ralentiza para *bitvectors* muy grandes, reflejado en los tiempos para el índice con $\mathtt{B}_{leaf} = 8192$. A pesar de que el par elegido no obtuvo los mejores tiempos, quedó en segundo lugar y, considerando los otros experimentos analizados, aún se puede concluir que los parámetros escogidos obtienen, en general, los mejores tiempos.

En síntesis, se pudo validar, con un dataset de gran tamaño, el análisis inicial dado para el diseño del Ring dinámico y cómo sus parámetros lo afectan. Así, se reafirmó la decisión de emplear $\mathtt{B} = 32$ y $\mathtt{B}_{leaf} = 2048$ en los experimentos comparativos con otras bases de datos del estado del arte.

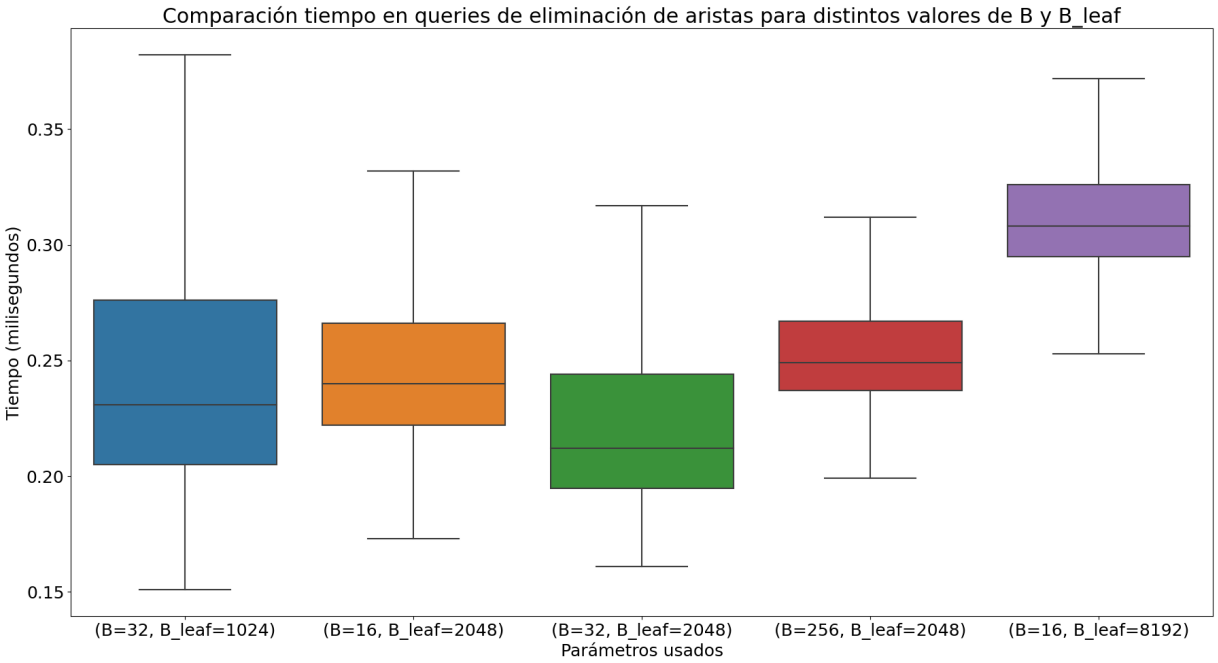


Figura 5.8: Tiempos de eliminación de aristas, categorizados por B y $\mathtt{B}_{leaf}$ utilizados. Las cajas van del percentil 25 al 75, con la mediana marcada dentro con una línea. Se removieron los valores atípicos o *outliers*.

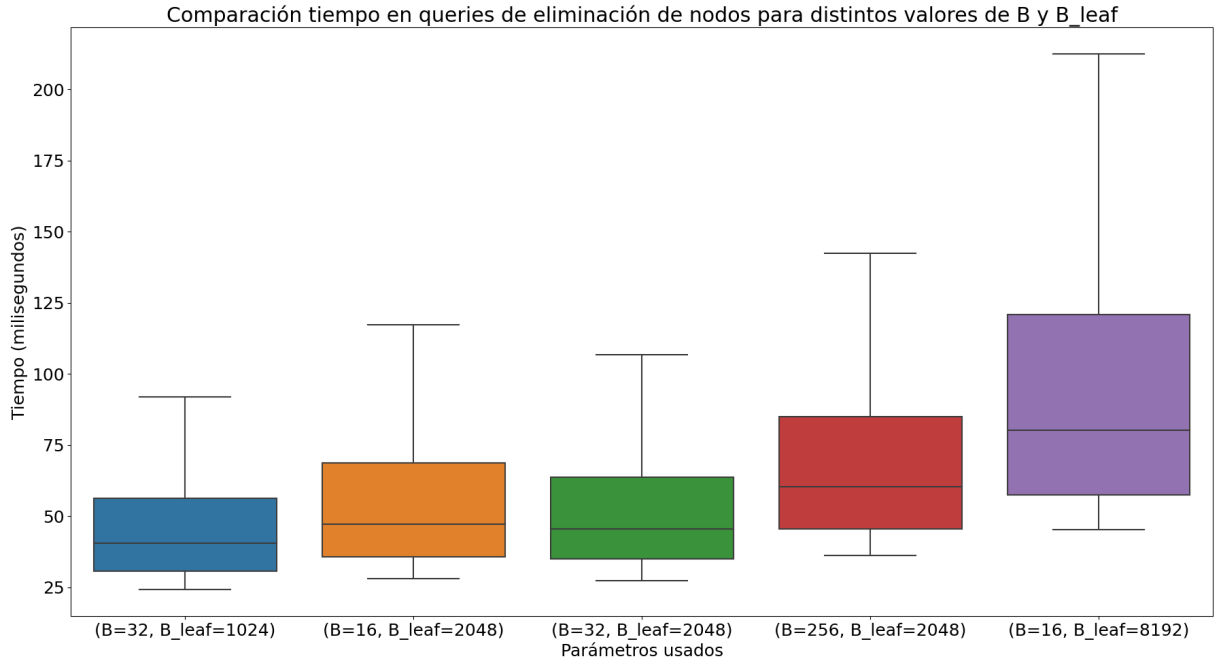


Figura 5.9: Tiempos de eliminación de nodos, categorizados por B y $\mathtt{B}_{leaf}$ utilizados. Las cajas van del percentil 25 al 75, con la media marcada dentro con una línea. Se removieron los valores atípicos o *outliers*.

5.2 Experimentos de comparación con el estado del arte

En esta sección se presentan los resultados donde se compara el sistema diseñado en la memoria con otras alternativas del estado del arte en términos de espacio ocupado al indexar y tiempo de consultas. Se comparan las bases de datos usando el dataset de Wikidata Graph Pattern Benchmark (WGPB) [20]. Como se explicó en la sección anterior, esta prueba de rendimiento presenta un subgrafo de Wikidata [39] con $n = 82.923.234$ triples, 16.369.027 sujetos, 2.101 predicados y 37.883.206 objetos. Sin embargo, hay valores que se usan como sujeto y objeto, por lo que se tiene un tamaño real de diccionarios de $|\mathcal{D}_{SO}| = 54.251.877$ y $|\mathcal{D}_P| = 2.101$. Al igual que en la comparación de parámetros, se realizaron 850 *queries*, 1500 inserciones y eliminaciones de triples y la remoción de 500 nodos. Estos experimentos tienen la misma naturaleza que los realizados en la sección anterior, donde se explicó con mayor detalle la construcción de las consultas.

Se emplearon los siguientes índices en los experimentos:

- **Dynamic-Ring**: El Ring dinámico diseñado en esta memoria. Se utilizaron los parámetros $\mathtt{B} = 16$ y $\mathtt{B}_{leaf} = 2048$ para las *wavelet matrices*, mientras que se aplicaron los valores $P_{min} = 32$ y $P_{max} = 128$ para los diccionarios $\mathcal{D}_{SO}$ y $\mathcal{D}_P$.
- **Ring y C-Ring**: El Ring estático con la implementación del trabajo de Navarro et. al. [4]. C-Ring es la versión comprimida del índice normal, la cual usa compresión en los *bitvectors* de las *wavelet matrices*. Se escogió un valor de $b := 16$ para obtener resultados comparables con el paper original. Estas estructuras son las únicas que no permiten operaciones de actualización.
- **Jena**: Base de datos de grafos RDF que permite almacenar y consultar datos estructurados usando el estándar SPARQL. Se usó la variante TDB para la construcción, con índices B+-Trees para tres órdenes: `spo`, `pos` y `osp`.
- **Jena-LTJ**: Implementación de Leapfrog Triejoin sobre Jena TDB [20]. Se indexaron los seis órdenes usando B+-Trees.
- **RDF-3X**: Esquema que indexa una única tabla de triples en un árbol B+ comprimido [31]. RDF-3X maneja patrones mediante la exploración de rangos de triples y utiliza un optimizador de consultas basado en uniones por pares. A pesar de tener un módulo para actualizar el índice, al intentar utilizarlo, la versión que se tenía resultaba en *segmentation fault* por lo que no fue posible probar las operaciones de dinamismo sobre esta base de datos.
- **Virtuoso**: Base de datos de grafos usada para soportar el *endpoint* de la API pública de DBPedia [15]. Provee un índice por columnas y añade un atributo extra `G`, con el que obtiene dos órdenes completos (`PSOG, POSG`) y tres índices parciales `SO, OP` y `GS`. Esto permite una optimización para patrones con predicados constantes.
- **Blazegraph**: Sistema de base de datos de grafos que aloja el servicio oficial de consultas de Wikidata [35]. Se corre el sistema en modo de triples, es decir, indexando árboles B+ en tres órdenes `spo, osp` y `pos`.

Como se mencionó en el listado, los únicos esquemas que no presentan datos en los experimentos de dinamismo son el Ring, C-Ring y RDF3X, debido a que no permitieron este tipo de operaciones. Es importante añadir que el Ring, en todas sus versiones, funciona en memoria principal, mientras que el resto de alternativas puede usar memoria secundaria. En estas bases de datos, las únicas alternativas que implementan algoritmos WCO son Jena-LTJ y las versiones del Ring.

5.2.1 Indexación y Joins

Se indexaron 82.923.234 triples en cada base de datos. El Ring dinámico tomó 6.26 minutos en construirse, esto es una velocidad de 13.2 millones de triples por minuto. Este valor es mayor al tiempo que necesitó el Ring y C-Ring, los cuales tardaron 3.08 y 3.18 minutos, respectivamente. El aumento al doble del tiempo de construcción se atribuye, principalmente, a que las versiones estáticas se construyen de una sola vez, utilizando esquemas especiales de inicialización. En cambio, el Ring dinámico se crea usando inserciones consecutivas. Este tiempo podría mejorar implementando un esquema de *bulk-loading*.

En una primera etapa, se experimentó usando el Ring dinámico sin la implementación del diccionario. Esto se llevó a cabo con el propósito de detectar el impacto generado por la traducción de las consultas. Estos resultados se encuentran representados en la Figura 5.10, donde el espacio ocupado se mide en cuántos bytes fueron necesarios para representar cada triple y el tiempo es un promedio (en milisegundos) de las 850 consultas que se realizaron. Adicionalmente, se presenta la cantidad de *queries* que ocuparon más de 10 minutos en terminar.

Se puede apreciar que el Ring Dinámico obtiene una muy buena compresión, alcanzando un espacio similar al Ring original. No obstante, el tiempo de consultas se ve drásticamente afectado, llegando a un promedio relativamente cercano al de Jena. Con esto se puede concluir que, a pesar de utilizar un algoritmo WCO, la constante asociada a este orden teórico es muy elevado, dejando al Ring dinámico por detrás de competidores que utilizan algoritmos que cumplen la misma propiedad. A pesar de esto, se obtuvieron mejores resultados que, por ejemplo, Blazegraph motor de base de datos que soporta en la actualidad a Wikidata.

Como opción estática, el Ring sigue siendo una de las mejores alternativas, debido a su bajo tiempo de consulta y poco espacio utilizado. No obstante, es preciso analizar los tiempos de cada patrón utilizado en la prueba de rendimiento. Si se observa la Figura 5.11, se puede apreciar que en la mayoría de los patrones, el Ring dinámico toma rangos de tiempos parecidos al resto de alternativas WCO, como el Ring o Jena-LTJ. En cambio, es en las consultas de tipo Tr1, Tr2, S1, S2, S3 y S4 donde existe un aumento sustancial del tiempo necesitado por el índice diseñado, muy por encima del rango presentado por las alternativas. Esto implica la existencia de varios *outliers* en este tipo de recorridos por el subgrafo.

Es preciso señalar que Tr1, Tr2, S1, S2, S3 y S4 presentan ciclos, patrón con el que el Ring estático también obtiene peores resultados. Analizando el código con herramientas de *profiling*, se descubrió que estas *queries* son las que más utilizan las funciones `range_next_value` y `select_next`. Estos métodos tienen la particularidad de llamar reiteradas veces a `rank` en el `spsi`, representando el nivel correspondiente de la *wavelet matrix*. Luego, al haber un

BD	Espacio (bytes por triple)	Timeouts	Tiempo promedio (ms)
Blazegraph	120.64	2	610.11
Jena	119.01	0	332.16
Jena-LTJ	189.83	0	24.81
RDF3X	105.76	0	51.14
Ring	11.16	0	38.26
C-Ring	6.93	0	129.96
Dynamic-Ring	11.86	0	263.72
Virtuoso	104.40	1	62.99

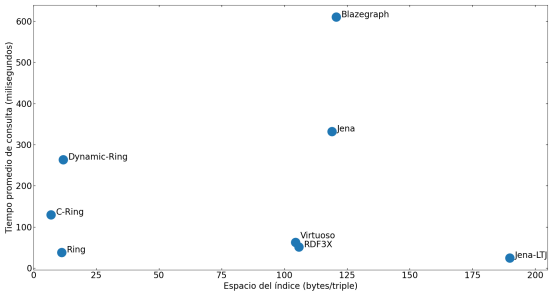


Figura 5.10: Resultados en espacio y tiempo promedio de consultas sobre el subgrafo de Wikidata.

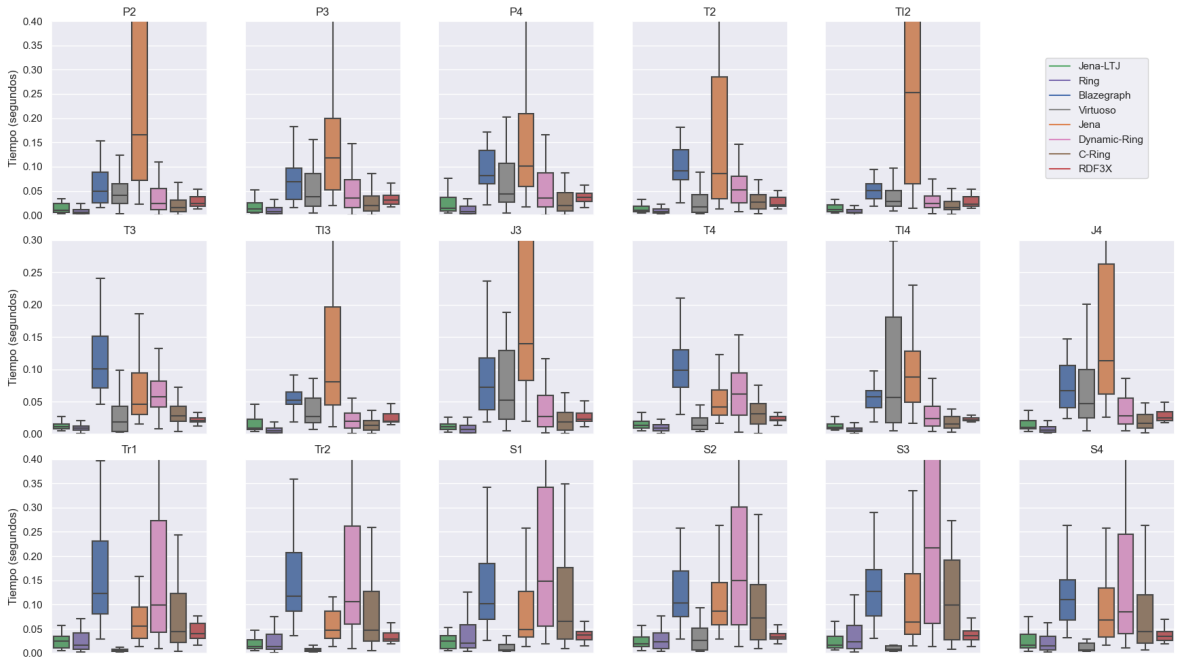


Figura 5.11: Comparación de tiempos de consultas en segundos. Se agrupan los resultados por patrón de *query*. Las cajas van del percentil 25 al 75, con la mediana marcada dentro con una línea. Se removieron los valores atípicos o *outliers*.

decremento en la rapidez de esta operación, se produjo una diferencia en los tiempos de respuesta, el cual se vio acentuado al reutilizar las funciones mencionadas. Para ejemplificar, en las consultas Tr1 se llamó más de 30.000.000 de veces la función `range_next_value`, donde se percibió la mayor cantidad del tiempo ocupado.

Para poder observar de mejor manera la diferencia entre los tiempos del Ring dinámico y estático se tomó el cociente entre ambos tiempos. Esta razón revela cuánto más tarda el Ring dinámico y en qué patrones se acentúa la diferencia. Se evidencia en la Figura 5.12 los resultados obtenidos al dividir el tiempo medido para el Ring dinámico sobre el resultado

del Ring original. Se puede apreciar que los patrones con ciclos obtienen una varianza y cuartiles mucho mayores que el resto de patrones, confirmando la problemática planteada. Adicionalmente, es interesante observar que existen *queries* no cíclicas donde el Ring dinámico tiene un rendimiento alrededor de 6 veces peor que su versión estática, como es el caso para T2, T3 y T4. Aún así, el tiempo de resolución para estos patrones no se diferencia demasiado del resto de alternativas WCO.

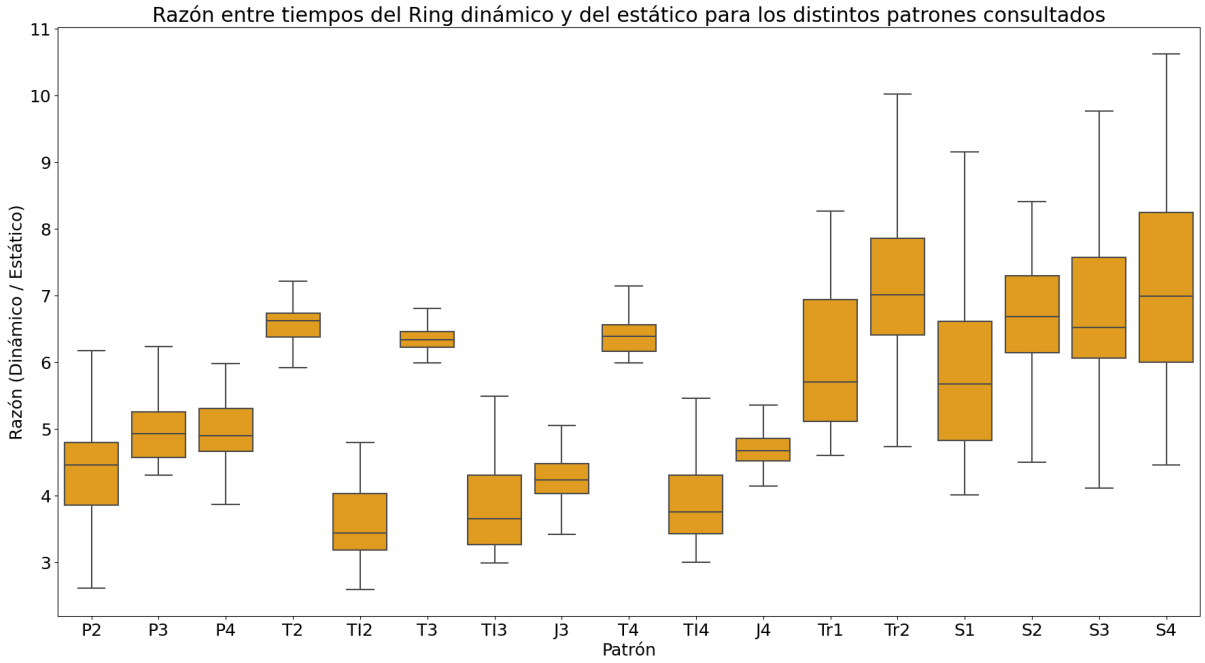


Figura 5.12: Razón entre los tiempos del Ring dinámico y estático. Se agrupan los resultados por patrón de *query*. Las cajas van del percentil 25 al 75, con la mediana marcada dentro con una línea. Se removieron los valores atípicos o *outliers*.

En una segunda serie de experimentos, se añadieron los diccionarios al Ring, C-Ring y Dynamic-Ring. Si bien, existen diccionarios compactos y estáticos que funcionarían mejor para las versiones estáticas del Ring, se utiliza la versión planteada en esta memoria para poder comparar cómo afecta esta estructura a índices con mejores resultados que el Ring dinámico. Además, así se demuestra la independencia del diccionario diseñado frente al índice que se utiliza.

Como se mencionó con anterioridad, se construyeron dos diccionarios $\mathcal{D}_{SO}$ y $\mathcal{D}_P$, ya que sujetos y objetos comparten alfabetos mientras que los predicados tienen sus propios identificadores en el Ring. De este modo, se puede comparar más equitativamente con el resto de alternativas, las cuales utilizan índices a bajo nivel pero aceptan consultas en lenguaje SPARQL.

La construcción de los diccionarios tomó 35.6 minutos, lo que es similar al tiempo de construcción de las alternativas, donde se daban tiempos de 30-50 minutos. Lo anterior posiciona el tiempo de construcción del índice entre 38 y 42 minutos, dependiendo de la versión del Ring utilizada.

Por otra parte, la representación plana de los diccionarios ocupa un tamaño de 1.7 GB

para $\mathcal{D}_{SO}$ y 92.7 KB para $\mathcal{D}_P$. Sin embargo, para realizar una comparación fiel, se debe añadir el espacio ocupado por los identificadores. Si se utilizan enteros de 32-bit, entonces los tamaños quedan como 1.9 GB para SO y 101.2 KB para P. En cambio, el tamaño del diccionario compacto propuesto fue de 1.1 GB para SO y 32.3 KB para P. En otros términos, y utilizando el tamaño de los alfabetos, la versión comprimida y dinámica de $\mathcal{D}_{SO}$ necesitó 20.1 bytes por palabra, mientras que la estructura construida para $\mathcal{D}_P$ utilizó 15.4 bytes por palabra.

Finalmente, la representación plana (en texto) de los 82.923.234 triples almacenados requiere de 9.9 GB. En este contexto, el Ring dinámico, incluyendo la adición de los diccionarios, logró comprimir la información utilizado un 21.2 % del tamaño original, ocupando únicamente 2.1 GB.

Sumado a ello, la aplicación de diccionarios tuvo una repercusión muy baja en los tiempos de consultas y en el espacio ocupado. Estos resultados se encuentran representados en la Figura 5.13, donde se puede observar que, a pesar de aumentar su tamaño, el promedio de tiempos aumentó en una cifra cercana a 1.9 milisegundos. Se debe precisar que se limitan las consultas a tan solo 1000 resultados, por lo tanto, también se están restringiendo las traducciones por *query*. Esto explica el pequeño tiempo de traducción. Una cifra más reveladora es cuánto toma la conversión de ID a String por cada resultado obtenido. Se obtuvo que, en promedio, el diccionario tomó 6 microsegundos en traducir un resultado. Además, los Rings siguen logrando la mejor compresión de todas las alternativas del estado del arte.

BD	Espacio (bytes por triple)	Timeouts	Tiempo promedio (ms)
Blazegraph	120.64	2	610.11
Jena	119.01	0	332.16
Jena-LTJ	189.83	0	24.81
RDF3X	105.76	0	51.14
Ring	29.28	0	40.31
C-Ring	25.06	0	132.02
Dynamic-Ring	29.63	0	265.77
Virtuoso	104.40	1	62.99

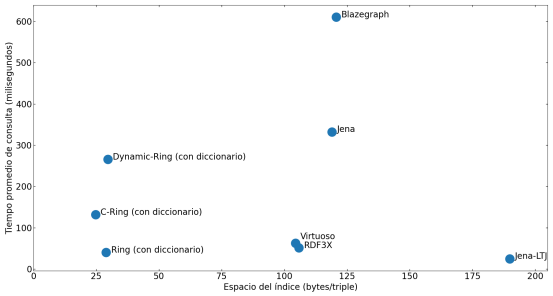


Figura 5.13: Resultados en espacio (bytes/triple) y tiempo promedio de consultas sobre el subgrafo de Wikidata utilizando el Ring con diccionario.

Para representar visualmente el impacto del diccionario en las *queries*, se puede observar la Figura 5.14, en la cual se muestra, en azul, el porcentaje de tiempo promedio usado por el algoritmo del Ring para consultas, en naranjo la proporción de tiempo promedio usado en la traducción de String a ID (traducción hacia adelante) y, por último, en verde, el porcentaje de tiempo promedio usado en la traducción de ID a String (traducción hacia atrás). De esta manera, se puede dar al Ring la facilidad de permitir consultas compuestas con palabras y no sólo con números enteros positivos, sin mermar el rendimiento característico del índice.

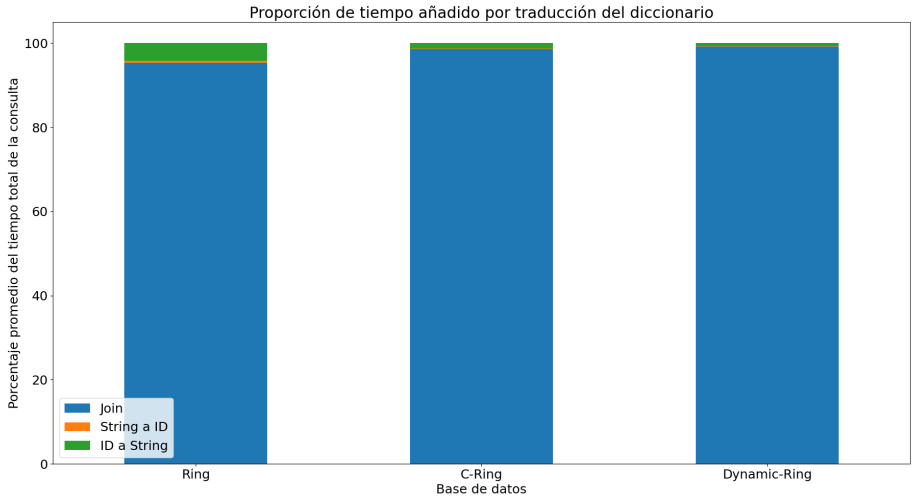


Figura 5.14: Descomposición del tiempo de consultas del Ring con diccionario (en porcentajes).

5.2.2 Dinamismo

En los experimentos de dinamismo se utilizaron todas las bases de datos, excepto el Ring, C-Ring y RDF3X. Además, el Ring dinámico se empleó con el diccionario integrado, ya que forma parte esencial de las operaciones de actualización.

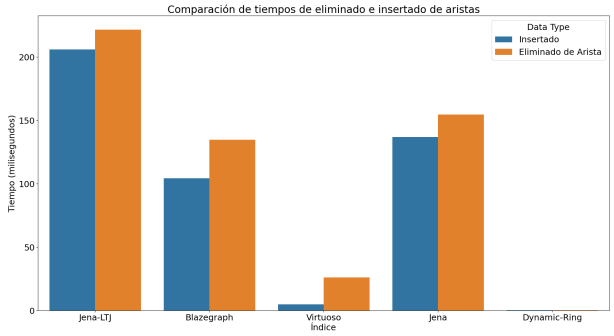
Los resultados se pueden observar en la Tabla 5.1. Ahí se presentan los tiempos promedio de cada operación de actualización en milisegundos. Es fácil notar que el Ring Dinámico obtuvo los mejores tiempos de insertado y borrado de triples, mientras que Virtuoso fue el índice con mejor rendimiento de eliminación de nodos. No obstante, el Ring dinámico obtuvo tiempos competitivos con las alternativas en este aspecto.

BD	Insertado (ms)	Borrado de triple (ms)	Borrado de nodo (ms)
Jena-LTJ	205.83	221.45	2421.66
Blazegraph	104.43	134.64	71660.09
Virtuoso	5.00	26.10	50.89
Jena	136.97	154.71	2840.56
Dynamic-Ring	0.36	0.31	203.39

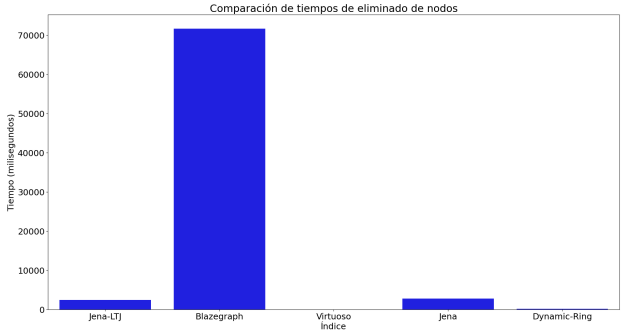
Tabla 5.1: Tiempos promedio de cada operación de actualización (en milisegundos).

Una representación más visual de estos resultados se puede ver en la Figura 5.15. Ahí se pueden observar las diferencias de órdenes que existen en los tiempos donde se puede apreciar la rapidez del Ring dinámico para actualizarse. También es posible apreciar cómo la cantidad de copias de los órdenes de las relaciones afecta la actualización, lo que se evidencia en que Jena LTJ presenta tiempos superiores que Jena. Por lo tanto, se atribuye parte de los buenos resultados del Ring dinámico a tener que actualizar únicamente tres columnas.

También, se realiza un análisis más detallado en la Figura 5.16, desglosando por tipo



(a) Inserción y eliminación de aristas.



(b) Eliminación de nodos.

Figura 5.15: Visualización de la diferencia en los promedios de tiempos de actualización (en milisegundos) para distintas bases de datos.

de operación. Se evidencia en los datos presentados que existe una gran varianza para las operaciones de actualización, independiente del índice que se esté evaluando. Adicionalmente, es evidente que Blazegraph no tiene un buen soporte para realizar una gran cantidad de eliminaciones, obteniendo tiempos mucho mayores que el resto de bases de datos comparadas.

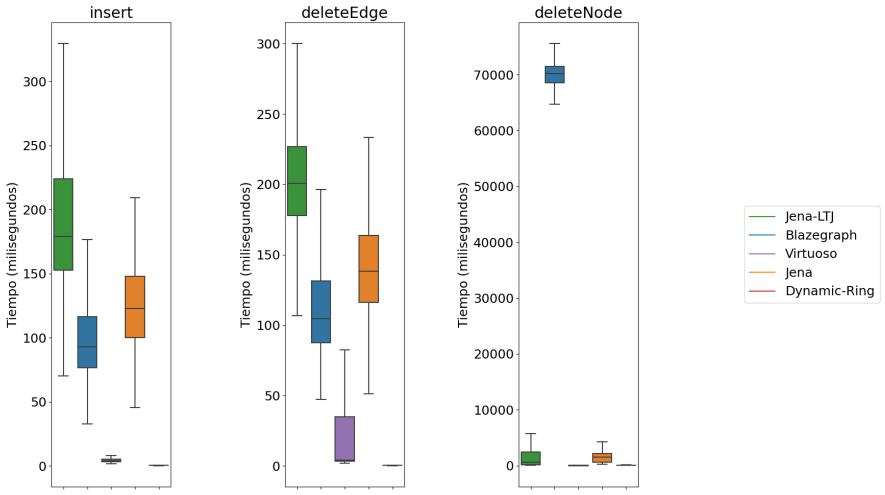


Figura 5.16: Comparación de tiempos de actualizaciones en milisegundos. Se agrupan los resultados por tipo de operación. Las cajas van del percentil 25 al 75, con la mediana marcada dentro con una línea. Se removieron los valores atípicos o *outliers*.

Por último, es necesario destacar el tiempo tomado en la eliminación de nodos en función de la cantidad de triples a suprimir. En estos resultados, presentados en la Figura 5.17, se puede observar que el Ring dinámico tiende a tener muy buenos tiempos, independiente de la cantidad de triples que se deben borrar. Es interesante observar como los resultados de Jena-LTJ tienen la misma forma que los tiempos de Jena pero aumentados por un valor dependiente de la cantidad de triples suprimidos. Con lo anterior, se revela el costo de almacenar una mayor cantidad de copias para las relaciones, en el contexto de la actualización del índice. Cabe mencionar también que Jena-LTJ fue la única base de datos en obtener un *timeout* al intentar eliminar más de 200.000 triples. Por último, Virtuoso obtuvo los mejores tiempos para eliminaciones masivas, a pesar de obtener peores resultados que el Ring dinámico para la remoción individual de triples.

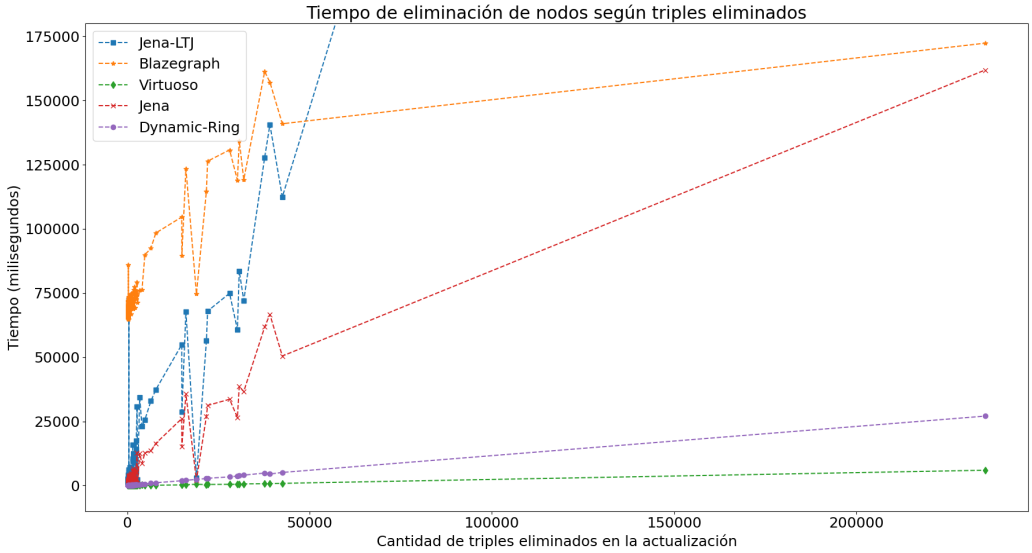


Figura 5.17: Comparación de tiempos de eliminación de nodos, en milisegundos, en función de la cantidad de triples a eliminar.

Con estos resultados se puede posicionar al Ring dinámico como una opción viable para obtener dinamismo en una base de datos de grafos. Sin embargo, se precisa analizar el impacto ocasionado por el diccionario en estas operaciones. En cuanto a la inserción, el diccionario proporciona alrededor de un 25 % del tiempo de la operación, tomando menos de 0.1 milisegundos. Cabe destacar que al insertar sólo hay una traducción de String a ID, en la cual se inserta la palabra si no existía en el diccionario. Los resultados se reflejan en la Figura 5.18.

Por otro lado, la eliminación de aristas y nodos debe traducir sólo de String a ID pero la eliminación del diccionario puede ocurrir en ambas direcciones. Es decir, se debe eliminar la palabra y traducirla para que el Ring lleve a cabo la eliminación con el ID. Pero luego el Ring retorna los IDs que deben ser eliminados del diccionario.

Además, el borrado de nodos toma un tiempo mayor y puede suprimir varios valores del índice, por lo que es preciso que el diccionario sea rápido en la traducción y eliminación de palabras. Esto resulta ser así, donde el tiempo necesitado por el diccionario es varios órdenes menor que la operación en el Ring. Esto se ilustra en la Figura 5.19, donde las áreas verde y naranja, que representan el tiempo de traducción hacia adelante y hacia atrás, conforman, en

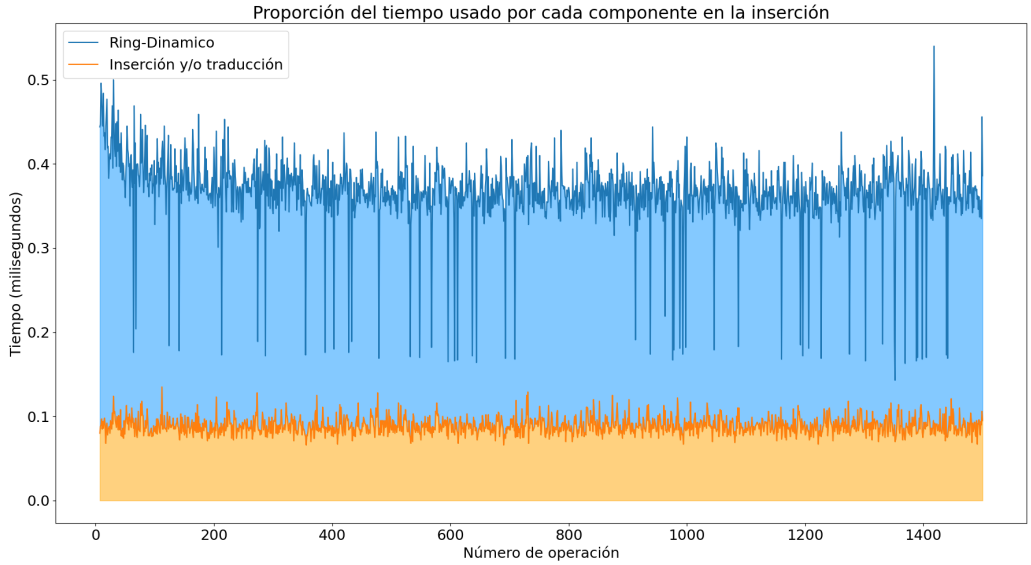


Figura 5.18: Desglose del tiempo de inserción. El tiempo de traducción se marca con el área naranja, mientras que el insertado en el Ring se marca con el área azul.

términos generales, un 25 % del tiempo total necesitado por la operación. Esto es simétrico a la inserción a pesar de tener un caso más de actualización sobre el diccionario. El bajo tiempo de borrado de palabras se debe a que la operación se realiza utilizando IDs, por lo que se aprovecha la rapidez de `extract` para encontrar el PFC correcto y, así, eliminar la palabra indicada.

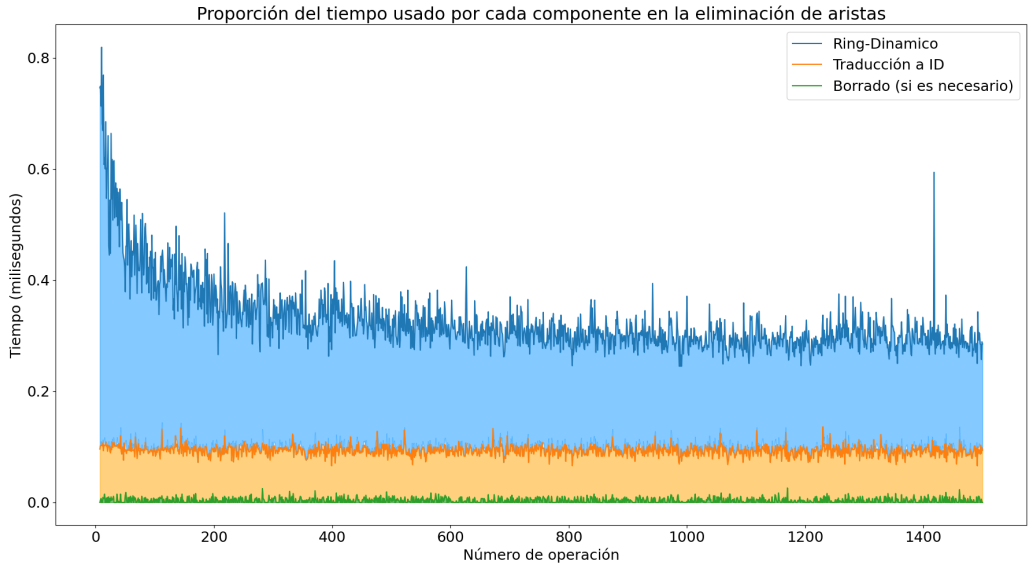


Figura 5.19: Desglose del tiempo de eliminación de aristas. El insertado en el Ring se marca con el área azul, mientras que las traducciones se marcan como áreas naranjas y verdes.

Finalmente, en la eliminación de nodos existe una diferencia mayor entre tiempos de traducción y tiempo de eliminado. Esto se puede deber a que, al forzar varias eliminaciones, se deben fusionar más hojas de los `spsi`, operación que ralentiza la actualización. Esto se ve reflejado en la Tabla 5.2.

Total (ms)	Índice (ms)	Adelante (ms)	Atrás (ms)
199.02	198.24	0.019	0.76

Tabla 5.2: Tiempos promedio de cada parte de la eliminación de nodos. Todos los tiempos están en milisegundos. Índice se refiere al tiempo tomado por el Ring para eliminar los triples asociados al nodo. Adelante es la traducción y eliminación de String a ID y Atrás es la eliminación de identificadores que ya no se utilizan.

No obstante, no siempre se eliminan palabras del diccionario, por lo que se toma como medida el promedio de tiempo que toma eliminar una palabra suprimida en la operación. Así el tiempo que toma remover una palabra usando su ID es, en promedio, 7.3 microsegundos.

Capítulo 6

Conclusiones

El trabajo realizado en esta memoria consistió en modificar la implementación original del Ring para poder aceptar dinamismo, es decir, adición y eliminación de información. Esto requirió de nuevas estructuras de datos compactas, las cuales debían aceptar actualizaciones, y el diseño de un diccionario compacto que permitiera la traducción de un alfabeto de cadenas a identificadores y viceversa.

Se logró diseñar e implementar una versión del Ring que permite dinamismo preservando un espacio utilizado muy cercano al Ring original. El índice creado resultó ser mucho más lento que su versión estática para resolver BGPs, sin embargo, logró superar en rendimiento a implementaciones populares para bases de datos de grafos. Adicionalmente, sólo fue superado en tiempo de consultas por implementaciones que requieren de mucho más espacio para funcionar. Además, se lograron tiempos competitivos en relación a bases de datos WCO, a excepción de aquellos patrones donde existen ciclos. Finalmente, se obtuvieron muy buenos resultados para las operaciones de actualización. Por lo tanto, se pudo cumplir parcialmente el objetivo general, ya que no se logró obtener buenos tiempos transversalmente para los *join*. Por otro lado, los objetivos específicos pudieron ser cumplidos completamente en este trabajo. Se diseñaron e implementaron las estructuras necesarias para construir el Ring dinámico y, a la vez, se realizaron exitosamente experimentos que permitieron validar el desempeño del índice creado.

Los resultados obtenidos fueron producto del diseño inicial de la estructura del Ring dinámico. La incapacidad del índice para realizar rápidamente consultas con cierto tipo de patrón cíclico proviene del costo de utilizar el `spsi` como base para los *bitvectors*. Esto implicó una sobrecarga en las operaciones `rank` y `select`, significando tiempos no competitivos con alternativas del estado del arte. A pesar de esto, sí se lograron muy buenos resultados al comprimir, ocupando un espacio muy similar al Ring original y logrando un 79 % de compresión sobre el dataset en texto plano. Asimismo, las operaciones de actualización obtuvieron muy buenos tiempos, superiores a las demás alternativas.

Con el trabajo realizado se da un paso inicial para obtener dinamismo en el Ring. Al ser una base de datos comprimida con muy buenos resultados, es pertinente llevarla a la mayor cantidad de usos posible. Por esto, se pueden realizar trabajos más exhaustivos sobre este problema basándose en el diseño actual y resolviendo los errores cometidos en este proceso.

Además, se implementó un diccionario comprimido agnóstico al índice usado, el cual tuvo muy buenos resultados en compresión y tiempo de traducción. Esta estructura puede ser utilizada y acoplada a cualquier futura base de datos basada en índices para poder implementar, por ejemplo, una interfaz usando el lenguaje SPARQL.

La memoria presentada requirió de un extenso pensamiento sobre estructuras de datos, algoritmos y optimizaciones sobre la arquitectura de un computador. Paralelamente, se aprendió a utilizar librerías basadas en trabajos científicos y a tomar un proyecto como lo es el Ring e implementar sobre éste. Se tuvo el desafío de aprender a profundidad C++11, para las implementaciones necesarias, y el lenguaje SPARQL, para realizar los experimentos sobre las distintas bases de datos. Sumado a esto, se pudo llevar a la práctica el conocimiento sobre diseño de experimentos y análisis de datos que se tenía.

A partir de los resultados presentados quedan algunos trabajos futuros que se pueden realizar sobre esta memoria. La primera gran problemática es modificar las estructuras propuestas para mejorar la eficiencia de las operaciones de `rank` y `select`, con el objetivo de obtener tiempos de *queries* competitivos con alternativas WCO. Otra futura implementación es añadir un intérprete SPARQL para que el Ring tenga las mismas capacidades que bases de datos como Jena o Virtuoso. Finalmente, un trabajo propuesto es adaptar estas estructuras y algoritmos para relaciones con aridades mayores a 3.

Para concluir, el Ring dinámico es una propuesta que logra almacenar y actualizar información utilizando poco espacio, lo que puede ser sumamente valioso en contextos de almacenamiento de grafos. Así, se posiciona como una alternativa viable del estado del arte de las bases de datos comprimidas y dinámicas.

Bibliografía

[1] Waqas Ali, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel Cyrille Ngonga Ngomo. A survey of rdf stores & sparql engines for querying knowledge graphs. *VLDB Journal*, 31:1–26, 2022.

[2] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40:1–39, 2008.

[3] Renzo Angles, Harsh Thakkar, and Dominik Tomaszuk. Mapping rdf databases to property graph databases. *IEEE Access*, 8:86091–86110, 2020.

[4] Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, Juan L. Reutter, Javiel Rojas-Ledesma, and Adrián Soto. Worst-case optimal graph joins in almost no space. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 102–114, 2021.

[5] Diego Arroyuelo, Gonzalo Navarro, Juan L. Reutter, and Javiel Rojas-Ledesma. Optimal joins using compressed quadtrees. *ACM Transactions on Database Systems*, 47:1–53, 2022.

[6] Albert Atserias, Martin Grohe, and Daniel Marx. Size bounds and query plans for relational joins. In *SIAM Journal on Computing*, volume 42, pages 1737–1767, 2013.

[7] Alexander Bigerl, Felix Conrads, Charlotte Behning, Mohamed Ahmed Sherif, Muhammad Saleem, and Axel Cyrille Ngonga Ngomo. Tentris – a tensor-based triple store. volume 12506 LNCS, pages 56–73, 2020.

[8] Gonçalo Carnaz, Vitor Beires Nogueira, and Mário Antunes. A graph database representation of portuguese criminal-related documents. *Informatics*, 8:37, 2021.

[9] David Richard Clark. *Compact Pat Trees*. PhD thesis, CAN, 1998. UMI Order No. GAXNQ-21335.

[10] Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez. The wavelet matrix: An efficient wavelet tree for large alphabets. volume 47, pages 15–32, 2015.

[11] Miguel E. Coimbra, Joana Hrotkó, Alexandre P. Francisco, Luís M.S. Russo, Guillermo de Bernardo, Susana Ladra, and Gonzalo Navarro. A practical succinct dynamic graph representation. *Information and Computation*, 285:104862, 2022.

[12] Joshimar Cordova and Gonzalo Navarro. Practical dynamic entropy-compressed bitvectors with applications. volume 9685, pages 105–117, 2016.

[13] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vañó, S. Gómez-Villamor, N. Martínez-Bazán, and J. L. Larriba-Pey. In *Survey of graph database performance on the HPC scalable graph analysis benchmark*, volume 6185 LNCS, pages 37–48, 2010.

[14] Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21:194–203, 1975.

[15] Orri Erling and Ivan Mikhailov. Rdf support in the virtuoso dbms. *Studies in Computational Intelligence*, 221:7–24, 2009.

[16] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[17] 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, page 345–354, New York, NY, USA, 1989. Association for Computing Machinery.

[18] Travis Gagie, Gonzalo Navarro, and Simon J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426-427, 2012.

[19] S. Golomb. Run-length encodings (corresp.). *IEEE Transactions on Information Theory*, 12(3):399–401, 1966.

[20] Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. A worst-case optimal join algorithm for sparql. volume 11778 LNCS, pages 258–275, 2019.

[21] Guy Jacobson. Space-efficient static trees and graphs. 1989.

[22] Salim Jouili and Valentin Vansteenberghe. An empirical comparison of graph databases. pages 708–715, 2013.

[23] Oren Kalinsky, Yoav Etsion, and Benny Kimelfeld. Flexible caching in trie joins. volume 2017-March, pages 282–293, 2017.

[24] Jon M. Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew S. Tomkins. The web as a graph: Measurements, models,and methods. volume 1627, pages 1–17, 1999.

[25] Patrick Klitzke and Patrick K. Nicholson. A general framework for dynamic succinct and compressed data structures. volume 2016-January, pages 160–173, 2016.

[26] Miguel A. Martínez-Prieto, Nieves Brisaboa, Rodrigo Cánovas, Francisco Claude, and Gonzalo Navarro. Practical compressed string dictionaries. *Information Systems*, 56, 2016.

[27] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A. Bader. A performance evaluation of open source graph databases. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, PPAA '14, page 11–18, New York, NY, USA, 2014. Association for Computing Machinery.

[28] Gonzalo Navarro. Wavelet trees for all. In *Journal of Discrete Algorithms*, volume 25, pages 2–20, 2014.

[29] Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, USA, 1st edition, 2016.

[30] Neo4j. Neo4j: The world leading graph database. *Neo4J.Org*, 2016.

[31] Thomas Neumann and Gerhard Weikum. Rdf-3x: A risc-style engine for rdf. *Proceedings of the VLDB Endowment*, 1:647–659, 2008.

[32] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *Journal of the ACM*, 65:1–40, 2018.

[33] Dung Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Join processing for graph patterns: An old dog with new tricks. In *3rd International Workshop on Graph Data Management Experiences and Systems, GRADES - co-located with SIGMOD/PODS*, pages 1–8, 2015.

[34] Nicola Prezza. A framework of dynamic data structures for string processing. In *Leibniz International Proceedings in Informatics, LIPIcs*, volume 75, pages 1–15, 2017.

[35] Bryan B. Thompson, Michael Personick, and Martyn Cutcher. The bigdata® rdf graph database. In *Linked Data Management*, pages 192–237, 2014.

[36] Santiago Timón-Reina, Mariano Rincón, and Rafael Martínez-Tomás. An overview of graph databases and their applications in the biomedical domain. *Database*, 2021, 05 2021.

[37] Kazuya Tsuruta, Dominik Köppl, Shunsuke Kanda, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. c-trie++: A dynamic trie tailored for fast prefix searches. *Information and Computation*, 285:104794, 2022.

[38] Todd L Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *Proc. of ICDT'14*, pages 96–106, 2014.

[39] Denny Vrandečić and Markus Krötzsch. Wikidata: A free collaborative knowledgebase. *Communications of the ACM*, 57:78–85, 2014.

[40] Hugh Williams and Justin Zobel. Compressing integers for fast file access. *The Computer Journal*, 42, 08 2002.

[41] Hugh E. Williams and Justin Zobel. Compressing integers for fast file access. *Computer Journal*, 42:193–201, 1999.