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

ESTRUCTURAS COMPACTAS DINÁMICAS MÁS EFICIENTES PARA BASES DE DATOS DE GRAFOS CON ATRIBUTOS

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

DAVID IGNACIO DE LA PUENTE VÉLIZ

PROFESOR GUÍA:
GONZALO NAVARRO BADINO
PROFESOR CO-GUÍA:
DIEGO ARROYUELO

MIEMBROS DE LA COMISIÓN:
JAVIER BUSTOS JIMÉNEZ
JORGE PÉREZ ROJAS

Esta memoria ha sido parcialmente financiada por el proyecto FONDECYT 1-200038

SANTIAGO DE CHILE
2021

Resumen

Los grafos son estructuras de datos muy útiles ya que permiten guardar conexiones entre entidades que comparten alguna relación. A veces estas estructuras se extienden para poder guardar información en sus nodos y aristas, a esta variante se le conoce como grafos con atributos. Debido a la importancia de los grafos con atributos, existen varios motores de bases de datos que buscan guardar estas estructuras de forma compacta incluyendo operaciones que permiten consultar la base de datos de forma eficiente. Pero guardar la base de datos y consultarla no es lo único que deben satisfacer los motores, además deben permitir dinamismo en sus estructuras, es decir la capacidad de cambiar la información que representan. Es por esto que existen motores de bases de datos que basan su implementación en estructuras de datos que no solo son compactas, si no que además son dinámicas.

En este trabajo se estudia el motor de base de datos Attk2DynTree. Este motor basa su implementación en una estructura de datos sucinta y dinámica llamada k^2-tree. Esta estructura permite comprimir la matriz de adyacencia dividiendo la matriz en sub matrices más pequeñas y guardando solo aquellas que contienen información. Debido a lo anterior, los k^2-trees suelen ser bastante eficientes ya que, en la práctica los grafos suelen ser esparsos y con subconjuntos de elementos densamente conectados.

Actualmente existen k^2-trees más eficientes que los que actualmente posee Attk2DynTree. Lo que se busca con este trabajo es elegir el mejor de ellos y reemplazar el antiguo k^2-tree que hay en Attk2DynTree con uno más eficiente y de esta forma presentar una nueva versión de Attk2DynTree. Para lograr lo anterior, se divide el trabajo total en tres grandes hitos. El primero consiste en implementar la operación de borrado de aristas, para un k^2-tree desarrollado recientemente. En esta parte del trabajo se muestra que la nueva operación tiene una complejidad de tiempo igual al resto de operaciones. En el segundo hito se comparan cuatro implementaciones de k^2-trees, en una serie de pruebas que intentan medir el tiempo que demoran sus operaciones y la memoria que ocupan. Al final de esta sección se elige un k^2-tree candidato para ser incorporado en la nueva versión de Attk2DynTree. En el último hito, se integra el nuevo k^2-tree en Attk2DynTree, y se ponen a prueba las dos versiones usando grafos reales de *ratings* de películas. Finalmente, se concluye que la nueva versión de Attk2DynTree es mucho más rápida que la versión antigua en las operaciones que involucran a los k^2-trees, se obtienen tiempos de inserción hasta 20 veces más rápidos y consultas de búsqueda hasta 3 veces más rápidas, ocupando un 28 % de memoria extra.

Tabla de contenido

1. Introducción **1**

2. Estado del arte **2**
2.1. Matriz de adyacencia . . . 2
2.2. k^2-tree . . . 3
2.3. k^2-trie . . . 3
2.4. Arreglo dinámico . . . 3
2.5. LOUDS y DFUDS . . . 4
2.6. Estado del arte de k^2-trees dinámicos . . . 5
2.7. Situación actual del motor de bases de datos de grafos . . . 6
2.8. Datos . . . 7
2.8.1. Datasets para testeo de k^2-trees . . . 7
2.8.2. Datasets para testeo de Attk2DynTree . . . 7

3. Objetivos **8**
3.1. Objetivo General . . . 8
3.2. Objetivos Específicos . . . 8
3.3. Motivación . . . 8
3.4. Limitantes . . . 8

4. Metodología **9**
4.1. Hito 1 . . . 9
4.2. Hito 2 . . . 9
4.3. Hito 3 . . . 10

5. Agregando borrados a Blocks **11**
5.1. Blocks . . . 11
5.1.1. Estructura de datos . . . 11
5.1.2. Funciones más importantes . . . 12
5.2. Tests para Blocks . . . 12
5.2.1. Materialize suite . . . 13
5.3. Evolución de tiempo y memoria en la inserción de aristas . . . 13
5.3.1. Configuración del test . . . 13
5.3.2. Resultados . . . 14
5.4. Borrado de aristas . . . 15
5.5. Tests para el borrado de aristas . . . 16
5.6. Evolución de tiempo y memoria en el borrado de aristas . . . 16
5.6.1. Resultados . . . 16
5.7. Optimizando el borrado de aristas . . . 18
5.7.1. Parámetros de la optimización . . . 18
5.7.2. Algoritmo para fusionar bloques . . . 18
5.8. Tests para la optimización del borrado de aristas . . . 19
5.9. Evolución de tiempo y memoria en el nuevo borrado de aristas . . . 19
5.9.1. Resultados . . . 19
5.10. Inserciones después de borrar . . . 20
5.10.1. Resultados . . . 20

5.11. Análisis general de los resultados del Hito 1 21

6. Alternativas de k^2-trees **22**
6.1. Dyn-array1 22
6.1.1. Estructuras de datos 22
6.1.2. Funciones más importantes 22
6.2. Dyn-array2 23
6.2.1. Estructuras de datos 24
6.2.2. Funciones más importantes 24
6.3. HashTrie 25
6.3.1. Estructuras de datos 25
6.3.2. Funciones más importantes 25
6.4. Ustatic 26
6.4.1. Estructuras de datos 26
6.4.2. Funciones más importantes 26
6.5. Nuevas funciones para Blocks 27
6.6. Complejidades recopiladas 28

7. Comparación de los k^2-trees **29**
7.1. Configuración de los experimentos 29
7.1.1. Configuración de las estructuras 29
7.2. Insertar 194M aristas 29
7.3. Buscar 194M aristas 31
7.4. Insertar y borrar 194M aristas 31
7.5. Inserciones post borrado de aristas 33
7.6. Oleadas 20-15 34
7.7. Vecinos 35
7.8. Consulta de rango 36
7.9. Otros experimentos 37
7.10. Elección del k^2-tree 39

8. Integración en el motor de base de datos **40**
8.1. Attk2DynTree 40
8.2. Estructura de datos 41
8.2.1. Esquema 41
8.2.2. Atributos 41
8.2.3. Relaciones 42
8.3. Inserciones en Attk2DynTree 45
8.3.1. Insertar en un esquema 45
8.3.2. Insertar en la topología 45
8.4. Consultas sobre Attk2DynTree 46
8.4.1. Consultas sobre los esquemas 46
8.4.2. Consultas sobre los atributos 46
8.4.3. Consultas sobre las relaciones 47
8.5. Evaluación de Dyn-array1 47
8.5.1. Insertar 194 millones de aristas 47
8.5.2. Buscar 194 millones de aristas 48

8.5.3. Consulta de rango 49
8.6. Integración de Blocks en Attk2DynTree 50
8.6.1. Estructuras de datos intervenidas 50
8.7. Inserciones en Attk2DynTree2 50
8.7.1. Consultas intervenidas 50

9. Evaluación de la integración **51**
9.1. Dataset ml100k 51
9.2. Datasets ml10M y ml25M 52
9.3. Comparación experimental 53
9.3.1. Tiempos de inserciones en Attk2DynTree 53
9.3.2. getNodeAttribute 55
9.3.3. getEdgeAttribute 56
9.3.4. selectNodes 57
9.3.5. selectEdges 57
9.3.6. neighbors 58
9.4. Análisis general de los resultados del Hito 3 59

10. Conclusión y trabajo futuro **60**

11. Bibliografía **61**

1. Introducción

Las relaciones binarias son importantes en el mundo de la información, ya que nos ayudan a representar conexiones entre entidades que comparten alguna correspondencia. La forma usual de representar estas relaciones es a través de un grafo, el que a su vez suele representarse con una matriz de adyacencia. Esta matriz no solo representa bien el grafo, sino que además su estructura le permite a la matriz ser consultada para responder a preguntas como: ¿cuáles son los vecinos que llegan a cierto nodo?, ¿cuáles son los vecinos que salen de cierto nodo?, ¿qué vecinos llegan a cierto rango de nodos?, entre otras. Además esta estructura permite tener operaciones para actualizar el grafo, como borrar o agregar aristas.

El problema con estas matrices es que reservan espacio para cada una de las n^2 posibles aristas que un grafo de n nodos puede tener, sin tener en cuenta si las aristas existen o no. Y en la era del *Big data*, lo anterior no escala bien en términos de memoria para guardar la matriz, y tiempo para consultarla. Podemos ver ejemplos de esto en grafos web [17] e interacciones en redes sociales [14], donde la cantidad de entidades es tan grande que dicha matriz simplemente no cabe en memoria.

Una solución al problema anterior es utilizar estructuras de datos compactas, que utilizan mucha menos memoria que las clásicas. Existen varias estructuras compactas para representar grafos y relaciones binarias. En particular, Brisaboa et al. [6] proponen utilizar una estructura de datos llamada k^2-tree para comprimir las matrices de adyacencia y poder realizar consultas sobre esta estructura, sin la necesidad de descomprimirla. El k^2-tree explota el hecho de que, en muchas aplicaciones reales, las relaciones son esparsas (relativamente pocas aristas) y tienen clusters (subconjuntos de elementos densamente conectados).

Las bases de datos de grafos son, sin embargo, más complejas. Requieren guardar información en los nodos y en las aristas (nombres, atributos, identificadores, etc.). A este tipo de grafos se le conoce como grafos con atributos. Para poder construir y hacer consultas sobre estos modelos de información, se han desarrollado diversos motores de bases de datos especializados en grafos con atributos. Algunos ejemplos son Dex [11], HyperGraphDB [9], y SQL Graph [18]. Estas bases de datos están compuestas por conjuntos de estructuras de datos y algoritmos que intentan optimizar los diferentes tipos de consulta sobre los grafos, en términos de memoria y tiempo.

Un requisito importante para implementar una base de datos de grafos es que las estructuras soporten funcionalidades como agregar y borrar aristas, incluso agregar y borrar nodos. Por ello se requiere que las estructuras no solo sean compactas, sino que también deben ser dinámicas. La estructura original del k^2-tree requiere reconstruirla desde cero para reflejar cambios, por lo que no es adecuada para implementar bases de datos de grafos. Existe una versión dinámica basada en arreglos dinámicos [15]. Esta versión es utilizada en el motor de base de datos de grafos Attk2DynTree [1]. Sin embargo, trabajos recientes [2] [3] [6] [7] muestran que el dinamismo se puede soportar en forma mucho más eficiente. El objetivo de este trabajo es estudiar estas alternativas, compararlas, extenderlas de ser posible para soportar operaciones requeridas por el dinamismo, e integrar el mejor candidato a Attk2DynTree para obtener un prototipo más eficiente.

2. Estado del arte

En esta sección se definen y explican las estructuras de datos básicas utilizadas en el trabajo. Luego, se muestra el estado del arte en los k^2-trees. Finalmente, se menciona la situación actual del motor de base de datos a intervenir.

2.1. Matriz de adyacencia

Dados dos conjuntos A y B, una relación binaria es un subconjunto del producto cartesiano entre A y B, que cumple cierta condición r. Se dice que $a \in A$ se relaciona con $b \in B$ si $r(a, b) = 1$, y no se relacionan si $r(a, b) = 0$. Por lo tanto el conjunto $R = \{(a, b) \in A \times B | r(a, b) = 1\}$ sería la relación. Con lo anterior, se puede construir una matriz M de tamaño $|A| \times |B|$ tal que $M[i][j] = r(A[i], B[j])$. Esta matriz representa la relación R y se llama matriz de adyacencia. En la parte superior de la Figura 1, se muestra un ejemplo de matriz de adyacencia, donde los puntos negros corresponden a $r(a, b) = 1$.

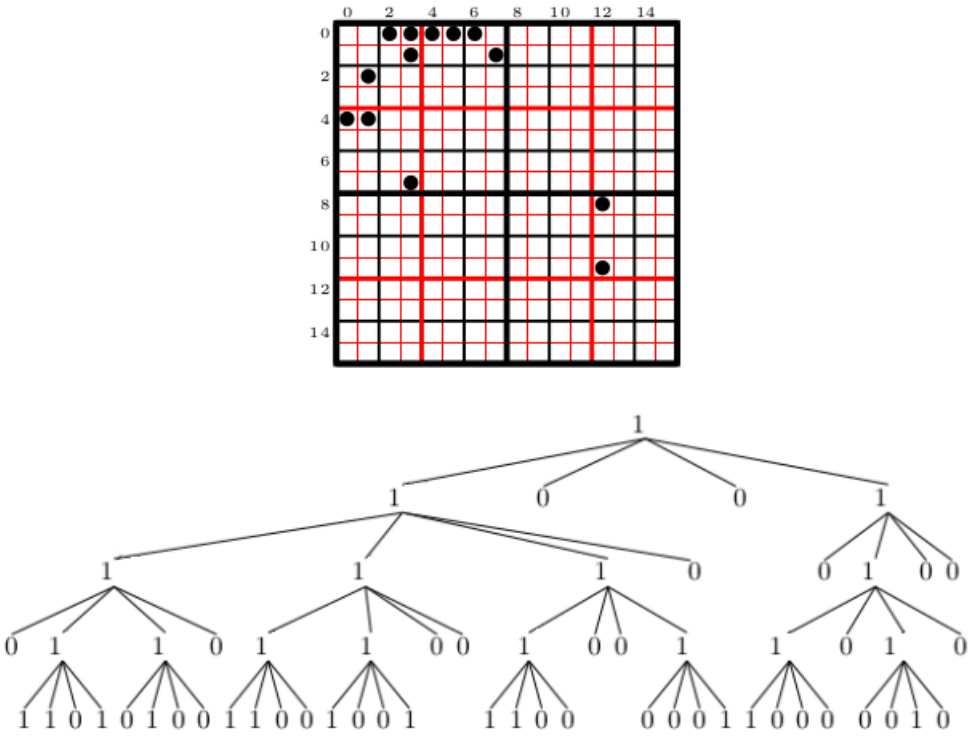

Figura 1: Una relación binaria (arriba) y el correspondiente k^2-tree (abajo)

2.2. k^2-tree

Dada una matriz de adyacencia M para una relación R, un k^2-tree es un árbol de aridad k^2, donde cada nodo representa una sub matriz cuadrada de M y cada rama representa un elemento de R [6]. Se puede asumir que la matriz es de $n \times n$, con n una potencia de k (si M es más pequeña, lo que se hace es rellenar con 0's los espacios restantes).

Cada nodo del árbol tiene un valor 0 o 1. Si el valor es 0, significa que no hay elementos que se relacionen en esa sub matriz. Por otro lado si el valor es 1, significa que hay algún elemento de R en la sub matriz, y esta se divide en k^2 sub matrices cuadradas de igual tamaño. Estas nuevas matrices son representadas por los hijos del nodo bajo algún orden. En particular en este trabajo se usa el siguiente orden de indexación: Por cada fila de sub matrices de arriba hasta abajo, se indexan primero las sub matrices de más a la izquierda, y al final las de más a la derecha (un caso particular de esta indexación es cuando k=2, y se llama código de Morton [12] o z-orden ya que al unir los índices de 0 a 3 se forma una Z). Por último, por cada hijo del nodo, si su sub matriz tiene elementos en R, guardamos un 1 en ese hijo y se repite el proceso recursivamente hasta que las sub matrices sean de 1×1. Por simplicidad desde ahora se asume $k = 2$, pero la idea general se mantiene. En la parte de abajo de la Figura 1 (obtenida del trabajo de Arroyuelo et al. [2]) se muestra un ejemplo de un k^2-tree ya construido para $k = 2$.

2.3. k^2-trie

Un k^2-trie es un trie que representa un k^2-tree bajo alguna codificación de sus aristas, por ejemplo la codificación de Morton [2]. Para esto, lo que se hace es codificar el camino que siguen los nodos del k^2-tree desde la raíz hasta las hojas, en un string de largo l que contiene los índices de las sub matrices elegidas, y se guarda esa codificación en un trie. Por ejemplo si $k = 2$ y se usa codificación de Morton y el largo del árbol es $l = 4$, un elemento de M puede estar representado por el string $s = 0311$, esto indica que ese elemento tomó la sub matriz 0, dentro de ella tomó la sub matriz 3, dentro de la sub matriz 3 tomó la sub matriz 1 y así sucesivamente. Si se codifican todas las aristas y se guardan en un trie, se obtiene un k^2-trie.

2.4. Arreglo dinámico

Un arreglo dinámico es un arreglo que permite inserciones o borrados en cualquier posición. Una forma básica de implementar esta estructura, es usar un arreglo estático de tamaño n con $k \leq n$ elementos. Si al insertar t elementos, resulta que $k + t > n$ entonces, se realoca la memoria a un arreglo de tamaño $n' > k + t$. Para insertar t elementos en una posición i que no sea la última, lo que hace hace es correr t espacios a la derecha todos los elementos del arreglo desde la posición i hasta k. Existen formas más sofisticadas de implementar arreglos dinámicos, por ejemplo basadas en *Searchable Partial Sums with Inserts* (spsi) [15], o basadas en weight-balanced B-trees (WBB) [16]

2.5. LOUDS y DFUDS

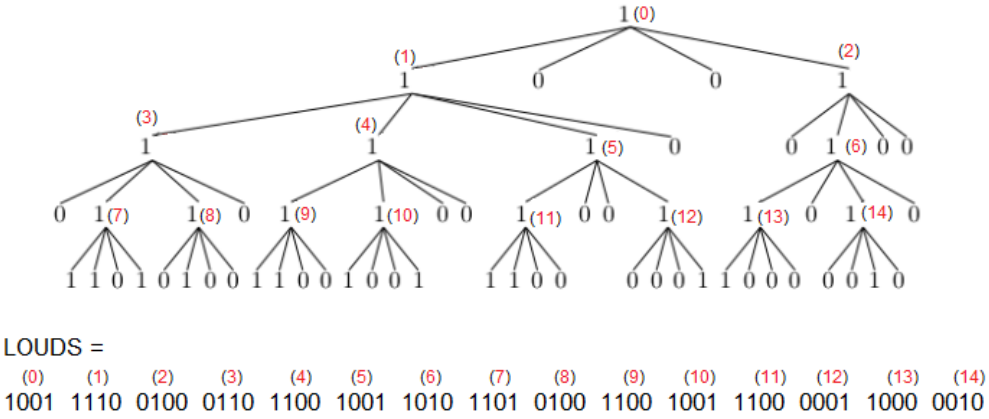

Figura 2: Arreglo LOUDS, creado recorriendo el árbol con BFS

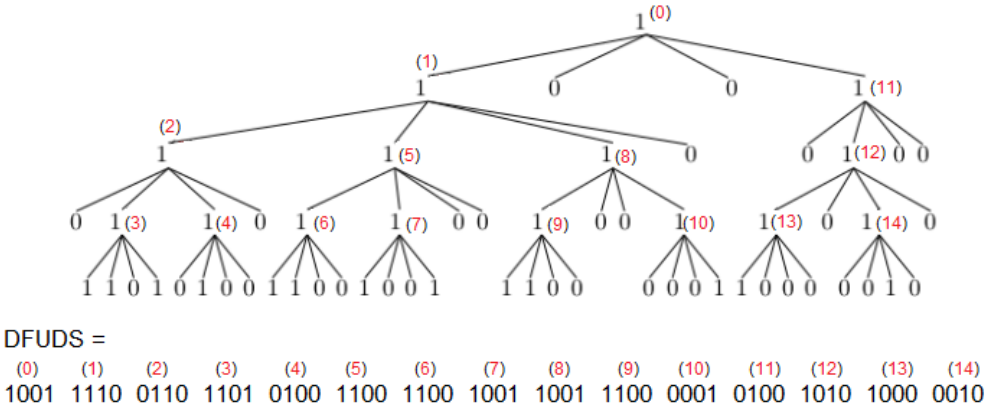

Figura 3: Arreglo DFUDS, creado recorriendo el árbol con DFS

LOUDS y DFUDS son dos estrategias que permiten construir arreglos de bits para representar los k^2-trees [13]. Por un lado, la estrategia LOUDS (*level order unary degree sequence*) se basa en recorrer el árbol siguiendo un recorrido BFS (*Breadth-First Search*), donde se escribe el arreglo nivel por nivel. Por el otro lado, DFUDS (*depth-first unary degree sequence*) se basa en recorrer el árbol siguiendo un recorrido DFS (*Depth-First Search*), donde se escribe el arreglo completando primero los caminos de más a la izquierda, y terminando con los de más a la derecha. La ventaja de LOUDS, es que permite tener todo el último nivel contiguo en el arreglo, Mientras que la ventaja de DFUDS es que todos los descendientes de un mismo nodo son contiguos. La Figura 2 muestra cómo se representa un k^2-tree con LOUDS, y la Figura 3 con DFUDS. Para navegar LOUDS se utiliza la función rank[1] para saber cuántos 1's hay hasta determinada posición. Con ese valor se puede calcular la posición de cualquier hijo de un nodo. Por otro lado, para navegar DFUDS se utilizan operaciones más complejas sobre paréntesis balanceados.

[1]Rank(B,i) cuenta el número de 1´s en el arreglo de bits B, desde la posición 0 hasta i

2.6. Estado del arte de k^2-trees dinámicos

A continuación se presentan las cinco versiones de k^2-trees que se utilizan en este trabajo. En cada versión se aprecia el uso de distintas estructuras de datos para soportar dinamismo. A priori las versiones no tienen un nombre en particular, así que se etiquetan con nombres en función de la estructura principal que usan. Más adelante, en la sección 5 y 6, se explica en detalle cómo funcionan.

1. **Dyn-array1:** El primer k^2-tree es el que se encuentra en el motor de base de datos a intervenir [1]. Consiste en dos arreglos de bits dinámicos llamado T y L. T contiene todos los nodos internos del k^2-tree siguiendo una estrategia LOUDS, y L contiene todas las hojas del k^2-tree, también siguiendo una estrategia LOUDS. El arreglo de bits dinámico es el que aparece en la librería propuesta por Nicola Prezza [15].
2. **Dyn-array2:** El segundo trabajo que se estudia, es el de Brisaboa et al. [6]. Este trabajo también usa dos arreglos de bits dinámicos T y L siguiendo una estrategia LOUDS. Pero estos arreglos particionan T y L en bloques de hasta B bits. Cada bloque es una hoja en un árbol balanceado. Los nodos internos del árbol contienen información que permite hacer operaciones dinámicas sobre el arreglo completo.
3. **Blocks:** En esta tercera implementación, Arroyuelo et al. [2] proponen interpretar el k^2-tree como un trie. Este k^2-trie es una estructura híbrida en el sentido de que los primeros l niveles del trie se componen de un trie básico con k^2 hijos (los hijos son punteros a otros tries). En cambio los niveles de más abajo se guardan en bloques de tamaño B que contienen un sub-trie codificado en un arreglo de bits dinámico, siguiendo una estrategia DFUDS. El bloque además contiene información como punteros a los bloques hijos, número de nodos, tamaño del bloque, y otros.
4. **HashTrie:** En este trabajo Arroyuelo et al. [3] proponen usar un trie basado en tablas de hash para guardar el k^2-tree. Los autores usan este trie para mejorar el algoritmo de compresión LZ78. Sin embargo, se puede usar la codificación de Morton del k^2-tree sobre este trie para crear un k^2-trie. Para guardar/buscar un nodo t del trie, la función de hash usa la información del padre de t y la etiqueta de la arista que los une (la codificación).
5. **Ustatic:** La última implementación de k^2-tree que se estudia, es la de Coimbra et al. [7]. En este trabajo los autores proponen crear un k^2-tree dinámico basado en la unión de k^2-trees estáticos. Lo que hacen es representar la matriz de adyacencia como un conjunto de k^2-trees estáticos de tamaños fijos $E_1, ...E_r$ y una lista de adyacencia dinámica no comprimida E_0, tal que $\forall i \in [0, r-1]\ |E_i| < |E_{i+1}|$. De esta forma tenemos que el k^2-tree dinámico es igual a $\cup_{k=0}^{r} E_k$. Para reflejar cambios, la estructura reconstruye desde cero algunos de los E_i involucrados.

nombre	buscar	insertar	borrar	vecinos	range	insertar nodo	borrar nodo
Dyn-array1	✓	✓		✓	✓	✓	
Dyn-array2	✓	✓	✓	✓	✓	✓	✓
Blocks	✓	✓					
HashTrie	✓	✓					
Ustatic	✓	✓	✓	✓		✓	✓

Tabla 1: Tabla de operaciones implementadas en código, las estructuras estudiadas

En la Tabla 1 se muestran las distintas operaciones que han sido implementadas para cada una de las estructuras. Un ✓significa que la operación está implementada, mientras que si no hay nada en la casilla significa que la operación no está disponible.

- **Buscar:** corresponde a buscar una arista entre dos nodos de grafo.
- **Insertar:** corresponde a insertar una arista entre dos nodos de grafo.
- **Borrar:** corresponde a quitar una arista entre dos nodos de grafo.
- **Vecinos:** corresponde a buscar todas las aristas que salen de un nodo (en la matriz de adyacencia, significa recuperar todos los 1's de la fila del nodo).
- **Range:** Corresponde a calcular la intersección entre la matriz de adyacencia y algún rectángulo de consulta. Se devuelven todas las aristas que están en la intersección.
- **Insertar nodo:** corresponde a agregar un nodo extra al grafo (esto implica agregar filas y columnas en la matriz de adyacencia).
- **Borrar nodo:** corresponde a eliminar un nodo del grafo (esto implica quitar filas y columnas en la matriz de adyacencia).

2.7. Situación actual del motor de bases de datos de grafos

En el trabajo de Álvarez-Garcia et al. [1] se propone Attk2DynTree, una estructura capaz de guardar multi grafos con atributos. En estos grafos los nodos y aristas pueden tener diferentes tipos y cada tipo tiene un conjunto de posibles atributos. Por ejemplo, pueden haber nodos de tipo película que tiene atributos (nombre, estreno) y nodos de tipo usuario con atributos (nombre, edad) y se relacionan a través de la arista de tipo *rating* con atributos (*rate*, fecha). Attk2DynTree representa el grafo con tres componentes: esquemas, atributos y relaciones; las cuales se resumen a continuación.

- **Esquema:** Es una estructura utilizada para guardar los posibles tipos que puede tener un nodo o una arista, y los posibles atributos que puede tener cada tipo.
- **Atributos:** Es la parte de Attk2DynTree que se encarga de guardar los valores que toman los atributos de cada tipo.
- **Relaciones:** Es la parte de Attk2DynTree que guarda la topología del grafo. Es decir, guarda las aristas que conectan los nodos del grafo (puede haber más de una arista entre dos nodos).

Actualmente, el motor de base de datos presentado en la sección anterior utiliza el k^2-tree **Dyn-array1** para implementar las estructuras que almacenan los **atributos** y las **relaciones**. El dinamismo de este k^2-tree permite que el motor pueda editar la topología de los grafos y editar algunos de los atributos de cada nodo o arista, se retomará esta descripción en la sección 8. Por otro lado, experimentos como los de Arroyuelo et al. [2] muestran que hay k^2-trees mucho más rápidos, tanto al momento de realizar inserciones como búsqueda de aristas. Se espera que una versión de Attk2DynTree que utilice estos k^2-trees tenga un mejor desempeño que el motor actual.

2.8. Datos

2.8.1. Datasets para testeo de k^2-trees

Para estudiar la capacidad de compresión de los k^2-trees, es necesario hacer pruebas con grafos de gran tamaño. Siguiendo los experimentos de Arroyuelo et al. [2], se usa el dataset de grafos web indochina-2004 extraido de *The Laboratory for Web Algorithmics* [2] [4, 5]. Se decide usar este dataset ya que es conocido por ser un grafo poco denso y compresible [2]. En *The Laboratory for Web Algorithmics* se pueden encontrar varios tipos de grafos (web, de redes sociales, y otros). La Tabla 2 contiene las características del grafo usado en los experimentos de las secciones siguientes.

Dataset	Tipo	Nodos (millones)	Aristas (millones)
indochina-2004	Grafo web	7.4	194.1

Tabla 2: Datasets de grafos

2.8.2. Datasets para testeo de Attk2DynTree

Para evaluar el motor de bases de datos, lo principal es conseguir datos que puedan ser representados usando multi grafos con atributos. Para esto, se siguen los experimentos de Álvarez-Garcia et al. [1]. Sus datos consisten en datasets de ratings de péliculas recopilados por *GroupLens Research* [3] [8]. Los datasets usados en este trabajo son tres:

- **ml100k:** Contiene 100.000 ratings (entre 1 y 5) sobre 1.682 películas, hechos por 943 usuarios. Cada usuario tiene los atributos (`userId, edad, género, ocupación, código postal`). Cada película posee (`movieId, titulo, estreno, url`) y un arreglo de géneros a los que pertenece la película (acción, drama, horror, etc). Además el dataset posee un archivo con tuplas (`userId, movieId, rate, timestamp`) que representa el *rating* que userId hace sobre movieId.
- **ml10M:** Contiene 10.000.054 ratings (entre 1 y 5) sobre 10.681 películas, hechos por 71.567 usuarios. Esta vez los usuarios aparecen anónimos, por lo que solo tienen el atributo userId. Cada película permanece igual que en ml100k, pero se le quita el atributo url. El archivo de *ratings* tiene la misma estructura que en ml100k.
- **ml25M:** Contiene 25.000.095 ratings (entre 1 y 5) sobre 62.423 películas, hechos por 162.541 usuarios. Los usuarios, las películas y el archivo de ratings, mantienen la misma estructura que en ml10M.

[2] `http://law.di.unimi.it/datasets.php`
[3] `https://grouplens.org/datasets/movielens/`

3. Objetivos

3.1. Objetivo General

El objetivo principal de este trabajo es desarrollar una versión optimizada de Attk2DynTree que implemente de forma eficiente las siguientes operaciones:

- inserción de aristas.
- inserción de valores de nodos.
- inserción de valores de aristas.
- búsqueda de valores de atributos para nodos.
- búsqueda de valores de atributos para aristas.
- búsqueda de todos los nodos con cierto valor en un atributo.
- búsqueda de todas las aristas con cierto valor en un atributo.
- búsqueda de vecinos sobre las relaciones.

Las operaciones se explican en detalle en la sección 8. Para lograr lo anterior se divide el objetivo general en cuatro objetivos específicos.

3.2. Objetivos Específicos

1. Implementar el borrado de aristas en el k^2-tree de Arroyuelo et al .[2].
2. Realizar experimentos sobre todos los k^2-trees propuestos para medir tiempo y memoria al realizar búsquedas, inserciones, borrados y cualquier otra operación necesaria.
3. Integrar en Attk2DynTree el k^2-tree más eficiente según los resultados del objetivo 2.
4. Comparar la nueva versión de Attk2DynTree con la de Álvarez-Garcia et al. [1].

3.3. Motivación

Lo que motiva a concretar estos objetivos es la oportunidad de mejorar Attk2DynTree, reemplazando los k^2-trees que utiliza con otros que resultan ser mejores en la práctica. De esta forma se espera que la nueva versión de Attk2DynTree optimice el tiempo de respuesta de las consultas que involucran a los k^2-trees sin afectar mucho la memoria que utiliza.

3.4. Limitantes

El actual motor de base de datos se basa en estructuras que no tienen la capacidad de borrar. Por lo tanto Attk2DynTree no permite borrados de ningún tipo. De todas formas se evalúa el borrado de aristas en los k^2-trees, ya que puede dar lugar a una actualización futura. Por otro lado, el Attk2DynTree aprovecha características de Dyn-array1 que le permiten soportar multi grafos. Algunos de los k^2-trees evaluados no poseen esas características, por lo que no pueden usarse para soportarlos (pero nada impide adaptarlos en un trabajo futuro). Es por esto que algunas de las consultas del motor se dejan fuera de los experimentos (en particular la consulta que retorna los vecinos de un nodo, relacionados a través de un tipo especifico de arista). En la sección 8 se rediscute la característica de Dyn-array1 que permite multi grafos.

4. Metodología

Para realizar el trabajo de manera más ordenada, se divide la metodología en tres hitos principales, los cuales se describen a continuación.

4.1. Hito 1

El Hito 1 consiste en trabajar en el k^2-tree Blocks (punto 3 de la sección 2.6). Se parte estudiando la implementación de la estructura, [4] luego se ponen a prueba las operaciones existentes, y finalmente se implementa la operación de borrado de aristas. Esta última parte se divide en las siguientes sub tareas:

1. **Implementar el borrado de aristas:** Como se observa en la Tabla 1, a Blocks le falta esta operación, por lo que la primera tarea es implementarla.
2. **Testing de borrado de aristas:** En esta parte se desarrollan tests que ponen a prueba la operación anterior. La idea es comprobar que la estructura tiene el comportamiento deseado.
3. **Optimizar el borrado de aristas:** Se busca implementar borrados de forma eficiente, evitando casos patológicos de peor caso. Es importante mencionar que, luego de esta optimización, la estructura debe seguir pasando los tests anteriores.

4.2. Hito 2

El Hito 2 consiste básicamente, en poner a prueba todos los k^2-trees de la sección 2.6. Como mínimo se prueban las operaciones de inserción y búsqueda de aristas. Los k^2-trees que soportan funcionalidades más sofisticadas (borrado de aristas, búsqueda de vecinos, o consultas de rango), son puestos a pruebas en dichas operaciones. El Hito 2 se divide en las siguientes sub tareas:

1. **Examinar el código de los k^2-trees:** En la primera etapa del Hito 2, se examinan los códigos del resto de k^2-trees (Dyn-array1, [5] Dyn-array2, [6] HashTrie, [7] Ustatic) [8].
2. **Testing de los k^2-trees:** En esta etapa se desarrollan mecanismos de testeo para comprobar que los k^2-trees implementan de manera correcta las operaciones de la Tabla 2.
3. **Diseñar mecanismos de comparación:** La tercera etapa del Hito 2 es una de las más importantes ya que se debe diseñar una forma de comparar todas las estructuras.
4. **Comparar las estructuras y analizar:** Finalmente, se grafican los resultados de la etapa anterior y se comparan teniendo en cuenta el tiempo ocupado en cada operación y la memoria total de la estructura. De esta etapa se espera elegir el mejor k^2-tree para incorporarlo en el motor de base de datos.

[4] `https://github.com/darroyue/k2-dyn-tries`

[5] `https://github.com/borjaf696/k2AttDyn/blob/master/include/plib/k2TreeDyn.cpp`

[6] No hay una versión publica de esta implementación, pero se le pidió a los autores

[7] `https://github.com/tudocomp/tudocomp/blob/public/include/tudocomp/compressors/lz78/HashTrie.hpp`

[8] `https://github.com/aplf/sdk2tree/tree/master/implementations/dk2tree`

4.3. Hito 3

El último hito se enfoca en reemplazar el k^2-tree que posee Attk2DynTree por aquel elegido en el Hito 2, y finalmente comprobar si se logra una mejora en las consultas mencionadas en la sección 3.1. Para lograr esto, se divide este hito en las siguientes sub tareas:

1. **Examinar el código del motor de base de datos:** Esta etapa se enfoca en estudiar el código de Attk2DynTree [9] para entender cómo utiliza los k^2-trees. También se aprovecha esta etapa para entender cómo funciona el motor de base de datos en general.
2. **Testing del motor de base de datos:** En esta fase se diseña un grafo artificial muy pequeño, que se utiliza para comprobar que la base de datos funciona correctamente.
3. **Reemplazar Dyn-array1 por el nuevo k^2-tree:** Esta es la etapa más importante del Hito 3, y consiste en modificar el código del motor de base de datos para reemplazar Dyn-array1 por el k^2-tree elegido de acuerdo a los experimentos del Hito 2. Es importante mencionar que una vez esté listo el reemplazo, la nueva base de datos debe pasar los mismos tests implementados en la fase anterior.
4. **Big test:** En esta etapa se utilizan los grafos creados a partir de los datos de la sección 2.8.2, para generar tests que pongan a prueba ambas versiones de Attk2DynTree. Se mide la memoria total de la base de datos y el tiempo total en ejecutar varias operaciones del mismo tipo.
5. **Comparar el nuevo motor con el anterior:** Finalmente se grafican y comparan los resultados obtenidos en la etapa anterior, y se comprueba si realmente hubo una mejora. También se hace un análisis sobre los pro y los contras de la nueva implementación.

[9] `https://github.com/borjaf696/k2AttDyn`

5. Agregando borrados a Blocks

En esta sección se muestra el trabajo realizado en el primer hito. Como se menciona en la sección 4.1, el primer paso de esta etapa consiste en buscar, leer y probar la implementación del k^2-tree **Blocks**. Más detalladamente, lo que se hace es leer y comentar el código de la estructura, a medida que se estudia el paper correspondiente. Se implementan varios tests [10] que ponen a prueba las distintas partes del código. Luego, se interviene la estructura para implementar la operación de borrado de aristas, junto con sus tests correspondientes. Este hito termina con una optimización sobre el borrado de aristas [11].

5.1. Blocks

Este k^2-tree es una estructura híbrida, ya que los primeros l niveles están representado de la manera clásica, usando punteros. El resto de niveles están almacenados en bloques que representan tries de aridad k^2 de forma compacta. Cada bloque puede marcar hojas que en realidad son nodos que almacenan sus caminos en otro bloque, a estas hojas se les conoce como nodos frontera. Cada bloque se encarga de guardar las posiciones de estos nodos en un arreglo de fronteras, utilizado para descender de bloque en bloque. El beneficio de tener los tries almacenados de forma compacta es que se reduce el espacio ocupado por la estructura. La manera especifica en que se almacenan en Arroyuelo et al. [2], Puede hacer que la estructura sea *cache friendly*. Pues se cargan en memoria cache los sub tries alojados en memoria contigua, de tal forma que recorrerlos es mucho más rápido.

5.1.1. Estructura de datos

Blocks está implementado de tal forma que por el momento solo soporta k^2-trees con $k = 2$ (aridad 4). Las principales estructuras de Blocks son:

- **trieNode:** Es la estructura que compone los primeros l niveles de Blocks. Esta posee un arreglo de tamaño 4 con punteros a sus 4 posibles hijos (la posición i indica la sub matriz correspondiente). trieNode además posee un puntero a un posible bloque (treeBlock), que solo se activa en el nivel l para hacer la transición a los bloques.
- **treeBlock:** Esta estructura representa un sub k^2-trie mapeado a un arreglo de bits. La topología del k^2-trie se guarda en un arreglo estático de enteros de 16 bits llamado DFUDS (que se reconstruye por fuerza bruta). treeBlock además posee un segundo arreglo llamado Ptr que contiene la frontera del bloque. Es decir, es un arreglo con punteros a otros bloques que descienden del bloque actual. Por último, estos bloques contienen información como tamaño actual, tamaño máximo, número de hijos, etc.
- **DFUDS:** El arreglo que posee treeBlocks se compone de enteros de 16 bits. Cada entero se divide en cuatro grupos de 4 bits, de tal forma que cada grupo contiene la información de un nodo de trie. Por ejemplo si un grupo tiene la codificación 1001, significa que el primer y último hijo de ese nodo contienen información. De esta forma todo nodo puede ser representado por una tupla (i, j), donde i corresponde al entero de 16 bits en DFUDS, y j corresponde al grupo de 4 bits de ese entero. A esta tupla se le conoce como **treeNode**.

[10] `https://github.com/daviddelapuente/k2DynTriesV2/tree/master/tests`
[11] `https://github.com/daviddelapuente/k2DynTriesV2/blob/master/treeBlock.c`

5.1.2. Funciones más importantes

Blocks posee una serie de funciones y métodos que le permiten realizar la inserción y búsqueda de aristas. Al ser una estructura híbrida, hay un kit de funciones para los trieNodes y otro kit para los treeBlocks. A continuación se muestran las funciones más importantes que posee la estructura (en cada función, solo se mencionan las variables más importantes):

- **isEdgeTrie(trieNode *t, uint8 *str):** Esta función indica si la arista cuyo código Morton corresponde al string str, está en Blocks. La función consume los primeros l caracteres de str usando el trie de punteros. Esto es, por cada str$[i] \in [0, 3]$, se accede al hijo str$[i]$ del nodo actual t. Si dicho hijo no existe, entonces retorna falso. Cuando llega al nivel l pregunta si el bloque existe. De ser así, sigue buscando la arista en el bloque, tal como se explica en Arroyuelo et al. [2]. En cualquier otro caso retorna falso.
- **isEdge(treeBlock *root, uint8 *str, uint64 length):** Esta función es la que continúa, después de isEdgeTrie, buscando la arista en el bloque. Para esto, consume los caracteres restantes de str. En cada iteración, se usa el método child del treeBlock. Este método nos entrega la posición en DFUDS del siguiente nodo del trie, tambien nos avisa si el nodo no existe (en cuyo caso retorna falso), o si el nodo es frontera (en ese caso nos entrega un *flag* con la posición del puntero en el arreglo de fronteras Ptr, y cambiamos de bloque). Si se logra consumir todo el string con éxito, retorna verdadero.
- **insertTrie(trieNode *t, uint8 *str):** Esta función es muy parecida a isEdgeTrie. Consume los primeros l caracteres de str, preguntando si el hijo $str[i]$ de t existe. Si aquel hijo no existe, lo crea en la posición str$[i]$, y continúa iterando desde ese nuevo hijo. Cuando llega el nivel l, pregunta si el bloque existe. De ser así, sigue la inserción en el bloque. En caso contrario, crea el bloque y sigue la inserción en este nuevo bloque.
- **insert(treeNode node, uint8 str[]):** Este método de treeBlock se encarga de insertar el resto de str en los bloques. Para esto, se recorren los bloques usando el método child. En el caso de que child determine que el nodo que se busca no existe, se crea un arreglo de bits contiguo que contiene el camino de nodos que falta (en DFS) y se inserta en DFUDS en la posición entregada por child. Al momento de insertar nodos en un bloque, puede ocurrir que DFUDS rebalse. En este caso, lo que se hace es realocar la memoria en un arreglo más grande. Todo bloque tiene un tamaño máximo que puede alcanzar, si este tamaño es alcanzado, se divide el bloque en dos, de tal manera que un bloque es hijo del otro. Esta división se realiza minimizando los recorrido DFS.

5.2. Tests para Blocks

Luego de estudiar la estructura, se implementan una serie tests unitarios que comprueban el buen funcionamiento de Blocks. La idea de estos tests es que funcionen cada vez que se modifique el código de la estructura. Los distintos tipos de test se hacen sobre:

- **Tablas pre computadas:** Estas tablas se usan para recorrer DFUDS. Se comprueba que sus cálculos son correctos.
- **treeNode:** Se comprueba que los treeNodes contengan la información necesaria para acceder a DFUDS.
- **DFUDS:** Se testean las funciones que achican o aumentan el tamaño de DFUDS.

- **inserciones:** Para comprobar las inserciones se crean varios tests, donde se insertan nodos con insertTrie, y luego se buscan con isEdgeTrie.

5.2.1. Materialize suite

Se desarrolla un kit de herramientas que permiten visualizar Blocks [12]. Este kit contiene una serie de funciones que guardan el k^2-tree en un directorio que se ve de la siguiente forma. Por cada trieNode se guarda una carpeta que contiene un archivo con 4 bits, cada bit encendido representa un hijo del trieNode. Por cada hijo, se guarda una nueva carpeta, con la información del hijo. En el caso de los treeBlock, se guarda un archivo que contiene DFUDS, Ptr, el número de nodos actual y el tamaño del bloque. Por cada nodo de la frontera se guarda una carpeta con la información del bloque hijo. Esta suite se usa para poder visualizar cómo va evolucionando el k^2-tree a medida que se le insertan o borran aristas.

5.3. Evolución de tiempo y memoria en la inserción de aristas

Aparte de los test unitarios, se hace necesario un test general, que ponga aprueba la estructura, en términos de memoria y tiempo. Se usa el dataset indochina-2004 presentado en la Tabla 2, el cual contiene aproximadamente 194 millones de aristas.

5.3.1. Configuración del test

Es importante mencionar que Blocks tiene una serie de parámetros que determinan el tamaño de los bloques de treeBlock. Estos parámetros son: l = nivel hasta donde los nodos de tries son de la forma clásica, (d_1, d_2,d_3) = niveles hasta donde los bloques tienen cierto tamaño máximo, N_1 = tamaño mínimo de los bloques, N_t = tamaño máximo de los bloques, α = porcentaje de llenado (α determina cuánto crecen los bloques, de tal manera que el espacio no ocupado sea a lo más $1-\alpha$). Para este experimento se ocupa: $l = 8$, d_1=8, d_2=16, $N_1 = 4$, $N_t = 64$ entre los niveles $[l, d_1]$, $N_t = 128$ entre los niveles $[d_1+1, d_2]$, $N_t = 1024$ para el resto de niveles y $\alpha = 0{,}99$. Se ocupan estos valores porque se usan en los experimentos de Arroyuelo et al. [2].

En cuanto al experimento, se insertan 194.109.311 aristas. Se mantiene una variable a la que se le suman el tiempo transcurrido en cada inserción. Cada 5 millones de aristas insertadas, se anota el tiempo transcurrido y la memoria total de la estructura (si a esas mediciones se les divide por el número de aristas que han sido insertadas, se obtiene la memoria y tiempo de inserción por arista). También, se crea un test que inserta la misma cantidad de aristas, pero tras cada inserción se busca la arista inmediatamente, guardando el tiempo de búsqueda cada 5 millones de inserciones. Existe otra versión de este experimento que se encarga de testear que las inserciones funcionen de manera correcta. En ese test, se inserta la misma cantidad de aristas, pero antes de insertar, el programa se fija en que la arista no esté, y luego de insertar se fija en que la arista sí esté.

Los experimentos se corren en una máquina con 4 procesadores AMD FX-9800P RADEON R7 y 8GB de RAM. La máquina corre Ubuntu 18.04.5. Los códigos están implementados en C/C++ y se compilan con g++ 7.5.0 usando la optimización -O9.

[12] `https://github.com/daviddelapuente/k2DynTriesV2/blob/master/materializeSuite.c`

5.3.2. Resultados

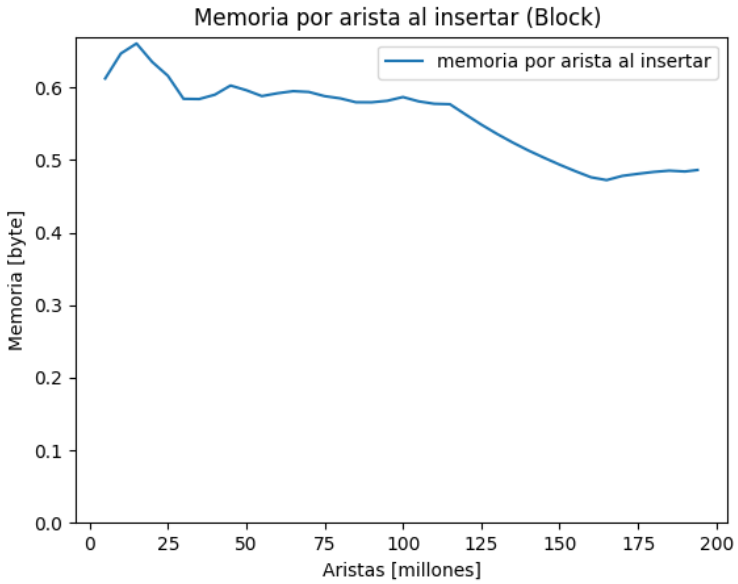

Figura 4: Memoria ocupada por cada arista en Block, en función del número de aristas

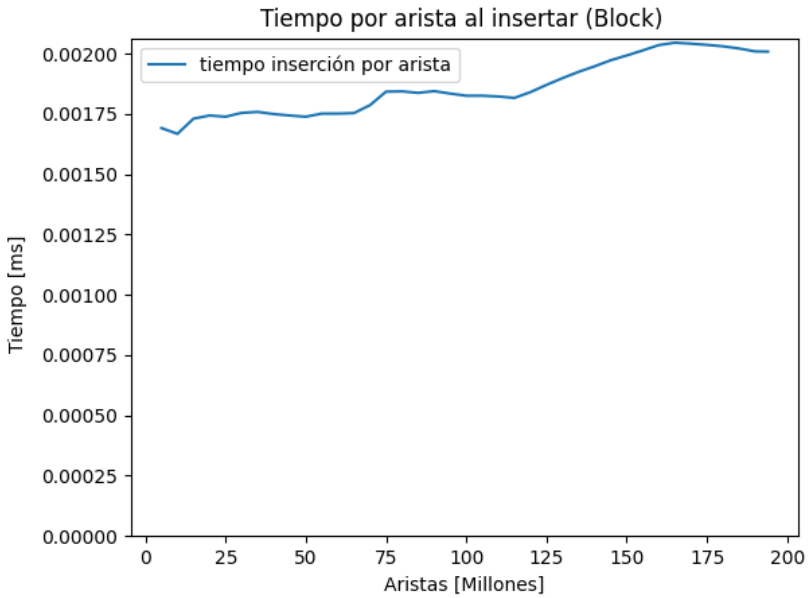

Figura 5: Tiempo de inserción promedio en Block, en función del número de aristas

El teorema 1 de Arroyuelo et al. [2], dice que la memoria total de este k^2-tree es del orden de $O(p\log(n^d/p) + pd\log(k))$, donde p es el número de aristas, d es la dimensión del tensor (en este caso es 2, porque se trabaja con matrices), n es el número de nodos en el grafo, y k^2 es la aridad del trie. Si se divide la fórmula anterior por p se obtiene que el tamaño por arista es del orden $O(\log(n^d/p) + d\log(k))$, que decrece a medida que aumenta p. En la Figura 4 se observa que en promedio, la memoria que ocupa cada arista decrece a medida que aumenta el número de aristas, lo que se corresponde con el análisis anterior.

En la Figura 5 se observa cómo aumenta levemente el tiempo promedio al insertar aristas, a medida que blocks aumenta de tamaño. Esto es de esperarse ya que, recorrer un bloque toma tiempo $O(N_t)$, pero N_t se elige para que sea del orden $O(\log^2(N))$[2], donde N es el número de nodos de trie. De esta forma, en un caso muy extremo, si cada bloque solo tuviera un nodo de trie, cada operación (inserción o búsqueda) es del orden de $\Theta(\log(n)\log^2(N))$. Aunque este resultado teórico está lejos de lo practico, pues un bloque suele contener varios nodos de trie.

5.4. Borrado de aristas

Para borrar una arista, se recorre el k^2-tree como si se estuviera buscando la arista. Es decir, vamos consumiendo el string y bajando por el trie (si en algún momento no hay camino, la función termina, pues no hay camino que borrar). Este recorrido se hace siguiendo una estrategia *top-down*, de tal manera que, cada vez que se baja un nivel en el trie, se guarda el nodo y el camino elegido en un stack. Esto se hace porque, una vez encontrada la arista del grafo, se debe recorrer el camino de vuelta, borrando los nodos correspondientes. Esto último se hace siguiendo una estrategia *bottom-up* para desapilar el stack. Cada vez que se desapila un nodo del stack, se le borra el hijo correspondiente, si el nodo se queda sin hijos, se borra y continuamos desapilando. Si al nodo aun le quedan hijos, la función termina. Como la estructura es híbrida, se crea una función para borrar trieNodes y otra para borrar treeNodes:

- **deleteTrie(trieNode *t, uint8 *str):** Esta es la función que borra trieNodes y se divide en dos etapas. En la primera, se consumen los primeros l caracteres de str y se apilan en un stack junto a sus trieNodes correspondientes (se guardan en stacks separados). Cuando se llega al nivel l, se llama a la función que borra los treeBlocks y se espera a que retorne para empezar siguiente etapa. En esta etapa se desapila el stack de caracteres y el stack de trieNodes, y se borra el hijo del trieNode correspondiente. Si todos los hijos del trieNode son nulos, se borra y se continúa con el siguiente elemento del stack. En cualquier otro caso, la función termina.
- **deleteBlockNodes(treeBlock *root, uint8*str, uint64 length):** Esta es la función que borra treeBlocks, y es muy parecida a la anterior. Se consume el resto de caracteres del string y se apilan en un stack junto a sus treeNodes (las tuplas de DFUDS) correspondientes. Pero además de lo anterior se usan dos stacks más para apilar treeBlocks y nodos frontera si es que pasamos de un bloque a otro. Una vez recorrido el camino, se desapila el stack de bloques y el nodo frontera, por cada bloque en el stack, se desapila el stack de caracteres y el stack de treeNodes. Por cada treeNode desapilado, se apaga un bit en DFUDS (la posición del bit la contiene el treeNode). Si todos los bits de un nodo en DFUDS son 0, se borra este nodo de DFUDS. Es decir, se corre toda la parte derecha de DFUDS, 4 bits a la izquierda (para esta última parte se agrega una optimización que borra todos los nodos que no tienen hijos, en una sola pasada). Si el bloque queda vacío, se elimina de memoria y se usa el nodo frontera del stack, para actualizar la frontera del bloque padre. Por otro lado, si no todos los bits del nodo son 0, entonces la función termina. Esta función retorna un booleano, que le dice a deleteTrie si continuar borrando o no.

5.5. Tests para el borrado de aristas

Una vez implementado el borrado de aristas, se crean varios tests que ponen a prueba el funcionamiento de la nueva operación. Los primeros tests se crean a mano. Se insertan y borran una cantidad pequeña de aristas, de forma que luego de cada operación se observa la forma del k^2-tree usando el kit de visualización (se dice que el test está hecho a mano porque no se hacen *asserts*, solo se observa la estructura). Luego, se crea una segunda tanda de tests[13], donde se usa la función isEdgeTrie antes y despues de borrar aristas, para comprobar que las aristas que se quieren borrar efectivamente fueron borradas, y ninguna otra.

5.6. Evolución de tiempo y memoria en el borrado de aristas

Para poner a prueba el borrado de aristas en términos de memoria y tiempo, se vuelve a usar el dataset indochina-2004. Se crea un gran test[14], en el que primero se insertan las 194 millones de aristas, y luego se borran todas. Durante todo el test, se mantiene una variable a la que se le suma el tiempo transcurrido en cada borrado. Cada 5 millones de aristas borradas, se anota el tiempo transcurrido y la memoria total de la estructura (estas variables se dividen por el número total de aristas para obtener sus valores por aristas). Existe otra versión de este experimento, que se encarga de testear que los borrados funcionen de manera correcta. En esa otra versión del test, se insertan los 194 millones de aristas y luego se borran todas, pero antes de cada borrado el programa chequea que la arista esté, y luego de borrar chequea que la arista no esté. El test se configura con los mismos parámetros de la sección 5.3.1.

5.6.1. Resultados

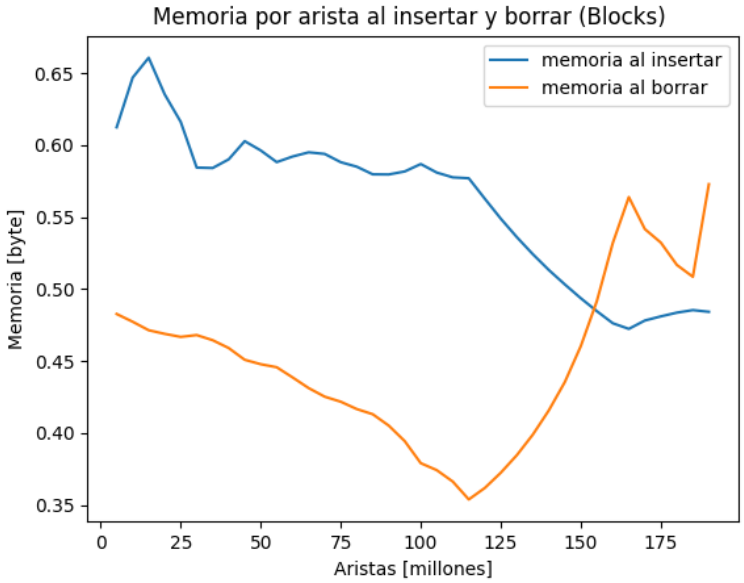

Figura 6: Memoria por arista de la estructura al insertar y borrar

[13] https://github.com/daviddelapuente/k2DynTriesV2/blob/master/tests/deleteionTests/testDeletion.c

[14] https://github.com/daviddelapuente/k2DynTriesV2/blob/master/tests/TestBorrarArbolCada5M/delTrie1/del1TrieCada5M.c

En el gráfico de la Figura 6, se observa el comportamiento de la memoria por arista de la estructura al insertar y borrar las 194 millones de aristas. El eje x representa cuántas aristas se han insertado, mientras que el eje y representa la memoria promedio que ocupa cada arista. El gráfico de color azul es el mismo que el de la Figura 4, mientras que el gráfico de color naranja corresponde a la memoria por arista al borrar, y muestra que después de las 100 millones de aristas borradas la memoria ocupada por aristas empieza a aumentar. Nuevamente esto último se corresponde con el teorema 1 de Arroyuelo et al. [2].

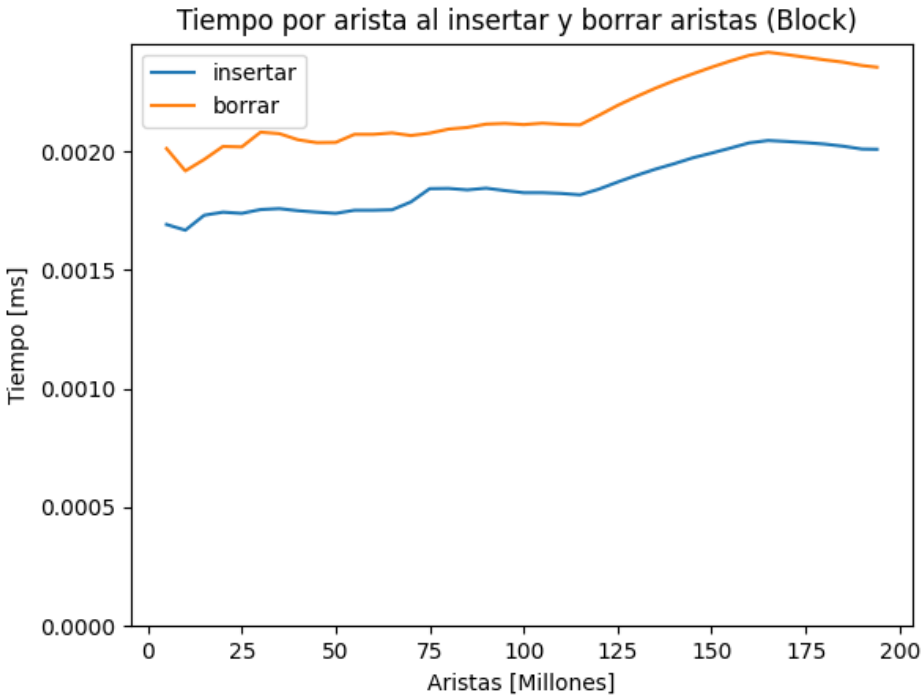

Figura 7: Tiempo de inserción y borrado, por arista

En la Figura 7 se observa el tiempo promedio de inserción y borrado de aristas. El eje x corresponde al número de aristas insertadas/borradas, mientras que el eje y corresponde al tiempo promedio de cada operación. En particular, para el borrado de aristas se observa que el tiempo crece levemente. Esto se debe a que a que en cada borrado se recorre un camino del k^2-tree dos veces (de ida y vuelta), por lo que las operaciones de borrado, al igual que las de inserción, toman tiempo $\Theta(\log(n)\log^2(N))$ (nuevamente, este resultado teórico está lejos de lo práctico, pero indican que el tiempo no es constante). Otra cosa que se puede notar en esta figura es que el borrado de aristas se demora más que la inserción. Esto se debe principalmente a que la inserción recorre un camino solo de ida, mientras que el borrado lo hace de ida y vuelta. Por último, es importante mencionar que al medir el tiempo transcurrido cada 5 millones de aristas borradas, el gráfico de color naranja adquiere un comportamiento creciente, pero debido a que el grafo tiene cada vez menos aristas el crecimiento es cada vez menor.

5.7. Optimizando el borrado de aristas

Un problema que tiene el borrado de aristas propuesto en la seccion anterior, es que no se preocupa de asegurar que los bloques tengan siempre un tamaño mínimo. Por ejemplo, podrían borrarse varias aristas de tal manera que algún bloque quede con un solo nodo de trie. Luego, si se borra una arista que involucre a ese nodo, el bloque quedaría vacío y se liberaría su memoria. Pero, si luego se inserta la misma arista que hace aparecer ese nodo, se crearía de nuevo el bloque con solo ese nodo. Finalmente, si repetimos la operación de agregar y quitar esa arista, no solo se repetiría la operación de insertar y borrar aquel nodo, si no que además se gastaría tiempo en pedir y liberar la memoria del bloque. Para amortiguar el caso anterior, lo que se hace es procurar que los bloques siempre tengan un tamaño mínimo al borrarles nodos de trie (excepto en el caso de que haya un solo bloque). Para esto, si al borrar nodos de trie, un bloque queda con menos nodos que el tamaño mínimo, se fusiona con alguno de sus parientes.

5.7.1. Parámetros de la optimización

Hay por lo menos dos variables fundamentales que influyen en el resultado de la optimización. Por un lado tenemos el parámetro β, que representa el porcentaje de tamaño mínimo que puede tener un bloque antes de fusionarse. La otra variables es la función que indica con quién se fusiona el bloque (se puede fusionar con alguno de sus hijos o con su padre). La fusión con cada pariente del bloque puede generar resultados diferentes, ya que afecta la forma en que se recorren los bloques (en DFS). Para propósitos de este trabajo se usa $\beta = 0{,}0625$ (el tamaño mínimo por bloque es de 64 nodos), y se decide que el bloque se fusiona con su padre. Por el momento no se ha investigado una forma de optimizar estas variables, pero puede hacerse en un trabajo futuro.

5.7.2. Algoritmo para fusionar bloques

Se implementa una segunda función para el borrado de treeBlocks. Esta función es muy parecida a la versión no optimizada, con la diferencia de que se preocupa de que el bloque al que se le están borrando nodos siempre tenga al menos 64 nodos. Si en algún momento ese bloque llega a bajar de ese límite, se fusiona con su padre (siempre y cuando el padre tenga espacio). Luego se continúa borrando el camino del padre. Las nuevas funciones creadas son:

- **unionBlocks(treeBlock * father, treeBlock * son, uint16 flag, uint16 flagIdx):** Esta función une a los treeBlock's *father* y *son*. Primero, se inserta el DFUDS de *son* en el de *father*. Luego, se inserta la frontera de *son* en la de *father*, y se actualizan las nuevas posiciones de los punteros (pues *son* ya no es parte de la frontera de *father*, y *father* ahora tiene los punteros de *son*). Finalmente se libera la memoria ocupada por *son*.
- **deleteBlockNodes2(treeBlock *root, uint8* str, uint64 length, uint16 level, uint64 md):** Esta es la versión optimizada de deleteBlockNodes. Cada vez que se desapila un nodo del stack para borrarlo, se pregunta si el bloque tiene menos de 64 nodos. De ser así, se llama a unionBlocks (solo si es que el padre no alcanza su tamaño máximo) para unir al bloque con su padre. Y se continúa el borrado en el padre actualizado. En caso de que el bloque tenga más de 64 nodos, se siguen borrando nodos como en deleteBlockNodes.

5.8. Tests para la optimización del borrado de aristas

Después de implementar la optimización, se utilizan los mismos tests usados en el borrado simple para probar el borrado con uniones. Además se agregan unos pequeños tests para verificar que los bloques efectivamente se están uniendo (estos últimos test se hacen de manera manual, utilizando las herramientas de visualización).

5.9. Evolución de tiempo y memoria en el nuevo borrado de aristas

Para poner a prueba el nuevo borrado de aristas, en términos de memoria y tiempo, se reutiliza el mismo experimento de la sección 5.6, pero esta vez se usa el nuevo borrado de aristas[15]. El k^2-tree se configura con $\beta = 0{,}0625$. Los tests corren en la misma máquina de la sección 5.3.1.

5.9.1. Resultados

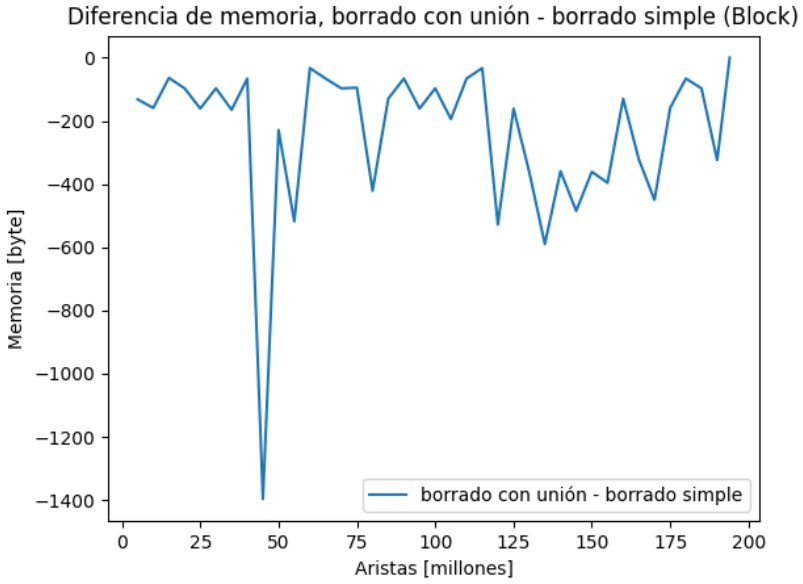

Figura 8: Diferencia de memoria ocupada entre el borrado con unión y el borrado simple

Al usar el borrado optimizado, la evolución de la memoria por arista se parece mucho al gráfico naranja de la Figura 6. Es decir, si se habla de memoria, no hay mucha diferencia entre usar un borrado simple u optimizado. Es por esto que en la Figura 8 se decide mostrar la diferencia de memoria al usar estos dos tipos de borrado. Más detalladamente, cada 5 millones de aristas, se grafica la memoria total ocupada al borrar con optimización menos la memoria total ocupada al borrar sin optimización. Se obtiene siempre una diferencia negativa de unos cientos de bytes. Es decir, el nuevo borrado de aristas administra mejor la memoria ocupada, pero la diferencia no es significativa.

[15] `https://github.com/daviddelapuente/k2DynTriesV2/blob/master/tests/TestBorrarArbolCada5M/delTrie2/del2TrieCada5M.c`

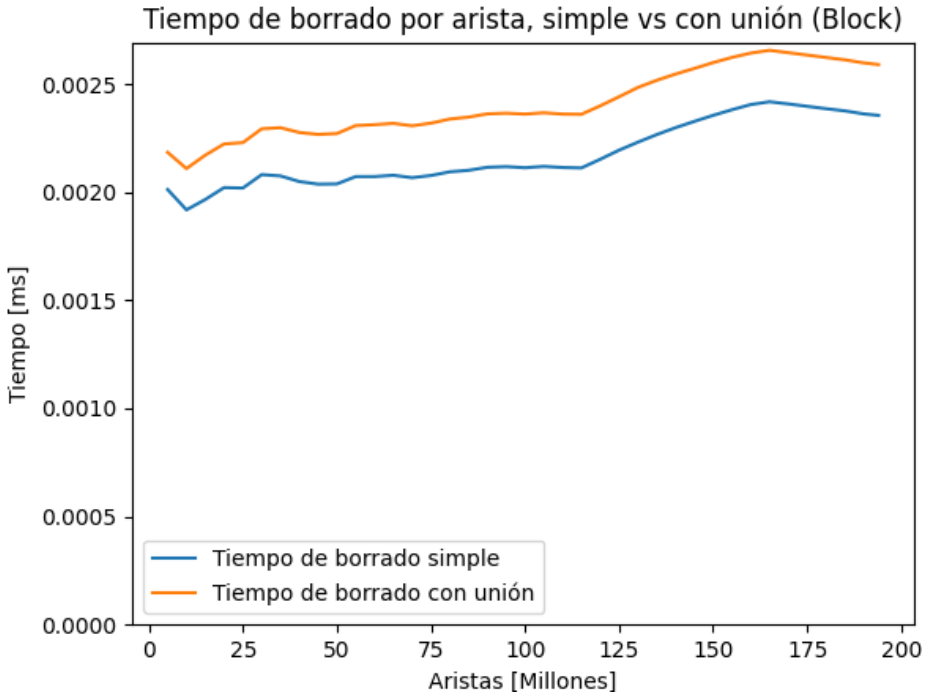

Figura 9: Tiempo de borrado por arista, borrado simple vs borrado con unión

En la Figura 9 se comparan los tiempos de borrado para el algoritmo de borrado simple y el algoritmo de borrado que une los bloques. La versión optimizada para el borrado de aristas tiene un comportamiento muy similar a la versión simple. Aunque se demora más en promedio. Esto último es algo esperable, ya que se invierte tiempo en unir bloques, para amortiguar futuras inserciones.

5.10. Inserciones después de borrar

La finalidad de implementar un algoritmo que une bloques al borrar aristas es poder amortiguar futuras inserciones. Es por esto que se decide crear un test que ponga a prueba esta amortiguación. En este test se insertan 194 millones de aristas, luego se borra el 75 % del k^2-tree, y por último se insertan las aristas borradas. Se mantiene una variable a la que se le suma el tiempo transcurrido en cada reinserción. Se mide el tiempo transcurrido y la memoria total de la estructura, cada 5 millones de aristas reinsertadas (dividiendo por el número de aristas se obtiene también, la memoria y tiempo de inserción por arista). Este test tiene la misma configuración que el test anterior.

5.10.1. Resultados

En cuanto a la memoria por arista, al insertar el 75 % de aristas borradas, los resultados son muy parecidos a los de la Figura 8. Es decir, en cuanto a memoria, no hay gran diferencia entre usar un algoritmo de borrado simple o con unión de bloques. Se obtiene que al unir bloques se administra mejor la memoria aunque, no hay una diferencia significativa.

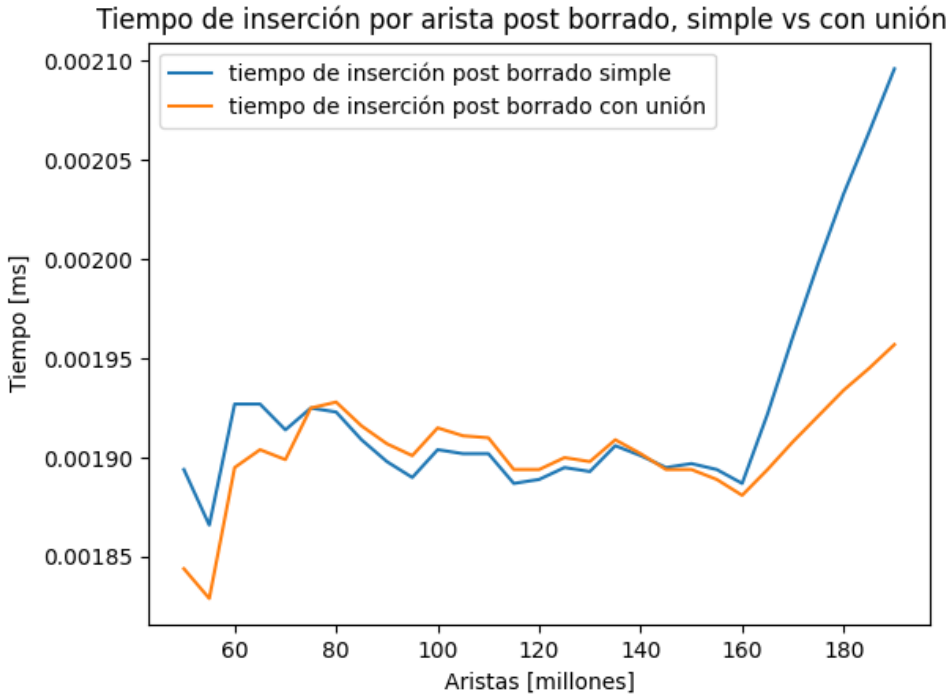

Figura 10: Tiempo de inserción post borrado (simple vs con unión)

En la Figura 10 se observa que al principio el tiempo de reinserción por arista es muy parecido en ambos gráficos, sin embargo al llegar a las 160 millones de aristas reinsertadas, el gráfico naranja (reinsertar después de borrar con unión) muestra ser más eficiente.

5.11. Análisis general de los resultados del Hito 1

Para este hito, la implementación del borrado de aristas fue la parte que más tiempo tomó desarrollar, pues no es fácil crear un algoritmo sobre un código que no es propio. De todas formas los resultados para el borrado de nodos son los esperados:

1. La memoria ocupada por arista aumenta al borrar.
2. El tiempo de borrado por arista, aumenta levemente a medida que aumenta el número de aristas borradas.
3. El tiempo de las dos versiones de borrado de aristas es similar. La versión optimizada es levemente más lenta.
4. La versión optimizada del algoritmo de borrado gestiona mejor la memoria de la estructura que la versión simple, aunque no hay una gran diferencia de memoria.
5. La versión optimizada del algoritmo de borrado muestra su eficiencia al insertar nodos después de borrar.

Se decide que la versión optimizada del borrado de aristas será la ocupada en las siguientes secciones, pues tiene un mejor rendimiento al reinsertar nodos, lo que es una característica deseable para un motor de base de datos. Es importante mencionar que para este hito se realizaron más experimentos y mediciones que las que se mencionan, todos los resultados se encuentran en el link[16] al pie de la página.

[16] `https://github.com/daviddelapuente/apendiceDeResultadosMemoria/blob/main/`

6. Alternativas de k^2-trees

En esta sección se muestra el trabajo realizado en el segundo hito. Como se menciona en la sección 4.2, el primer paso de esta etapa consiste en buscar, leer y probar la implementación del resto de los k^2-trees: **Dyn-array1**, **Dyn-array2**, **HashTrie** y **Ustatic**. Más detalladamente, lo que se hace es estudiar los papers correspondientes, a medida que se prueba el código de las estructuras. Por cada estructura se implementan varios tests de gran tamaño, que comprueban el correcto funcionamiento de sus funciones más importantes, y ponen a prueba las estructuras en términos de memoria y tiempo. Además, se implementan las funciones de búsqueda de vecinos y consultas de rango para **Blocks**. Este hito se termina comparando los resultados de todas las estructuras, y eligiendo la mejor para ser incorporada en Attk2DynTree. A continuación se explica brevemente cómo se componen, y cómo funcionan el resto de los k^2-trees.

6.1. Dyn-array1

Esta versión de k^2-tree es la que se usa en la primera versión de Attk2DynTree. Esta estructura basa su implementación en arreglos de bits dinámicos. La idea es guardar los nodos del k^2-tree en un arreglo de bits dinámico, siguiendo una estrategia LOUDS.

6.1.1. Estructuras de datos

Dyn-array1 está implementado de tal forma que puede soportar cualquier k para la aridad del árbol. Para los arreglos de bits dinámicos, se utiliza la librería de Nicola Prezza [15], la que contiene arreglos de bits dinámicos basados en sumas parciales. Una de las desventajas que posee está implementación es que la versión de la librería utilizada no soporta borrados para las sumas parciales, por lo que Dyn-array1 tampoco soporta borrados.

- **T:** Es un arreglo de bits dinámico que contiene todos los nodos internos del k^2-tree, recorriendo el árbol en LOUDS (es decir, primero se coloca la raíz, luego los nodos del segundo nivel, luego los del tercero, y así sucesivamente).
- **L:** Es un arreglo dinámico que contiene todas las hojas del k^2-tree, recorridas también en LOUDS.

6.1.2. Funciones más importantes

Dyn-array1 soporta inserciones, búsqueda de aristas, de vecinos y de rango:

- **insert(int edgeId, int param[2]):** Para insertar una arista, se recibe la tupla param, que contiene la fila y la columna de la arista en la matriz de adyacencia. Se recorre T buscando el primer nodo del camino que no aparezca. Luego, los nodos restantes del camino se insertan en T en su nivel correspondiente (no se insertan contiguamente). El último nodo se inserta en L.

`resultadosExperimentos.pdf`

- **isNeighbor(int node1, int node2):** Para buscar una arista se utiliza la función isNeighbor, que recibe la la fila (node1) y la columna (node2) de la arista en la matriz de adyacencia. Se parte en la raíz, y se calcula cuál sub matriz es candidata a contener la arista (en función de node1, node2, el tamaño de la sub matriz, y el origen). Si aquella sub matriz no existe (tiene un bit 0), retorna falso. En caso contrario, se usa la función rank, para bajar al siguiente hijo, y se repite el proceso recursivamente actualizando el tamaño de la sub matriz y el origen. Si llega al final del camino, retorna verdadero.
- **neighbors(int node):** Esta función retorna todos los nodos (del grafo), que son vecinos de node. Se parte en la raíz, preguntando qué sub matrices intersectan con la fila en ese nivel y tienen su bit encendido. Por cada sub matriz intersectada, se repite la misma pregunta en el siguiente nivel (recordar que en el arreglo de bits, las matrices del mismo nivel están contiguas, por lo que el recorrido por nivel se hace en BFS). Si la sub matriz intersecta y es una hoja, se agrega a la lista de vecinos.
- **rangeQuery(int nodes[2],int connected[2]):** Esta función retorna todas las aristas que se encuentren en el rectángulo determinado por los puntos nodes y connected. Para hacer lo anterior, se recorre cada nivel del k^2-tree, igual que en la función anterior, buscando sub matrices que intersectan el rectángulo. Si la sub matriz intersecta y es una hoja, se agrega al resultado.

6.2. Dyn-array2

Al igual que el k^2-tree anterior, esta estructura basa su implementación en arreglos de bits dinámicos siguiendo una estrategia LOUDS. En este caso, los arreglos de bits se parten en bloques de tamaño B, de tal manera que el arreglo es representado por un árbol balanceado, donde los nodos internos contienen información que ayuda a llegar a las hojas (número de bits, número de unos, tamaño de los bloques, etc), y las hojas son los bloques que particionan el arreglo dinámico. [6].

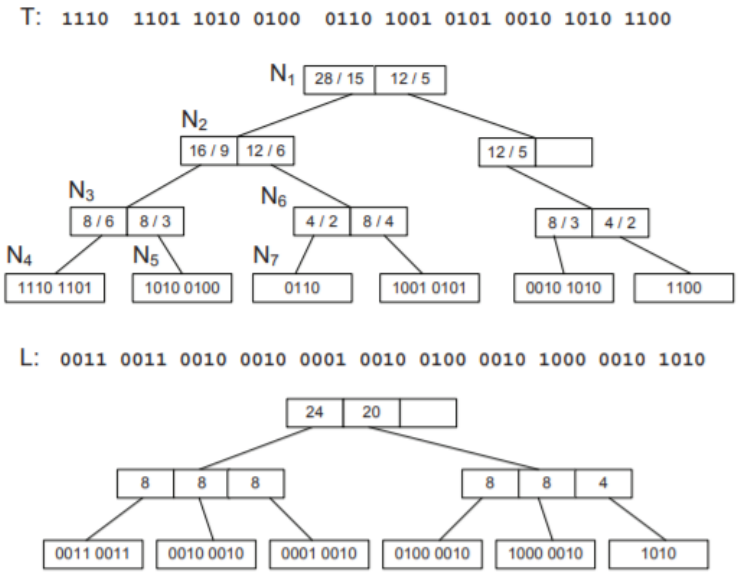

Figura 11: Representación dinámica de los arreglos T y L [6]

6.2.1. Estructuras de datos

- **T:** Se representa con un árbol balanceado que contiene el arreglo de bits de los nodos internos del k^2-tree. Cada nodo interno de T es una tupla $< b, o, p >$, donde p es un puntero al hijo correspondiente, b el número de bits de todas las hojas a las que llega p, y o el número de 1's de todas las hojas a las que llega p. Las hojas de T son bloques de tamaño B que representan la partición del bitmap.
- **L:** también es un árbol balanceado. Representa las hojas del k^2-tree. Cada nodo interno de L es una tupla $< b, p >$, donde p es un puntero al hijo correspondiente y b el número de bits de todas las hojas a las que llega p. No es necesario guardar o, ya que no se hacen operaciones de rank sobre este nivel. Las hojas de L son bloques de tamaño B que representan la partición del bitmap.
- **blockSize:** Constante que determina el tamaño de los bloques que particionan T y L.
- **nTamaños:** Es una constante que determina el número de tamaños distintos que puede tener un bloque. Esto se usa para hacer crecer los bloques si es que se llenan (si se alcanza el tamaño máximo, se dividen).
- **k:** Este k^2-tree usa distintos valores de k según el nivel de profundidad del árbol.

6.2.2. Funciones más importantes

Dyn-array2 soporta inserciones, búsqueda y borrado de aristas. También soporta búsqueda de vecinos y range queries:

- **findEdge(K2Tree * k2tree, uint nodeFrom, uint nodeTo):** Esta función busca la arista (nodeFrom, nodeTo). Para esto, usa una estrategia parecida a la usada en Dyn-array1, pero adaptada al nuevo arreglo dinámico. Para recorrer el k^2-tree, se usan las tuplas $< b, o, p >$, para bajar por T o L hasta llegar a una hoja que contenga en su bloque el primer nodo del camino. Si el nodo existe, se busca la hoja que contiene siguiente nodo, y así sucesivamente hasta llegar al último nivel del k^2-tree. Los autores hablan de una mejora del algoritmo, en donde el próximo nodo en el k^2-tree se parte buscando desde el último bloque y no desde la raíz.
- **insertEdge(K2Tree * k2tree, uint nodeFrom, uint nodeTo):** Para insertar la arista (nodeFrom, nodeTo), se recorre T y L hasta llegar al primer nodo del camino, cuyo hijo no esté en un bloque. Luego, se enciende el bit correspondiente de este nodo, y se busca el bloque que debería contener al hijo. Como el hijo no existe, se insertan k^2 ceros en el nuevo bloque, y se repite el proceso recursivamente. Si estamos buscando el nodo en L, se detiene la función ya que es el último nivel conceptual del k^2-tree.
- **removeEdge(K2Tree * k2tree, uint nodeFrom, uint nodeTo):** Eliminar la arista (nodeFrom, nodeTo) es muy parecido a insertar. Para borrar un hijo de un nodo, se apaga su bit correspondiente. Si un nodo tiene todos sus bits en 0, se elimina del nivel y se continúa borrando. Si queda con algún bit en 1, la función termina.
- **Vecinos y consultas de rango:** Se implementan de manera similar a sus versiones en Dyn-array1. Ambas intentan buscar un área que intersecta la matriz de adyacencia. Se parte desde la raíz del k^2-tree y se itera por cada nivel, buscando los nodos de sub matrices que intersectan el área. Para bajar a los hijos involucrados en una intersección, se utilizan las estructuras T y L de igual manera que en la búsqueda de aristas.

6.3. HashTrie

Un hash trie, es un trie que guarda sus aristas en una tabla de hash. Como un k^2-tree puede ser representado con un trie, se puede usar un hash para representarlo y obtener un k^2-trie. La tabla de hash ocupa memoria del orden de $O(N \log(\sigma))$, donde N es el número de nodos en el trie, y σ el tamaño del diccionario de etiquetas del trie. Al trabajar con k^2-trees, $\sigma = k^2$ y $N = O(p \log_{k^2}(n^2/p))$ donde n es el número de nodos del grafo y p el número de aristas. Es importante mencionar que el hash trie que se usa en este trabajo fue inicialmente pensado para ser usado en la compresión LZ78 [3], por lo que solo posee operaciones de inserción y búsqueda de aristas.

6.3.1. Estructuras de datos

Entre las estructuras de datos más importantes que posee HashTrie, están:

- **H:** Es una tabla de hash de tamaño inicial M y factor de carga α. Esta tabla es la que contiene las aristas del k^2-trie. H[0] es la raíz del trie.
- **D:** Es un arreglo de tamaño M que contiene información sobre las colisiones. Inicialmente todos los valores de D parten en -1.

6.3.2. Funciones más importantes

HashTrie solo posee inserción y búsqueda de aristas:

- **trie.find_or_insert(x, c):** Esta función busca o inserta las aristas que unen los nodos del trie. Cada arista y está identificada con la tupla (x, c), donde x es la posición del padre de y en H y c es la etiqueta de la arista que une x con y. Para insertar (x, c), se usa una función de hash $h(y) = h(x, c)$ que nos da la posición de la arista en H. Si aquella posición no está ocupada (es decir $D[h(x, c)] = -1$), se guarda en $H[h(x, c)]$ el valor de una función $v(x, c)$ [3], y se actualiza $D[h(x, c)] = 0$. Por otro lado, si aquella posición está ocupada, se busca contiguamente la próxima posición desocupada $p = h(x, c)+k$, y se actualizan $H[p] = v(x, c)$ y $D[p] = k$. La función v se usa para saber si $H[h(y)]$ representa realmente al nodo y, o si esa casilla fue ocupada por otro nodo con el mismo valor para la función de hash, en ese caso se prueban casillas consecutivas hasta encontrar aquella que nos da el valor correcto de $v(y)$.
- **Insertar/buscar aristas de la matriz de adyacencia:** HashTrie no tiene funciones para insertar aristas de grafos, por lo que se implementan las inserciones y búsquedas como sigue: Se recibe un string str, que contiene el camino de la arista de grafo en código de Morton. Luego por cada carácter c de str, junto a su nodo padre, se inserta/busca en H el nodo de trie correspondiente. De esta forma, primero se inserta/busca la tupla $y_0 = (H[0], str[0])$, luego se inserta/busca $y_1 = (H[h(y_0)], str[1])$, y así sucesivamente.

6.4. Ustatic

Esta versión de k^2-tree basa su implementación en la unión de k^2-trees estáticos. La estructura posee una lista de adyacencia no comprimida E_0, y un conjunto de k^2-trees estáticos representados por bitmaps (al ser estáticos, las operaciones de búsqueda se pueden hacer eficientemente porque cada rama toma tiempo constante [7]). El número de nodos que ocupa un k^2-tree E_i es $m_i = |E_i|$. Esta estructura basa sus operaciones dinámicas en la reconstrucción de sus k^2-trees estáticos.

6.4.1. Estructuras de datos

Las principales estructuras de datos que componen Ustatic son:

- E_0**:** Corresponde a la lista de adyacencia no comprimida que contiene aristas del grafo. Esta lista está representada por un hash, que inserta aristas (x, y) con una función de hash $h(x, y)$ (no se menciona cómo es la función h, solo importa que usa la arista (x, y) para calcular posiciones en el hash; Se pueden encontrar más detalles en el código [17]).
- **E:** Es un arreglo de tamaño r, donde cada elemento E_i es un k^2-tree estático. Cada E_i tiene un bitmap estático que contiene la codificación del árbol en LOUDS (la concatenación de T con L). Cada E_i puede contener hasta $m_i = m/\log(m)^{2-i\varepsilon}$ aristas, donde m es el número de aristas del grafo (suele usarse r=8 y $\varepsilon = 0{,}25$).

6.4.2. Funciones más importantes

- **check_link(data *dk2t, uint32 x, uint32 y):** Esta función dice si la arista (x, y) está en dk2t. Primero revisa si la arista está en el hash E_0 y luego itera por el resto de los E_i, revisando si está. Para buscar una arista en E_i se recorre el arreglo estático usando operaciones de rank que indica por cual nodo bajar (se cuentan los unos que hay en el nivel hasta la hoja actual, se baja al siguiente nivel y se cuentan los nodos hasta llegar al número de unos contados).
- **add_link(data *dk2t, uint32 x, uint32 y):** Esta función inserta la arista (x, y) en dk2t. Primero busca la arista con check_link, para asegurarse que no está. Suponiendo que no está, si E_0 tiene espacio, se inserta ahí. En caso contrario, se crea un k^2-tree estático que contiene a E_0. Luego se itera E buscando un j tal que $\sum_{k=0}^{j} m_k <= m_j$ y se reconstruye E_j a través de la unión sucesiva de los k^2-trees involucrados.
- **del_link(data *dk2t, uint32 x, uint32 y):** Para borrar una arista de dk2t, se hace lo siguiente: si la arista está en E_0, se borra de ahí. Si no, se busca j tal que $(x, y) \in E_j$. Se busca el bit que representa la arista y se apaga. Si el número de aristas totales borradas llega a ser mayor que $m/\log(\log(m))$ se reconstruyen todos los k^2-trees.
- **list_neighbors(struct data *p, uint32 x):** Primero busca los vecinos de x que están en E_0 usando una lista de adyacencia que posee la estructura. Luego por cada E_i se buscan los vecinos usando una estrategia similar a la búsqueda de vecinos vista en las estructuras anteriores (solo varía en que el bitmap ahora es estático).

[17] `https://github.com/aplf/sdk2tree/blob/master/implementations/sdk2tree/main.c`

6.5. Nuevas funciones para Blocks

Se decide aprovechar el conocimiento adquirido durante el Hito 1 para implementar las funciones que buscan los vecinos de un nodo (de grafo) y las consultas de rango en el k^2-tree Blocks. Es importante mencionar que las implementaciones que se explican a continuación son básicas, en el sentido de que no aprovechan la localidad de DFUDS, y podrían no ser las más eficientes (es decir, podrían optimizarse en un trabajo futuro).

- **getNeighboursTrie(trieNode *t, int nodeid, int rleft,int rright, int cleft,int cright,std::vector<int>&neigs):** Esta función se encarga de retornar los puntos de la matriz que resultan de intersectar el rectángulo formado por la fila completa nodeid, y el rectángulo que representa la matriz completa (rleft= 0, rright= $|V|$, cleft= 0 y cright= $|V|$, donde $|V|$ = número de nodos en el grafo). El resultado se guarda en el arreglo neigs. Para lo anterior, se recorren los primeros l niveles del trie, siguiendo un recorrido DFS. Como se intersecta una fila, en cada nivel se pregunta si la intersección ocurre en las dos sub matrices de arriba o en las de abajo (es decir, se divide el segundo rectángulo en dos, en la dimensión de las filas). Luego se dividen ambos rectángulos en dos partes iguales en la dimensión de las columnas. Se pregunta si los rectángulos de la derecha intersectan entre ellos, y si los de la izquierda intersectan entre ellos. De ser así y si la sub matriz existe, se continúa recursivamente en esos rectángulos. Si se llega al nivel l, se llama a la función que continúa en los bloques.
- **getNeighboursBlock(treeBlock *root, int nodeid, int rleft, int rright, int cleft, int cright, std::vector<int>&neigs):** La búsqueda de vecinos en los bloques sigue la misma lógica anterior, solo que esta vez se usa el método child, que permite navegar los bloques, buscando los treeNodes correspondientes. Si se llega a una sub matriz de tamaño 1, se agrega a la lista de vecinos, y termina esa recursión.
- **rangeQuery(trieNode *t, int r1,int r2,int c1,int c2, int rleft,int rright, int cleft,int cright,std::vector<std::vector<int> >&answer):** La consulta de rango retorna todas las aristas que se encuentren en un determinado rectángulo. Para efectuar esta consulta el procedimiento es similar al de buscar vecinos, solo que ahora en vez de intersectar una fila se intersecta otro rectángulo. El primer rectángulo está compuesto por las filas [r1,r2] y las columnas [c1,c2] (este rectángulo representa la consulta), mientras que el segundo rectángulo está compuesto por las filas [rleft,rright] y las columnas [cleft,cright] (este rectángulo representa toda la matriz). Para obtener la respuesta, se recorre el trie en DFS. En cada nodo de trie se pregunta cuáles sub matrices contienen una intersección de los dos rectángulos y se continúa buscando en esas sub matrices. Se actualizan los rectángulos para que sus límites estén dentro de las sub matrices. Si se llega al nivel l, se llama a la función que continúa en los bloques.
- **rangeQueryBlock(treeBlock *root, int r1,int r2, int c1,int c2, int rleft, int rright, int cleft, int cright, std::vector<std::vector<int> >&answer):** La consulta de rango en los bloques sigue la misma lógica anterior. Los bloques se recorren de igual manera que en getNeighboursBlock. Si se llega a una sub matriz de tamaño 1, se agrega a la lista de aristas que pertenecen al rectángulo que se busca, y termina esa recursión.

6.6. Complejidades recopiladas

A continuación se actualiza la Tabla 1, para agregar las operaciones creadas, se muestran las complejidades de memoria de las estructuras, y las complejidades de tiempo de algunas de las funciones mencionadas en este capítulo.

nombre	insertar	buscar	borrar	vecinos	range	insertar nodo	borrar nodo
Dyn-array1	✓	✓		✓	✓	✓	
Dyn-array2	✓	✓	✓	✓	✓	✓	✓
Block	✓	✓	✓	✓	✓		
HashTrie	✓	✓					
Ustatic	✓	✓	✓	✓		✓	✓

Tabla 3: Tabla de operaciones implementadas en código, actualizada

nombre	Estructura	base	memoria (bits)	paper
Dyn-array1	dyn succinct tree	dyn bitvector	$pk^2 \log_{k^2}(n^2/p)$	[1]
Dyn-array2	dyn succinct tree	dyn bitvector	$pk^2 \log_{k^2}(n^2/p)$	[6]
Blocks	dyn succinct trie	blocks	$O(p \log(n^2/p) + p \log(k)))$	[2]
HashTrie	dyn succinct trie	hash	$O(p \log(n^2/p))$	[3]
Ustatic	$\bigcup$ (succinct tree)	static bit vector	$pk^2 \log_{k^2}(n^2/p) + O(pk^2 \log(\log(n)))$	[7]

Tabla 4: Información básica de los k^2-trees

nombre	insertar	buscar	borrar
Dyn-array1	$O(\log(n) \log_k(n))$	$O(k \log_k(n) \log(n))$	$O(\log(n) \log_k(n))$
Dyn-array2	$O(\log(n) \log_k(n))$	$O(k \log_k(n) \log(\frac{n}{k \log_k(n)}))$	$O(\log(n) \log_k(n))$
Blocks	$O(\log(n) \log^2(N))$	$O(\log(n) \log^2(N))$	$O(\log(n) \log^2(N))$
HashTrie	$O(\log_k(n))$ esperado	$O(\log_k(n))$ esperado	
Ustatic	$O(\log_k(n) \log^{\varepsilon}(p)/\varepsilon)^*$	$O(k \log_k(n)/\varepsilon)$	$O(\log_k(n)/\varepsilon + \log_k(n) \log(\log(p)))^*$

nombre	range $a \times b$
Dyn-array1	$O((a + b + occ\ k \log_k(n)) \log_k(n))$
Dyn-array2	$O((a + b + occ\ k \log_k(n)) \log(\frac{n}{k \log_k(n)}))$
Blocks	$O((a + b + occ \log(n)) \log^2(n))$
HashTrie	
Ustatic	$O((a + b + occ\ k \log_k(n)) \frac{1}{\varepsilon})$

Tabla 5: Complejidades de tiempo de las operaciones

Las tablas 4 y 5 contienen la información principal de los k^2-trees explicados anteriormente, Se usan las siguientes variables: p=|aristas|, k^2=aridad=4, n=|nodos de grafo|, N=|nodos de trie|, $\varepsilon > 0$ (constante para controlar el tamaño de los árboles de Ustatic), $a \times b$ representa un rectángulo de lados a y b, occ = número de aristas que hay en el rectángulo $a \times b$ y $*$ representa una complejidad amortizada. Es importante mencionar que para el cálculo de memoria, todas las complejidades poseen una componente sub lineal que se omite por simplicidad.

7. Comparación de los k^2-trees

En este capítulo, se muestran diversos experimentos realizados para poner a prueba las estructuras del capítulo anterior. Solo se ponen a prueba **Dyn-array2**, **Blocks**, **HashTrie** y **Ustatic**. No se considera a **Dyn-array1**, ya que este k^2-tree es el que está contenido en la primera versión de Attk2DynTree, y se compara más adelante. Es importante mencionar que algunas estructuras no pueden participar de ciertos experimentos, ya que no poseen las operaciones necesarias. Esta sección termina con la eleción del mejor k^2-tree para ser ocupado en Attk2DynTree. Es importante mencionar que, además de todos los experimentos que se muestran en esta sección, existen documentos[18] que poseen aún más experimentos y resultados. No se colocan en este trabajo, porque extenderían el número de paginas, sin embargo se adjunta el link al pie de página.

7.1. Configuración de los experimentos

Los experimentos se corren en una máquina con 4 procesadores AMD FX-9800P RADEON R7 y 8GB de RAM. La máquina corre Ubuntu 18.04.5. Los codigos están implementados en C/C++ y se compilan con g++ 7.5.0 usando la optimización -O9.

7.1.1. Configuración de las estructuras

- **Dyn-array2:** blockSize= 512, nTamaños= 4, $k = 4$ primeros 3 niveles, y $k = 2$ en el resto.
- **Blocks:** idéntico a la sección 5.3.1
- **HashTrie:** M=4 y $\alpha = 0{,}30$.
- **Ustatic:** $r = 8$ **y** $\varepsilon = 0{,}25$

7.2. Insertar 194M aristas

El primer experimento consiste en insertar 194 millones de aristas en cada k^2-tree. Se mantiene una variable a la que se le suma el tiempo transcurrido en cada inserción. Se mide el tiempo de inserción, y la memoria ocupada cada 5 millones de aristas insertadas (esas variables se dividen por el número de aristas insertadas, para obtener los valores por arista respectivos).

En la Figura 12 se puede observar que, en los tres k^2-trees la cantidad de bytes ocupados por aristas decrece a medida que se insertan aristas. En esta misma figura también se observa que no aparece la memoria ocupada por HashTrie, esto se debe a que esa estructura ocupa mucha más memoria que los demás k^2-trees (hasta 25 bytes por arista), de todas formas se analiza su comportamiento en la sección 7.9, donde se muestran otro tipo de resultados. El mejor de los k^2-trees para este item es Ustatic.

[18] `https://github.com/daviddelapuente/apendiceDeResultadosMemoria/blob/main/experimentosTresTristesTreesTrieBased.pdf` y `https://github.com/daviddelapuente/apendiceDeResultadosMemoria/blob/main/experimentosTresTristesTreesTrieBasedParte2.pdf`

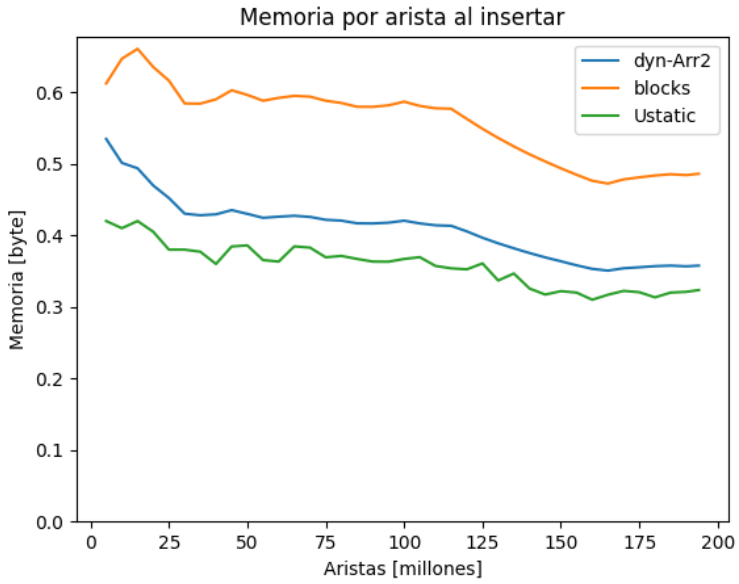

Figura 12: Memoria promedio ocupada, por arista, en función del número de aristas

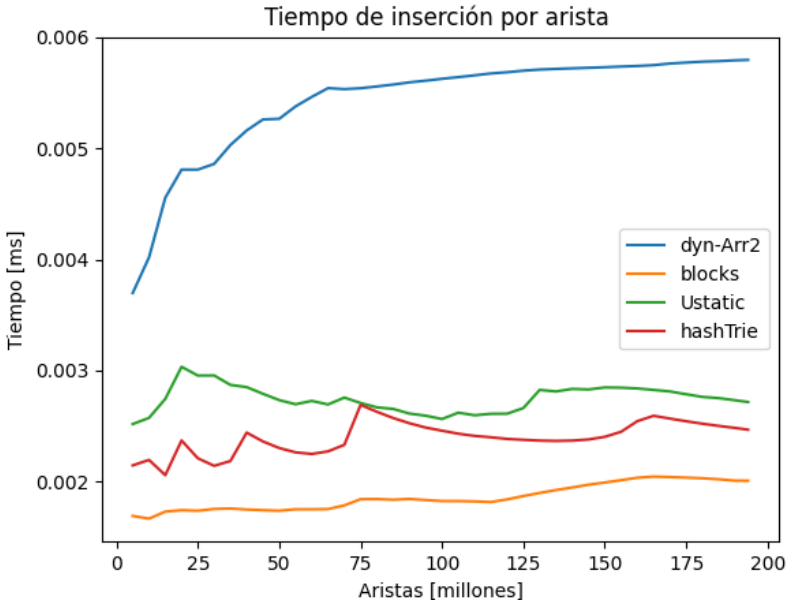

Figura 13: Tiempo promedio de inserción, por arista, en función del número de aristas

En la Figura 13, se observa el tiempo de inserción por arista. A medida que las estructuras aumentan su tamaño, el tiempo aumenta levemente, lo que nuevamente se corresponde con la Tabla 4. Puede observarse que Ustatic y HashTrie tienen comportamientos más irregulares, de hecho se puede observar que cada cierto tiempo las funciones alcanzan unos *peaks* de tiempo. Esto puede deberse a que en esos puntos las estructuras se reconstruyen (Ustatic y HashTrie) o aumentan su tamaño (HashTrie). El k^2-tree ganador de este item es Blocks.

7.3. Buscar 194M aristas

El segundo experimento consiste en insertar y buscar 194 millones de aristas en cada k^2-tree. Cada vez que se inserta una arista, se busca inmediatamente. Se mantiene una variable a la que se le suma el tiempo transcurrido en cada búsqueda. Se mide el tiempo transcurrido cada 5 millones de aristas buscadas (y se divide por el número de aristas buscadas, para obtener los valores por arista).

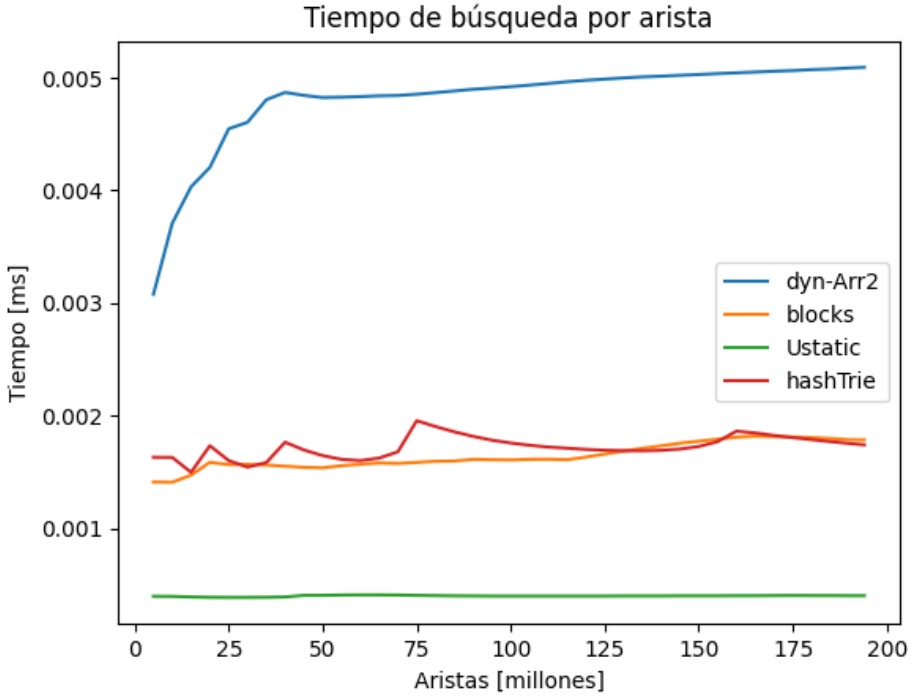

Figura 14: Tiempo promedio de búsqueda de aristas en función del número de aristas

En la Figura 14 se observa que Dyn-array2 y Blocks tienen un comportamiento levemente creciente. Por otro lado, la búsqueda de aristas en HashTrie tiene tiempos muy parecidos a la inserción, esto se debe a que la tabla de hash se va llenando (aumentando el tiempo de búsqueda) hasta que se reconstruye y baja. El ganador de este *item* es Ustatic, pues es mucho más rápido que todos los demás.

7.4. Insertar y borrar 194M aristas

El tercer que experimento consiste en insertar 194 millones de aristas, y luego borrarlas todas. Se mantiene una variable a la que se le suma el tiempo transcurrido en cada borrado. Cada 5 millones de aristas borradas. se anota el tiempo transcurrido y la memoria total de la estructura (junto a sus valores por arista). En este experimento no participa HashTrie, ya que no tiene operaciones de borrado de aristas.

Antes de mostrar la memoria, se muestra el tiempo de borrado de aristas promedio, ya que ocurre algo muy interesante. En la Figura 15, Blocks muestra un comportamiento levemente creciente, Dyn-array2 muestra un comportamiento constante, y Ustatic también muestra un comportamiento levemente creciente, excepto al final donde explota abruptamente.

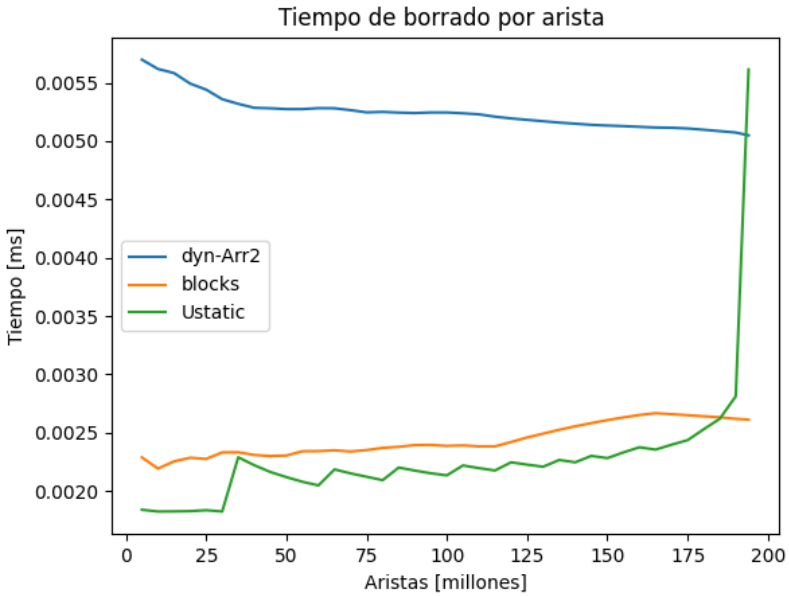

Figura 15: Tiempo promedio de borrado de aristas en función del número de aristas borradas

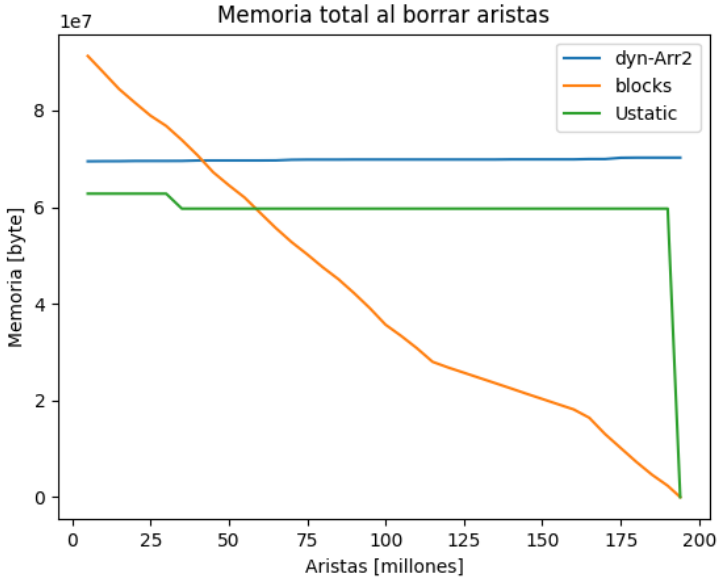

Figura 16: Evolución de la memoria total al borrar todos los nodos

En la Figura 16 se muestra la evolución de la memoria **total** de las estructuras al borrar aristas. Se observa que la memoria de Blocks es la única que disminuye a medida que se van borrando aristas. Dyn-array2 mantiene su memoria constante todo el tiempo. Esto se debe a que la implementación en código apaga los bits de la arista, pero no libera la memoria ocupada. Ustatic por su parte tampoco borra la memoria ocupada hasta que la estructura esté casi vacía (esto explicaría el abrupto incremento de tiempo al final del borrado). Dicho todo lo anterior, el k^2-tree ganador de este experimento es Blocks, ya que es el que mejor gestiona la memoria al borrar.

7.5. Inserciones post borrado de aristas

El cuarto experimento consiste en insertar 194 millones de aristas, luego borrar el 75 % de las aristas, y por último reinsertar todas las aristas borradas. Se mantiene una variable a la que se le suma el tiempo transcurrido en cada reinserción. El tiempo de inserción y la memoria total se miden cada 5 millones de aristas reinsertadas (junto a sus valores por arista). La idea de este experimento es ver cómo se comportan las estructuras cuando se les aplica más de un tipo de operación.

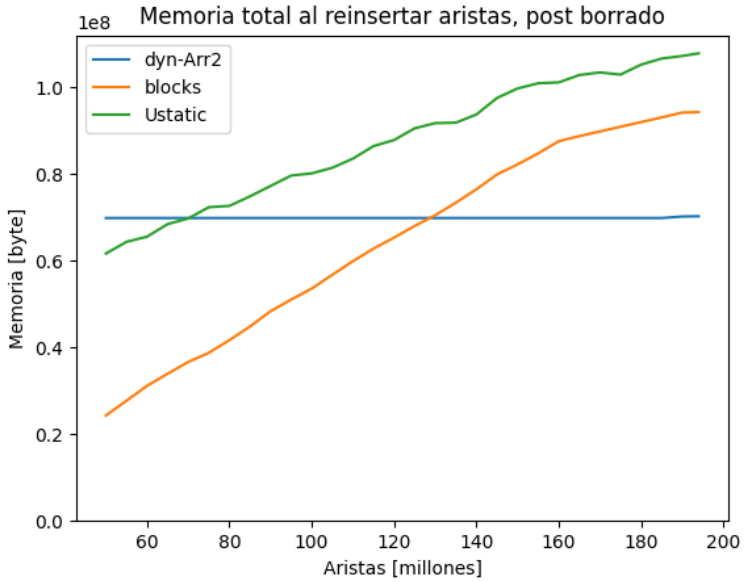

Figura 17: Evolución de la memoria total ocupada al reinsertar todas las aristas borradas.

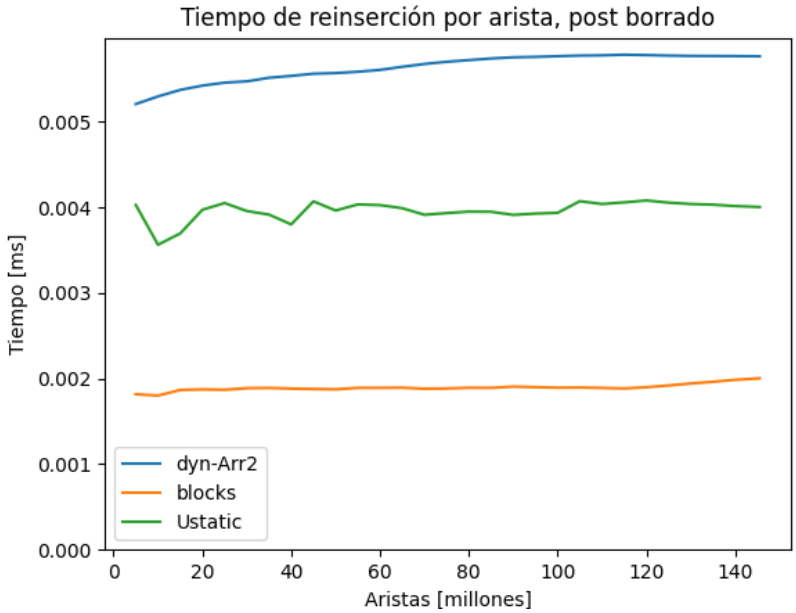

Figura 18: Tiempo de inserción por arista, al reinsertar todas las aristas borradas.

Para Dyn-array2, en la Figura 17 se observa que la memoria total ocupada permanece constante. Esto se debe a que se reutilizan los bits apagados que no fueron borrados. En la Figura 18 se muestra que el tiempo de reinserción es muy parecido a una inserción normal. Esto quiere decir que, para Dyn-array2, el tiempo de inserción no varía si es que ya se han borrado aristas (pero la memoria permanece constante, lo cual no es muy bueno). Para Ustatic, en la Figura 17 observamos que esta vez tiene un peor desempeño que los demás k^2-trees, esto puede deberse a que al borrar el 75 % de las aristas, no se alcanza el umbral necesario para liberar la memoria. Luego, al reinsertar los nodos estos pueden llegar a pedir más memoria para construir nuevos k^2-trees estáticos. Lo anterior claramente afecta el tiempo de inserción. Por último, para Blocks, tanto en la Figura 17 como 18 se observa que mantiene un comportamiento muy similar a la inserción sin borrados, por lo que para este item el k^2-tree ganador es Blocks.

7.6. Oleadas 20-15

El siguiente experimento solo compara Blocks con Ustatic, ya que son las únicas estructuras que realmente liberan memoria de aristas borradas. Este experimento consiste en insertar 20 millones de aristas y luego borrar 15 millones. Luego se insertan los siguientes 20 millones de aristas y se repite el proceso (notar que las aristas insertadas no son las mismas que se borran, es decir son aristas nuevas). La idea de este experimento es poner a prueba las estructuras, en una situación donde reciben ráfagas de una misma operación, y luego reciben ráfagas de otra operación. La memoria ocupada se mide cada vez que se insertan 20 millones o se borran 15 millones de aristas. En cuanto al tiempo, por cada *batch* de 20 millones de aristas, se mide el tiempo promedio de inserción cada 5 millones, y luego se reinicia la variable de tiempo a 0. El resultado es un conjunto de gráficos de 4 puntos cada uno.

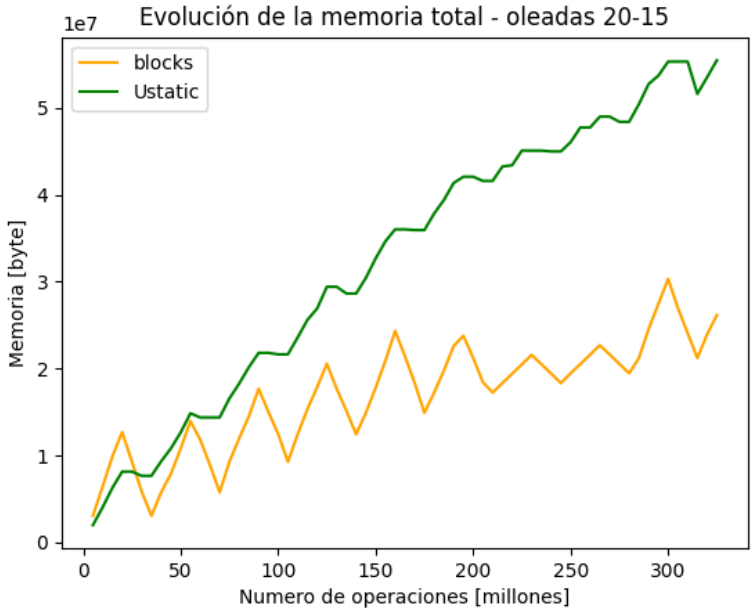

Figura 19: Memoria total tras oleadas de 20M de inserciones y 15M de borrados.

En la Figura 19, para Blocks, se puede ver que la memoria tiene forma de "dientes de tiburón". Este es un comportamiento esperable, ya que la memoria aumenta al insertar 20M de aristas y baja al borrar 15M. Por otro lado, la memoria ocupada por Ustatic empieza a crecer bastante, casi como si no se le borraran aristas. Esto se debe a que las aristas insertadas son siempre nuevas, por lo que Ustatic debe pedir más espacio par almacenarlas, y como no libera memoria al borrar aristas, el tamaño total empieza a aumentar.

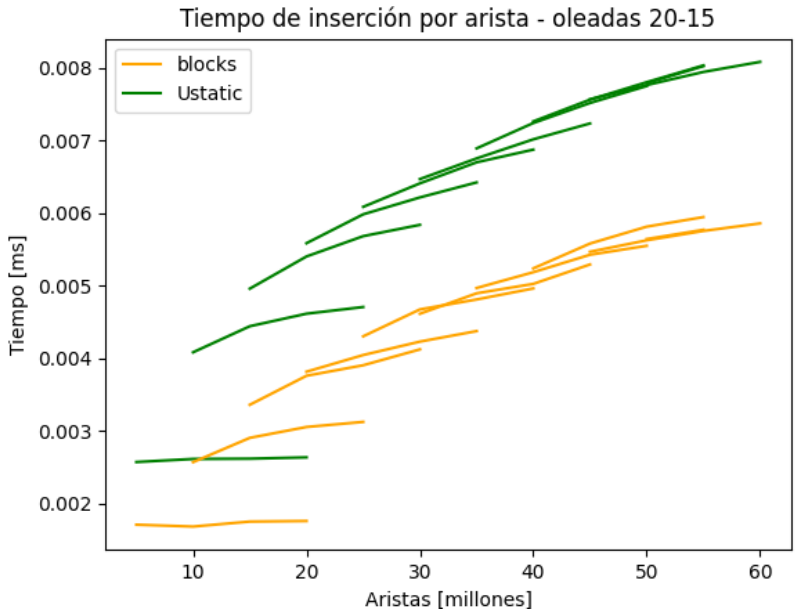

Figura 20: Tiempo de inserción por arista tras oleadas 20-15

En la Figura 20 se observan varios gráficos (uno por cada oleada), donde se muestra el tiempo de inserción de las 20 millones de aristas por oleada. Los gráficos posicionan cada vez más arriba, pues las estructuras van creciendo de a poco (crecen de a 5 millones de aristas). Por otro lado, se nota que ambas estructuras se demoran más que una inserción sin borrados, esto se debe a que al ir insertando y borrando nodos las estructuras se van desgastando, por ejemplo para el primer gráfico amarillo y verde (los de más abajo) el tiempo de inserción es muy parecido al que se muestra en la Figura 13, pero para el resto de gráficos los tiempos se duplican para Blocks y se cuadriplican para Ustatic. En este item gana Blocks en tiempo y memoria.

7.7. Vecinos

En este experimento se pone a prueba la búsqueda de vecinos de un nodo (de grafo) u. Esto es recuperar todas las columnas v, tal que en la matriz de adyacencia $M[u][v] = 1$. Para ver cómo aumenta el tiempo de búsqueda de vecinos, se insertan 190 millones de aristas, y cada 5 millones de aristas insertadas se buscan los vecinos de mil nodos (en cada iteración se buscan siempre los vecinos de los mismos nodos) y se calcula el tiempo promedio.

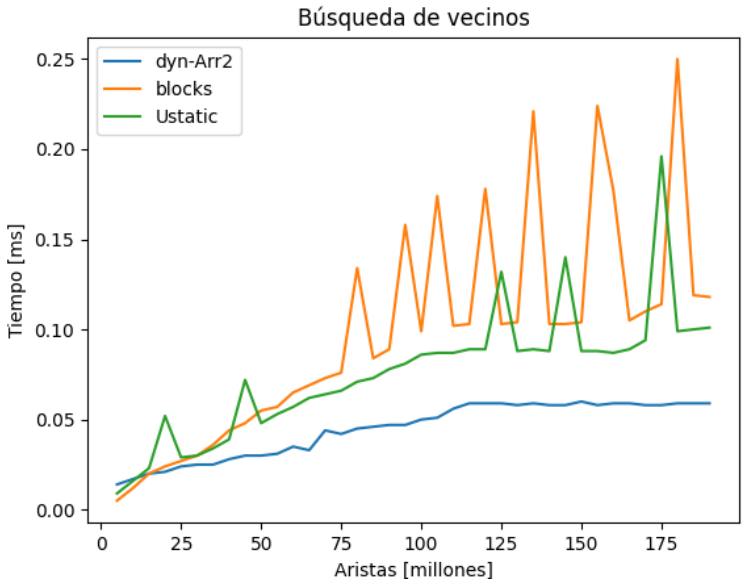

Figura 21: Tiempo de búsqueda de vecinos.

Una búsqueda de los vecinos de u es una *range query* donde el rectángulo es una fila completa. Dicho lo anterior, en la Figura 21 se observa que el tiempo de búsqueda de vecinos, mantiene un comportamiento creciente, esto se corresponde con los resultados de la Tabla 5. En la figura también se observa, que Blocks y Ustatic tienen tiempos muy parecidos, y mas lentos que Dyn-array2. Por lo que en este item, se considera que gana Dyn-array2. Por otro lado, se piensa que el tiempo de Blocks puede ser optimizado (en la sección 6.5 se menciona que la implementación de la operación de búsqueda de vecinos en Blocks es básica).

7.8. Consulta de rango

Una consulta de rango busca todas las aristas de la matriz que están en algún rango delimitado por un rectángulo, formado por las filas ($r1$,$r2$) y las columnas ($c1$,$c2$). La respuesta suele entregarse como otra matriz de tamaño $occ \times 2$ (occ es el número de aristas, y 2 corresponde al número de nodos en la arista). Este experimento, busca poner a prueba la capacidad de realizar consultas de rango sobre los k^2-trees. Se elige el rango $[312824, 412823] \times [0, 7414866]$, ya que es un rango bastante amplio, que contiene muchas de las primeras aristas insertadas. Para realizar las mediciones, se insertan 190 millones de aristas, y se efectúa una *range query* cada 5 millones. Es importante mencionar que Ustatic posee funciones para realizar range queries en los k^2-trees estáticos, pero la API [19] no posee funciones que realicen range queries en el hash. Debido a lo anterior, Ustatic no se considera en este item.

[19] `https://github.com/aplf/sdk2tree/blob/master/implementations/sdk2tree/main.c`

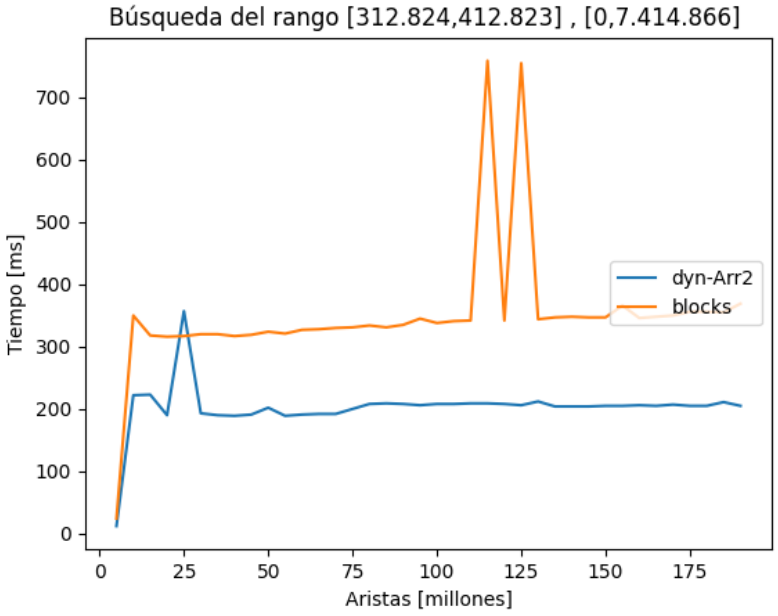

Figura 22: Tiempo de búsqueda de rango.

En la Figura 22 se observa que Dyn-array2 es más rápido que Blocks. Esto puede deberse, nuevamente, a que la implementación de las consultas de rango en Blocks es básica. Por otro lado, al final de cada función vemos que el tiempo crece muy poco (casi permanece constante). Esto puede deberse a que después de un tiempo la respuesta de la consulta retorna las mismas aristas (un futuro experimento interesante puede ser hacer crecer también el rango).

7.9. Otros experimentos

Como se menciona en la sección 7.2, el HashTrie suele tener unos *peaks* de tiempo de inserción. Se dijo que esto se podía deber a que el hash debe ir pidiendo más memoria para guardar más nodos de trie. En la Figura 23 se muestra un experimento que consiste en insertar 194 millones de aristas en el HashTrie. Se observa que la estructura mantiene su memoria constante, y la aumenta cada cierto tiempo. Esto puede ser la causa de los *peaks* del gráfico de la Figura 14 (de hecho pareciera que los *peaks* aparecen en el mismo número de aristas en que aumenta la memoria). En la misma sección 7.2, se menciona que HashTrie ocupa mucha más memoria que las demás estructuras. Esto puede deberse a que es la única estructura que se mide con una herramienta[20] que calcula la memoria total que está usando el programa (se procura medir solo la memoria que ocupa el hash). El resto de las estructuras se miden con sus propias funciones que calculan el número de bytes ocupados.

En la sección 7.4 se menciona que Ustatic no borra la memoria hasta que alcanza cierto umbral. El gráfico de la Figura 24 busca exponer este comportamiento de Ustatic, mostrando el tiempo de borrado por arista para k^2-trees con 10, 100 y 190 millones de aristas. En los 3 gráficos se observa que el tiempo aumenta abruptamente al final. Esto explicaría por qué, al borrar aristas en Ustatic, la memoria se mantiene constante hasta casi el final.

[20] `https://github.com/tudocomp/tudostats`

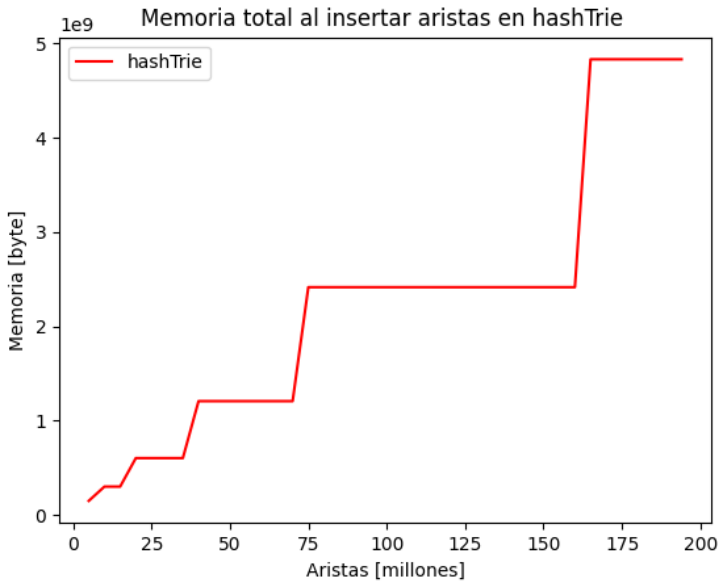

Figura 23: Evolución de la memoria de HashTrie al insertar aristas.

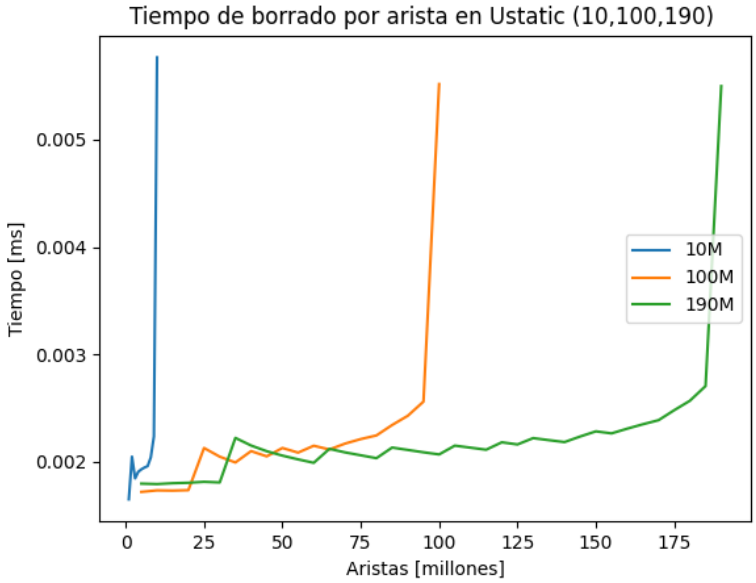

Figura 24: Tiempo de borrado por arista en Ustatic, para k^2-trees de distinto tamaño.

7.10. Elección del k^2-tree

Para elegir el k^2-tree que será usado en Attk2DynTree, es necesario tener en cuenta varios aspectos que el k^2-tree elegido debe satisfacer. Entre los aspectos tenemos:

- Capacidad de borrar. Aunque Attk2DynTree no soporte borrados en este momento, se considera que es una capacidad importante que debería ser incluida a futuro, por lo que es importante que el k^2-tree que soporta el motor soporte borrados eficientemente.
- Velocidad de acceso a aristas. Idealmente nos querríamos acercar a los tiempos de una versión estática del k^2-tree, por razones de desempeño del motor.
- Velocidad de inserción. Agregar aristas debe hacerse de forma eficiente.
- Predictibilidad de tiempos de actualización. Esto quiere decir si es posible estimar cuánto se va a demorar la estructura en realizar una actualización individual. Una varianza muy alta es indeseable en un motor de base de datos.
- Buen uso de memoria. Para un motor que puede estar funcionando en memoria principal durante largos periodos de tiempo es importante que la memoria se gestione eficientemente, incluyendo liberarla adecuadamente cuando ocurren borrados.
- Capacidad de hacer consultas de rango. En Attk2DynTree se realizan consultas de rangos arbitrarios. Por ello es necesario soportar este tipo de consultas.

	Dyn-array2	Blocks	HashTrie	Ustatic
Capacidad de borrar	✓	✓	×	✓
Velocidad de acceso	×	×	×	✓
Velocidad de inserción	×	✓	✓	✓
Predictibilidad de tiempos de actualización	✓	✓	×	×
Gestión de memoria	×	✓	×	×
Consultas de rango	✓	✓	×	×

Tabla 6: Aspectos importantes a considerar para la base de datos

La tabla 6 indica qué versiones de k^2-tree cumplen con qué requisitos. En particular:

- HashTrie es la única estructura que no puede borrar.
- Ustatic es la estructura más rápida para acceder a aristas.
- Blocks, HashTrie y Ustatic son las estructuras más rápidas para hacer inserciones.
- Como las actualizaciones en HashTrie y Ustatic son amortizadas, hay casos muy malos donde una actualización podría tardar mucho. Existen técnicas de desamortización, pero éstas incrementan el uso de la memoria y no están implementadas.
- En cuanto a la gestión de la memoria, Blocks tiene mejor desempeño en general. Aunque ocupe un 28 % más de espacio que otras estructuras, responde mucho mejor que las otras al considerar los borrados.
- Solo Blocks y Dyn-array2 son capaces de hacer consultas de rango.

Dadas las observaciones anteriores, nuestra elección es Blocks, porque es la estructura que cumple con el mayor subconjunto de requisitos. Su única debilidad es que no es tan rápido para hacer operaciones de rango o búsqueda de vecinos como otras estructuras, pero esto no es un problema inhabilitante como lo es el no cumplir con otros requisitos.

8. Integración en el motor de base de datos

Para concretar el último hito del trabajo, se empieza leyendo la documentación y el código[21] del motor de base de datos Attk2DynTree. Más detalladamente, lo que se hace es estudiar el paper correspondiente, a medida que se prueba el código. Se diseñan pequeños grafos, y se guardan en Attk2DynTree, a través de pequeños tests que ponen a prueba el sistema. Luego de los tests pequeños, se inserta el k^2-tree elegido en la sección anterior (**Blocks**), y por último, se crean una serie de tests grandes que ponen aprueba el nuevo prototipo, comprobando si funciona bien y si es más eficiente. Este hito termina con la comparación entre las dos versiones de Attk2DynTree.

8.1. Attk2DynTree

Attk2DynTree es una estructura de datos basada en k^2-trees, que se encarga de guardar multi grafos con atributos de una manera muy compacta. Los multi grafos con atributos son como un multi grafo normal (dos nodos pueden estar conectados por más de una arista), pero los nodos y aristas pueden ser de más de un tipo, y pueden tener atributos. Por ejemplo, en la Figura 25 se muestra un grafo donde hay 2 tipos de nodos: usuarios y películas. Las aristas son *ratings* y *tags* que los usuarios pueden hacer sobre las películas. Cada elemento del grafo tiene sus propios atributos: usuario(u_nombre, u_edad, u_género), película(p_nombre, p_fecha), rating(rate), tag(tag). Cada atributo es una variable que puede tomar algún valor. Por ejemplo el nodo $n2$ toma los valores (Dania, 25, Femenino) para sus atributos.

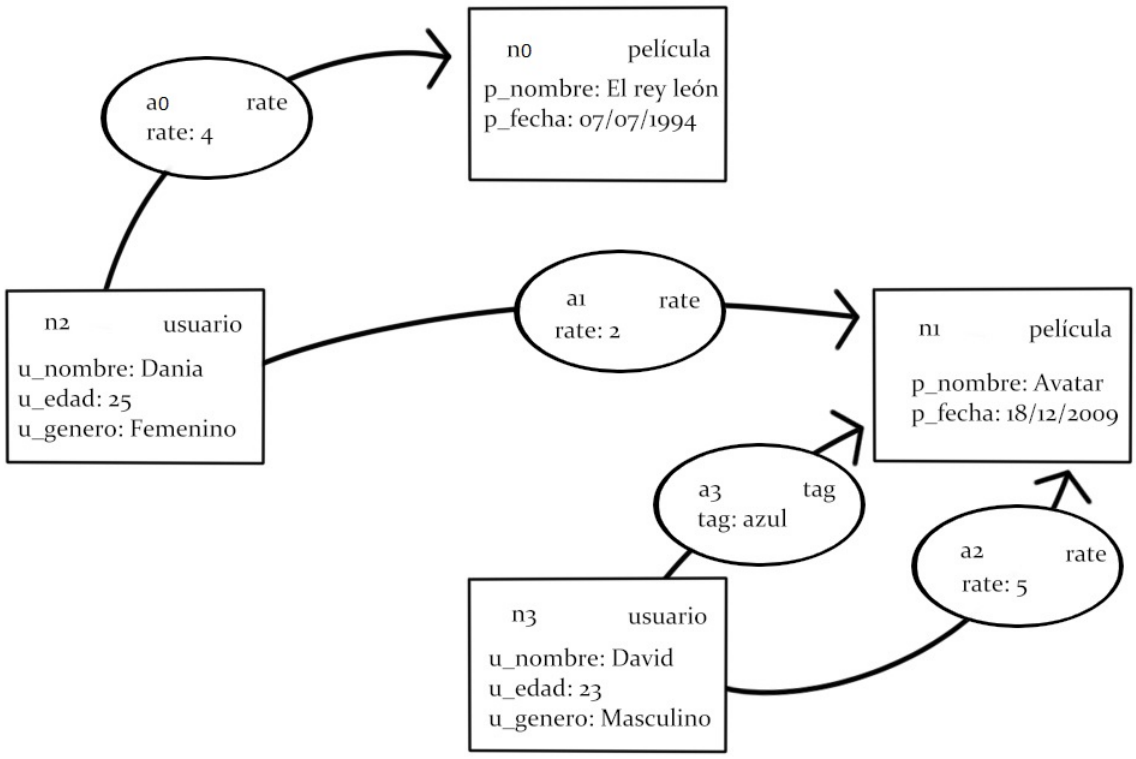

Figura 25: Ejemplo de grafo

[21] `https://github.com/borjaf696/k2AttDyn`

8.2. Estructura de datos

Para entender cómo Attk2DynTree guarda los grafos, y la importancia de los k^2-trees, es necesario explicar cada una de las partes que conforman este sistema. Los autores dividen la estructura en tres partes principales: esquemas, datos y relaciones. Estas, se describen brevemente a continuación (se recomienda mirar la Figura 27 ya que contiene un ejemplo de cómo se guardaría el grafo de la Figura 25).

8.2.1. Esquema

Es la parte de Attk2DynTree que se encarga de guardar el conjunto de atributos válidos para los diferentes tipos de nodos y aristas. Hay un esquema para nodos y otro para aristas. Estos esquemas se usan como la capa de entrada para acceder a las otras capas de información.

- **Esquema de nodos:** Guarda los distintos tipos de nodos y el tipo de cada nodo del grafo, en una lista (arreglo dinámico) ordenada lexicográficamente. Para guardar el tipo de cada nodo se usa una secuencia dinámica, representada por un wavelet tree [10]. Este permite eficientemente alocar todos los nodos de un tipo y también recuperar el tipo de un nodo, usando poco espacio[1]. Por cada tipo de nodo, se guarda un arreglo dinámico de punteros que apuntan a la información de los atributos que posee ese tipo. Junto a este arreglo, se guarda un arreglo de bits dinámico que indica si los atributos son densos o esparsos (ver sección 8.2.2).
- **Esquema de aristas:** Guarda la misma lista que el esquema de nodos, pero para aristas.

8.2.2. Atributos

Es la parte de Attk2DynTree que se encarga de guardar los distintos valores que toman los atributos de cada nodo y arista. Existen dos clasificaciones de atributos, los **esparsos** y los **densos**. Un atributo es esparso si es una variable que puede tomar muchos valores distintos (por ejemplo una URL), mientras que un atributo se considera denso si es una variable que toma un conjunto acotado de valores (por ejemplo, la edad o la nacionalidad). La forma de guardar cada atributo dependen de si es esparso o denso.

- **Atributos esparsos:** Cada uno de estos atributos tiene su propia lista dinámica doblemente indexada. Por un lado se guarda cada string en orden de llegada (indexada por el índice de su nodo o arista). Por otro lado se guardan los índices ordenados lexicográficamente, según el valor de su atributo. Por ejemplo, en la Figura 25, para el atributo p_nombre de las películas tenemos que primero se guarda "El rey león" y luego "Avatar" (el id indica quién llega primero). Por ello en la Figura 28, la lista p_nombre por un lado tiene guardado el arreglo ["El rey león", "Avatar"] (orden de llegada) y por otro lado tiene guardado el arreglo [3,2] (índices de los nodos ordenados lexicográficamente). La gracia de ordenar las listas es que luego se pueden hacer búsquedas binarias para buscar valores.

- **Atributos densos:** Todos los atributos densos se guardan en dos k^2-trees, uno para los nodos y otro para las aristas. Para construir el k^2-tree de los atributos densos de nodos, se hace lo siguiente (para las aristas se hace igual): Cada atributo denso puede verse como una relación binaria entre los nodos y el conjunto de todos los posibles valores de los atributos. Esta relación puede ser representada con una matriz de adyacencia, donde las filas son los id's de los nodos, y las columnas son los posibles valores que pueden tomar los atributos densos. Esto quiere decir que en la fila i, todos los 1's representan los atributos densos que posee el nodo i. Es muy importante mencionar que los atributos densos se guardan consecutivamente (por ejemplo, si nuestro esquema permite 5 nacionalidades, en la matriz de adyacencia las columnas que representan las nacionalidades se guardan juntas). En la Figura 27, las primeras 4 columnas de la matriz de los atributos densos de los nodos, representan los 4 valores que puede tomar la edad del usuario, mientras que las últimas dos columnas, representan el género del usuario. Esta matriz es guardada en un k^2-tree. Además del k^2-tree, es necesario guardar una pequeña estructura por cada atributo denso. Esta estructura contiene el nombre del atributo, el límite superior del rango de columnas que posee ese atributo, y una lista dinámica con los posibles valores del atributo.

8.2.3. Relaciones

La última parte de las estructuras de datos es la encargada de guardar la topología del grafo (las conexiones entre nodos). Todas las relaciones se guardan en un k^2-tree, que debe ser extendido para soportar multi aristas (guardando los id's de las aristas). Las estructuras de datos que componen esta parte son:

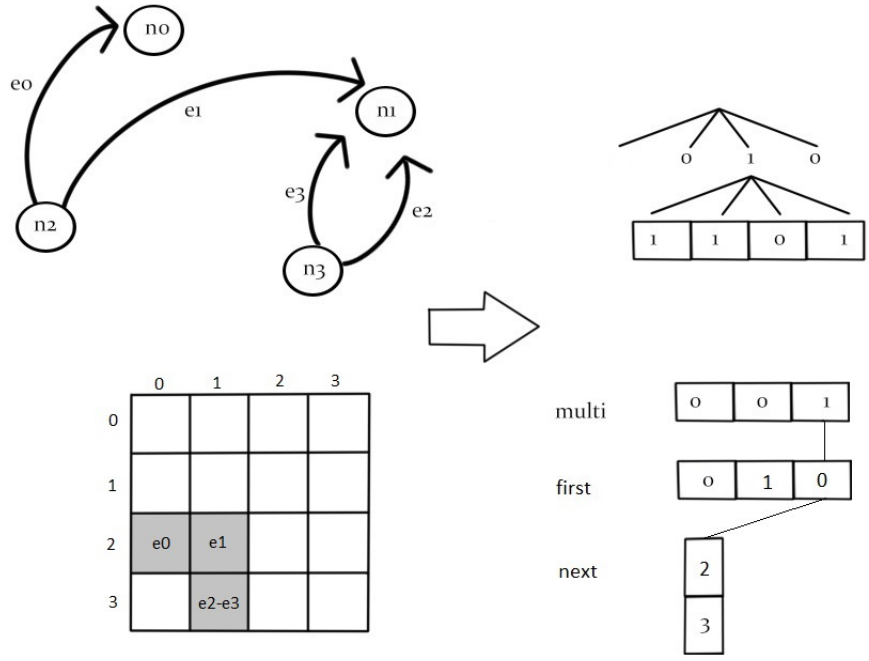

Figura 26: Topología del grafo de *rating* de películas

- **k^2-tree:** Es la estructura encargada de guardar la matriz de adyacencia que relaciona los nodos a través de las aristas. Esta matriz tiene un 1 en la posición $[i][j]$ si i y j están conectados por al menos una arista.
- **Multi:** Es un bitmap dinámico de tamaño igual al número de 1's en el último nivel del k^2-tree. De esa manera, $multi[i]$ representa si la arista cuyo 1 en el último nivel del k^2-tree (que está en la posición i de ese nivel), es multi arista o no (un 1 indica que es multi arista, y un 0 indica que no). Recordar que el último nivel del k^2-tree, tiene un 1 por cada arista. Por ejemplo en la Figura 26, $multi[2]$, representa la arista cuyo 1 está en la tercera posición en el último nivel del k^2-tree (tercera posición solo contando los 1's). y está arista corresponde a las aristas e_2 y e_3, por lo que es múltiple y por lo tanto $multi[2] = 1$.
- **First:** Es un arreglo dinámico que, por cada índice de multi, guarda el id de la arista correspondiente al 1 en el último nivel del k^2-tree que representa ese índice. Por ejemplo, sea a_i el par de nodos (u, v) representado por el 1 número i del último nivel del k^2-tree. Si $multi[i] = 0$ entonces $first[i]$ contiene el id de la arista que une u con v. Por otro lado, si $multi[i] = 1$, significa que aquella arista es múltiple, y $first[i]$ guarda un 0. Por ejemplo, en la Figura 26, para los nodos $(2, 1)$, la arista que los une es $e1$, en el k^2-tree corresponde al segundo 1 del último nivel. Como solo hay una arista uniendo esos nodos, se tiene que $multi[1] = 0$ y $first[1] = 1$ (el id de la arista). Por otro lado, los nodos $(3, 1)$, tienen las aristas $e2$ y $e3$, y corresponden al tercer 1 en el k^2-tree. Luego $multi[2] = 1$ y $first[2] = 0$.
- **Next:** Es un arreglo de arreglos (dinámicos). En cada arreglo se guardan todos los id's de las aristas que unen dos nodos. Cada 1 en multi representa un índice en next (el primer 1 de multi tiene el índice 0, el segundo 1 tiene el índice 1, y así sucesivamente). Por ejemplo para la Figura 26, el primer 1 en multi (que corresponde al tercer 1 en el último nivel del k^2-tree), representa a las aristas e_2 y e_3, por lo que la primera posición de next guarda un arreglo con los id's 2 y 3.

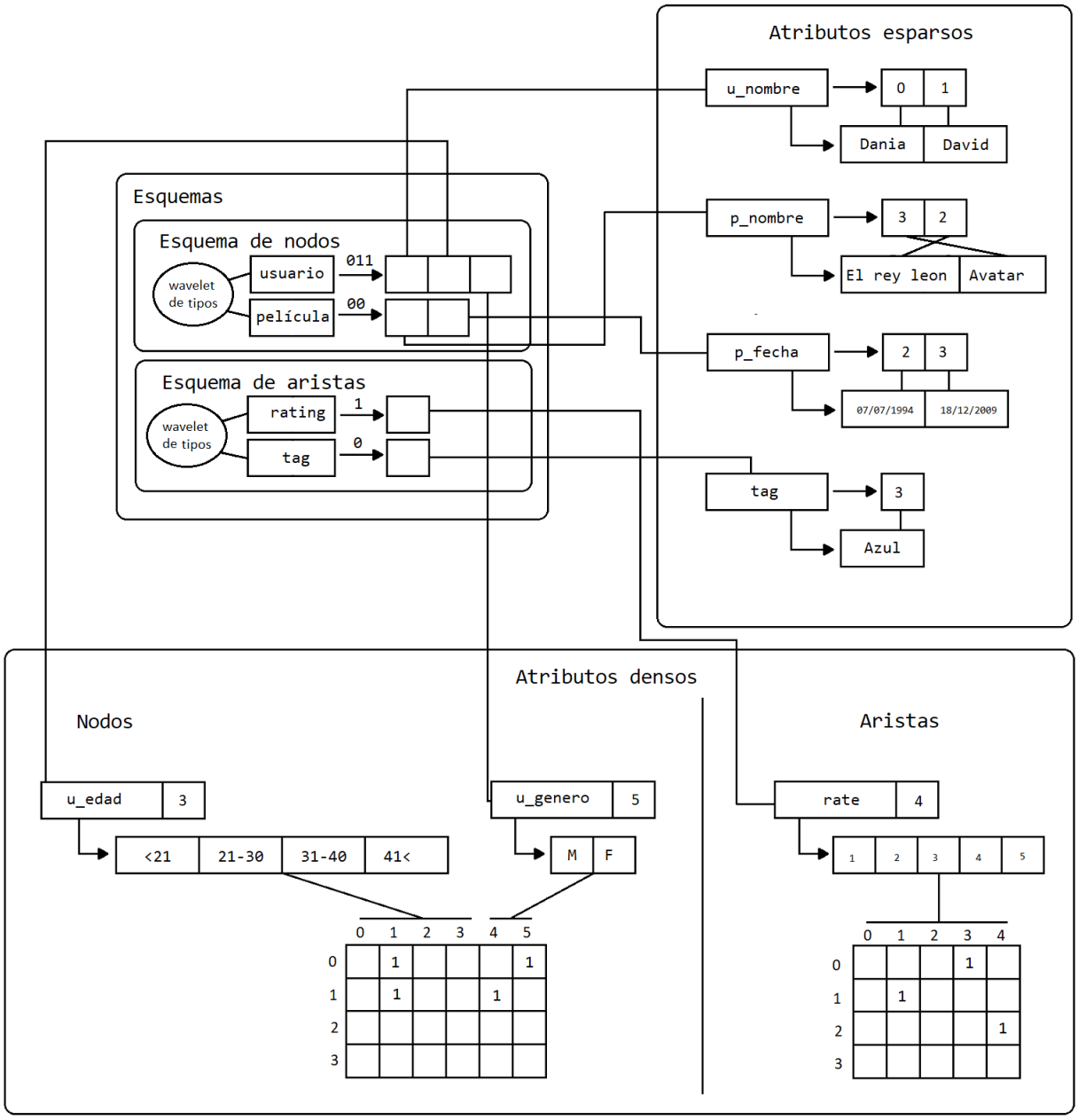

Figura 27: Ejemplo de capa de datos del grafo de *rating* de películas

8.3. Inserciones en Attk2DynTree

Como se menciona en la sección anterior, Attk2DynTree tiene distintas capas de datos, que representan las distintas partes de un grafo con atributos, por lo que a continuación se explica brevemente en qué consisten los distintos tipos de inserciones y cómo se implementan. Existen 3 tipos de inserciones, una para insertar en el esquema de nodos, otra para el esquema de aristas, y otra para la topología.

8.3.1. Insertar en un esquema

Las inserciones en el esquema de nodos o de aristas ocurren de la misma forma, así que solo se explicará la inserción de información en los nodos. Primero que todo, la inserción recibe un arreglo que contiene el tipo del nodo, el id y los valores que tienen cada uno de sus atributos. Usando el tipo del nodo, el esquema puede saber cuáles de sus atributos a insertar son densos y cuáles esparsos. Si el atributo es denso, primero se busca la columna que le corresponde al valor a insertar en el k^2-tree (por ejemplo en la Figura 27, en los atributos densos de los nodos, u_géneros dice que los valores M y F reciben las columnas 4 y 5 respectivamente). Una vez que se obtiene la columna, se inserta en el k^2-tree el nodo (id,columna). Si el atributo es esparso, se busca el arreglo que contiene los valores de ese atributo, y se inserta el id al final de la lista de id's, y el valor se inserta al final de la lista de valores (luego el arreglo de valores se ordena).

8.3.2. Insertar en la topología

Antes de explicar cómo ocurre una inserción de este tipo, es importante mencionar que el Attk2DynTree posee un parámetro que informa si el grafo permite múltiples aristas o no (esto no impide que pueda haber más de un tipo de arista). Si el grafo no permite múltiples aristas, solo se inserta la arista en el k^2-tree (el resto de estructuras no importan mucho, ya que se usan cuando el grafo permite multi aristas). Por otro lado, si el grafo tiene multi aristas, si la arista es la primera vez que aparece, se inserta un 0 en la posición correspondiente de multi, y se guarda el id de la arista en first. En el caso de que ya haya una arista en esa posición, si el valor de first no es 0, se cambia el valor correspondiente en multi por un 1, se inserta en next (en la posición determinada por multi) el valor que había antes en first junto con el id de la arista insertada, y se cambia el valor de first por un 0. Si el valor de first es 0, solo se inserta en next el id de la arista en la posición determinada por multi.

8.4. Consultas sobre Attk2DynTree

En esta sección se presentan las consultas que se pueden hacer sobre los grafos representados por Attk2DynTree. Estas se dividen en tres grupos, uno para cada una de las capas presentadas en la sección anterior.

8.4.1. Consultas sobre los esquemas

- **getNodeType/getEdgeType:** Retornan los diferentes tipos de nodos o aristas que hay. Lo que se hace es simplemente retornar la lista que de tipos que posee cada esquema. Por ejemplo en la Figura 27, getEdgeType retornaría la lista [rating, tag].
- **scanNodes(type)/scanEdges(type):** Esta consulta filtra los nodos o aristas, según un tipo, y retorna el rango de índices del tipo especificado. Se implementa buscando el tipo del nodo/arista en el wavelet tree, lo cual retorna todos los nodos/aristas que pertenecen a ese tipo (retorna un conjunto de rangos, para mantener el resultado compacto). Por ejemplo, para la Figura 27, scanNodes(película) retorna el rango $[2, 3]$, ya que desde el id 2 hasta el 3 son nodos películas.
- **getNodeType(id)/getEdgeType(id):** Retorna el tipo del nodo/arista con identificador id. Por ejemplo para el grafo de la Figura 25, getEdgeType(3) retorna tag, ya que la arista $e3$ es del tipo tag. La consulta se implementa buscando el tipo del nodo en el wavelet tree.

8.4.2. Consultas sobre los atributos

- **getNodeAttribute(id,att)/getEdgeAttribute(id,att):** Esta consulta retorna, para el nodo (o arista) con identificador id, el valor de su atributo att. Por ejemplo, getNodeAttribute(2,"u_nombre"), retorna "Dania" (ya que el nodo $n2$ tiene el valor "Dania" para el atributo "u_nombre"). La implementación de esta consulta es igual para nodos y aristas, y empieza igual para atributos densos y esparsos. Por ejemplo para los nodos, se parte buscando el tipo del nodo con getNodeType(id). Luego se recorre la lista de atributos válidos buscando att. Si el atributo no existe, se retorna vacío. En caso contrario se chequea si el atributo es esparso o denso.

  Si es esparso, se accede a la capa de atributos esparsos, buscando la lista att. Luego, se usa el wavelet tree para recuperar lo posición del id del nodo en la lista y se retorna ese valor. Por otro lado, si el atributo es denso, se accede a la capa de atributos densos, buscando la lista att. Luego, se hace una búsqueda de rango entre la fila id, y las columnas que pertenecen a att. Por ejemplo getNodeAtribute(2,"u_edad") hará una búsqueda de rango para el rectángulo $(r1 = 2, r2 = 2, c1 = 0, c2 = 4)$
- **selectNodes(type,att,val)/selectEdges(type,att,val)** Retorna todos los nodos/aristas del tipo especificado que tengan su atributo att con el valor val. Por ejemplo, selectNodes("usuario","p_género","M"), retorna la lista [3] ya que el nodo $n3$, es el único usuario de género masculino (si hubieran más, la lista seria más grande).

La implementación de esta consulta es la misma para nodos y aristas, y funciona de la siguiente manera: Se busca att en la listas de atributos del tipo especificado. Si el atributo es esparso, se accede a la capa de atributos esparsos y se hace una búsqueda binaria sobre la lista del atributo (la cual está ordenada también lexicográficamente) buscando el valor val, y se retornan todos los id's encontrados. Si el atributo es denso, se accede a la capa de atributos densos, se busca el atributo att junto a las columnas que le pertenecen $[c1, c2]$. Luego se hace una consulta de rango sobre todos las filas y para las columnas $c1$ y $c2$.

8.4.3. Consultas sobre las relaciones

Estas consultas se hacen sobre la topología del grafo.

- **neighbors(Type,id):** Retorna todos los nodos del tipo especificado que son vecinos con el nodo id. La implementación de esta consulta empieza buscando los rangos de nodos del tipo especificado $[c1_i, c2_i]$ (usando scanNodes(type)), luego hace una consulta de rango sobre el k^2-tree por cada rango, usando el rectángulo $(id, id, c1_i, c2_i)$.
- **related(type,id):** Retorna todos los nodos, relacionados con el nodo id, que se conectan a través de una arista del tipo especificado. La implementación de esta consulta empieza buscando los rangos de aristas del tipo especificado. Luego se buscan todos los vecinos de id. Por cada vecino se obtiene una arista con un id, si el id está dentro de alguno de los rangos, se guarda en el resultado. Esta consulta utiliza los arreglos multi, first y next, para buscar los id's de las aristas.

8.5. Evaluación de Dyn-array1

Antes de insertar el nuevo k^2-tree en la base de datos, se pone a prueba Dyn-array1 en algunos de los experimentos de la sección 7.

8.5.1. Insertar 194 millones de aristas

Se realiza el mismo experimento de la sección 7.2, incluyendo a Dyn-array1. En la Figura 28 se muestra el tiempo de inserción por arista. Se puede observar que Dyn-array1 se demora hasta 10 veces más que Blocks. Por otro lado, en la Figura 29 se muestra que la memoria total ocupada por Dyn-array1 es bastante parecida a la de las demás estructuras.

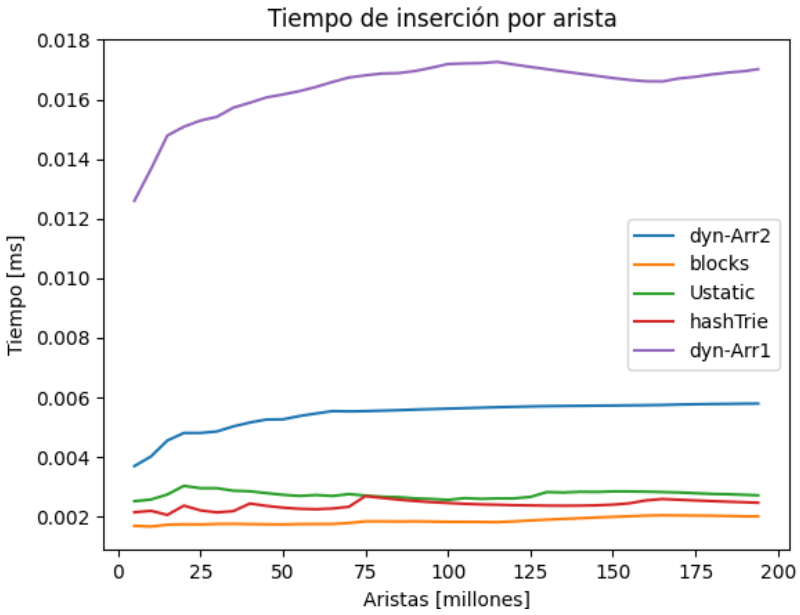

Figura 28: Tiempo promedio de inserción, por arista. Dyn-array1 incluido.

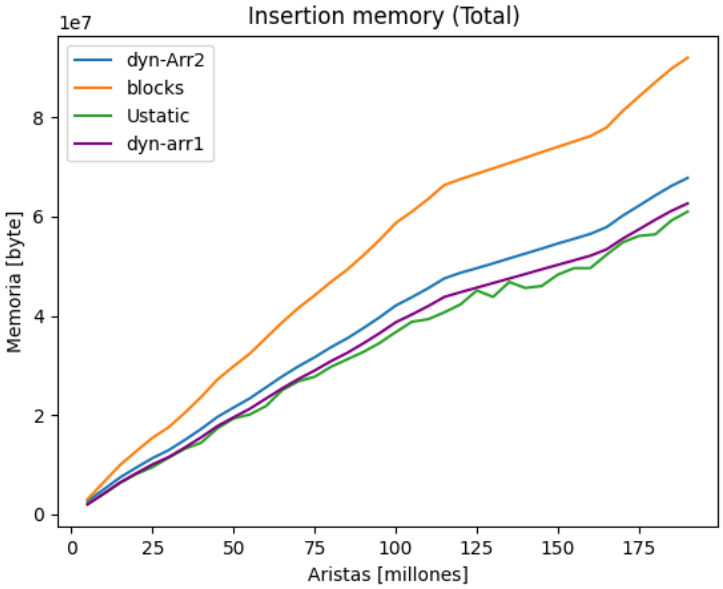

Figura 29: Evolución de la memoria total al insertar aristas, Dyn-array1 incluido.

8.5.2. Buscar 194 millones de aristas

Se realiza el mismo experimento de la sección 7.3, incluyendo ahora a Dyn-array1. En la Figura 30 se muestra el tiempo promedio de búsqueda de aristas. El tiempo de búsqueda de Dyn-array1 es hasta 5 veces mayor que Blocks.

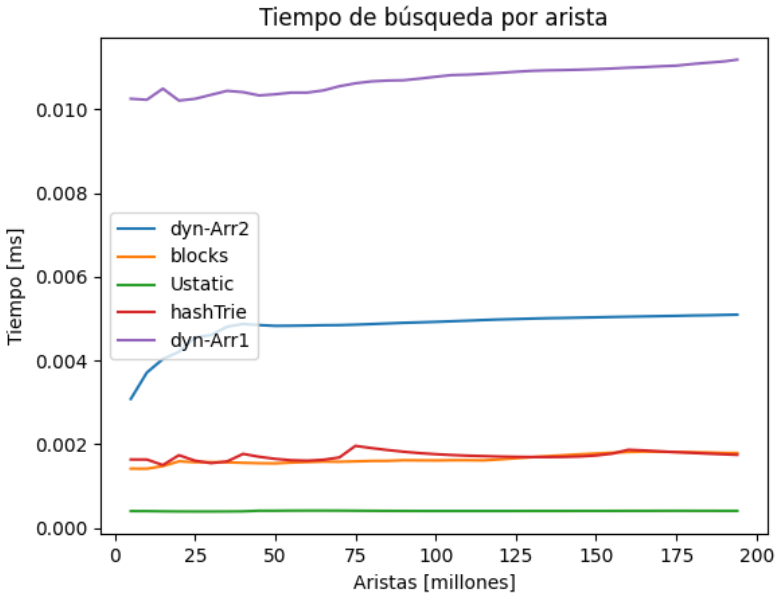

Figura 30: Tiempo de búsqueda, por arista, con Dyn-array1 incluido.

8.5.3. Consulta de rango

Se hace el mismo experimento de la sección 7.8, pero esta vez se incluye a Dyn-array1. En la Figura 31 se observa que Dyn-array1 es la estructura que más se demora en hacer consultas de rango. Este experimento y el anterior son muy importantes, ya que la búsqueda de vecinos y las consultas de rango son operaciones muy usadas por la base de datos, y sus resultados sirven para explicar los resultados de los experimentos del capítulo 9.

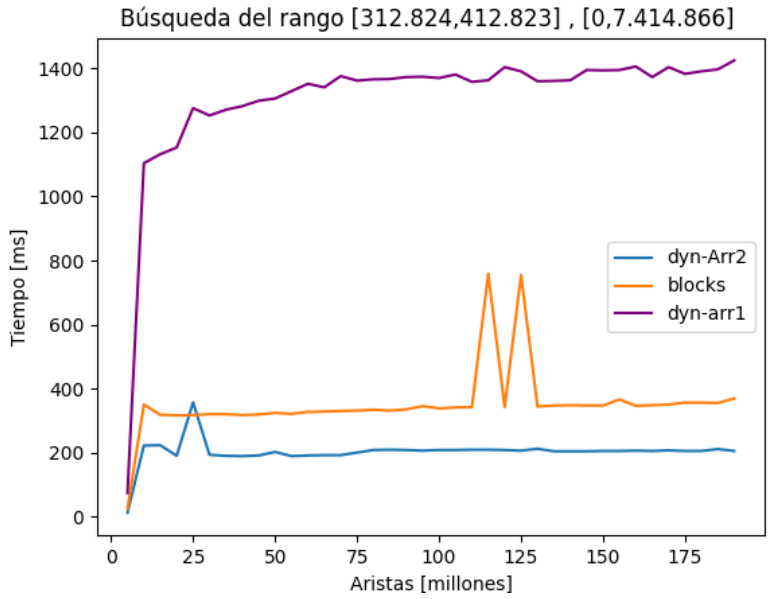

Figura 31: Tiempo de búsqueda de rango, con Dyn-array1 incluido

8.6. Integración de Blocks en Attk2DynTree

Para integrar Blocks en Attk2DynTree se crea una segunda versión de Attk2DynTree, a la que se le da el nombre de Attk2DynTree2[22]. Esta nueva estructura, se considera una versión experimental, ya que posee ambos tipos de k^2-trees (Blocks y Dyn-array1), junto con un duplicado de cada operación de Attk2DynTree que utiliza k^2-trees, pero adaptado a Blocks. Se decide implementarlo de esta forma, ya que como el código intervenido es de terceros, mantener la versión antigua ayuda a entender cómo implementar la nueva.

8.6.1. Estructuras de datos intervenidas

En la sección 8.2, se menciona que Attk2DynTree posee un k^2-tree (Dyn-array1) para el esquema de nodos, otro para el esquema de aristas y uno para la topología del grafo. Por ello Attk2DynTree2 debe agregar un segundo k^2-tree (Blocks) para cada esquema y para la topología (se decide hacerlo así, ya que de esta forma se puede seguir el rastro de los Dyn-array1 mientras se integran los Blocks).

8.7. Inserciones en Attk2DynTree2

Para insertar información en la capa de atributos, solo se cambia la parte del código que inserta atributos densos. Lo que se hace es cambiar la función que inserta la arista (a, b) en Dyn-array1, por la función que inserta aristas en Blocks. En cuanto a la inserción de aristas en la topología del grafo, se pierde la capacidad de tener aristas múltiples entre dos nodos (de grafo). Esto se debe a que para actualizar los valores de los arreglos multi, first y next es necesario hacer operaciones de rank y select en el último nivel del k^2-tree. Este último nivel puede ser accedido fácilmente por un k^2-tree que sigue una estrategia LOUDS (ya que el árbol se recorre en BFS). Como Blocks usa una estrategia DFUDS, es mucho más difícil poder acceder a este último nivel. Por esto se decide dejar de lado la capacidad de tener aristas múltiples (no se usan los arreglos multi, first y next), al menos en este trabajo.

8.7.1. Consultas intervenidas

Las consultas que se intervienen son solo aquellas que usan k^2-trees.

- **Capa de esquema:** No cambia.
- **Capa de atributos:** Cambia getNodeAttribute(id,att), getEdgeAttribute(id,att), selectNodes(type,att,val) y selectEdges(type,att,val). En todas las operaciones anteriores, se hacen búsquedas de rango sobre el k^2-tree, por lo que en las nuevas versiones de estas operaciones la búsqueda se hace sobre el nuevo k^2-tree (Blocks).
- **Capa de relaciones:** Cambia neighbors(type,id). En esta operación se hace una búsqueda de rango sobre el k^2-tree que representa la topología del grafo. Nuevamente se realiza la búsqueda sobre el nuevo k^2-tree. Se ignora la función related(type,id), ya que funciona usando la información de multi, first y next.

[22] `https://github.com/daviddelapuente/dynAttK2treeDatabase`

9. Evaluación de la integración

En estos experimentos se trabaja con tres grafos de distinto tamaño, con los que se busca poner a prueba las dos versiones de Attk2DynTree. Los grafos usados son los que se mencionan en la sección 2.8.2. Estos grafos, contienen *ratings* de películas hechos por usuarios de la página `https://movielens.org/`. A continuación se explican más detalladamente.

9.1. Dataset ml100k

Este es un conjunto de archivos que contiene 100.000 *ratings* hechos por 943 usuarios, sobre 1682 películas. Entre los archivos más importantes se tiene:

- **u.user:** Es una lista con la información de todos los usuarios. De cada usuario se guarda un id, su edad (entre 0 y 60 años), su género (masculino o femenino), su ocupación (cada usuario tiene solo una ocupación de entre 21 posibilidades) y código postal. El archivo posee una fila por usuario.
- **u.item:** Es una lista con la información de todas las películas. De cada película se guarda un id, el título, año de estreno (entre 1930 y 1998), una url y una lista de los géneros a los que pertenece la película. Una película puede tener más de un género (por ejemplo, puede ser de comedia y romántica). El archivo posee una fila por película.
- **u.genre** Una lista con todos los 19 géneros que puede tener una película. Los géneros son: *unknown*, *Action*, *Adventure*, *Animation*, *Children's*, *Comedy*, *Crime*, *Documentary*, *Drama*, *Fantasy*, *Film-Noir*, *Horror*, *Musical*, *Mystery*, *Romance*, *Sci-Fi*, *Thriller*, *War*, *Western*.
- **u.info:** Es una lista con todos los *ratings*. Cada fila del archivo contiene el id del usuario, el id de la película, el *rate* que el usuario le dio a la película (un número entre 1 y 5), y la fecha (*time-stamp*).

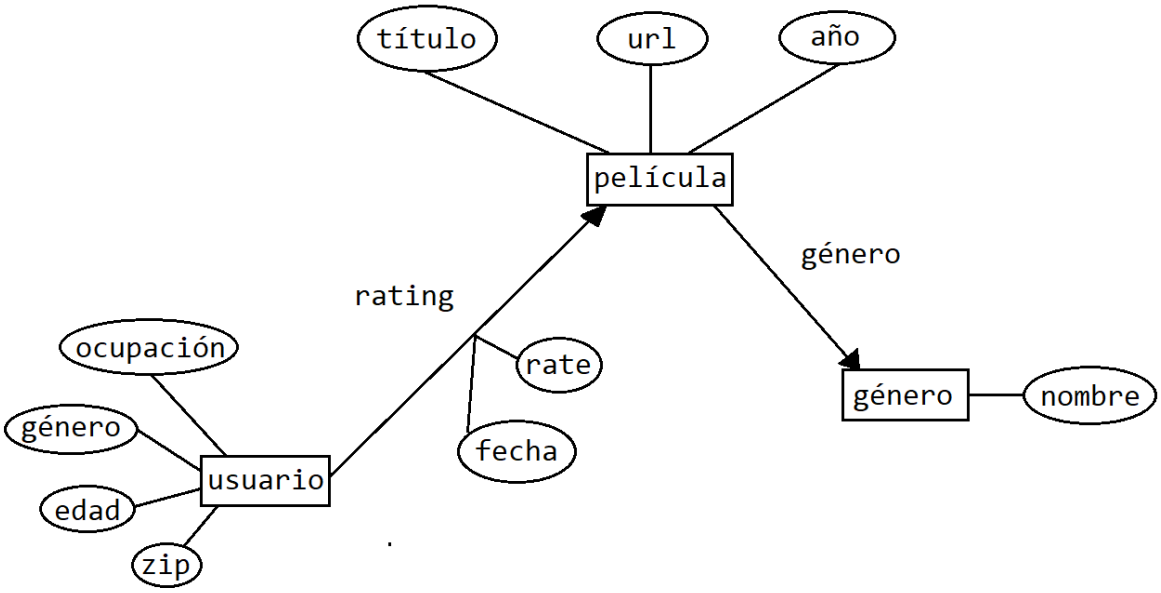

Figura 32: Grafo de ml100k

La información anterior se puede ver como un grafo con atributos (figura 32). Este grafo tiene como nodos los usuarios, las películas y los posibles géneros de las películas. Como aristas tiene los *ratings* hechos por usuarios, y el género que puede tener una película. Los *ratings* son aristas dirigidas que van desde un nodo usuario hacia un nodo película (un usuario puede calificar más de una película y una película puede ser calificada por más de un usuario). Las aristas de tipo género también son aristas dirigidas, que van desde una película hacia uno de los 19 posibles géneros (de esta forma, una película puede apuntar a varios géneros). La clasificación entre denso y esparso de los atributos de cada nodo y arista es la siguiente:

nodos:
usuario(edad=denso, género=denso, ocupación=denso, zip=esparso).
película(título=esparso, url=esparso, año=denso).
género(nombre=denso).

aristas:
rating(*rate*=denso, fecha=esparso).
género(sin atributos).

Notar que las aristas género no poseen atributos (no es necesario que los nodos y aristas tengan atributos). También es importante mencionar que no se colocan los id's de nodos ni aristas, ya que esa información aparece a medida que se insertan los nodos y las aristas, y se guarda en las capas de datos. Por ejemplo, para insertar los nodos usuario, lo que se hace es iterar el archivo u.user, de tal forma que la posición de la fila puede ser el id del usuario. Este id queda guardado en el k^2-tree de topología y en el de atributos densos de usuarios. Para las aristas ocurre lo mismo.

9.2. Datasets ml10M y ml25M

El dataset ml10M es un conjunto de archivos muy parecido al anterior, contiene 10.000.054 *ratings* hechos por 71.567 usuarios, sobre 10.681 películas. Los archivos también contienen 95.580 tags, hechos por usuarios sobre películas, pero esta información no se considera en los experimentos, ya que implicaría que el grafo soporte aristas múltiples. Entre los archivos usados, se tiene:

- **movies.dat:** Es una lista con la información de las películas. Cada película contiene un id, un título, año de estreno y un conjunto de los géneros a los que pertenece la película (son los mismos que posee ml100k).
- **ratings.dat:** Es una lista con todos los *ratings*, cada fila del archivo contiene el id de la película, el id del usuario, el *rate* que el usuario le dio a la película, entre 1 y 5, y la fecha (*time-stamp*).

Para ml10M, se decide mantener los usuarios anónimos, por lo que solo poseen un id (un número entre 0 y 71.566). La información anterior tambien se puede representar con un grafo con atributos (figura 33), de hecho el grafo es muy similar al de la Figura 32, solo que los nodos tienen menos atributos. En cuanto a la clasificación entre atributos densos o esparsos, los atributos que quedan mantienen esa característica (por ejemplo, el año de estreno de una película sigue siendo un atributo denso).

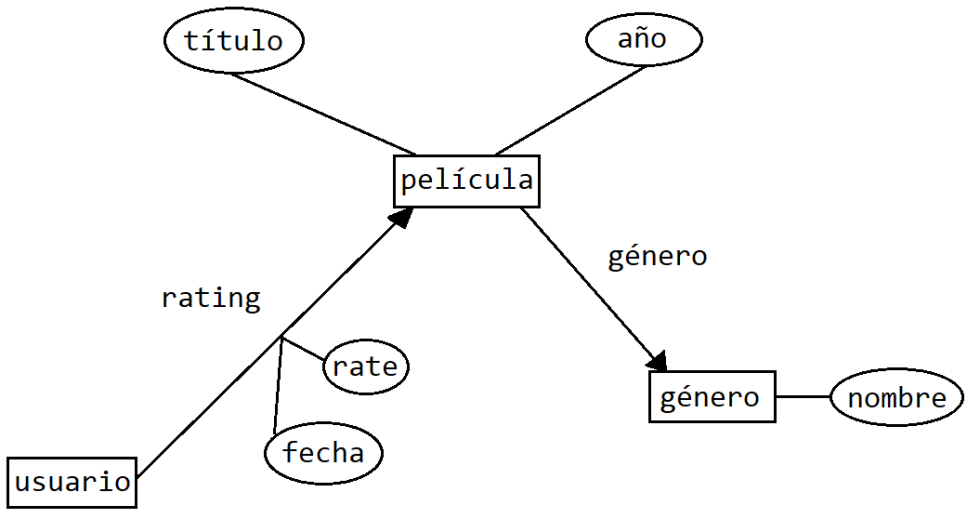

Figura 33: Grafos de ml10M y ml25M

El dataset ml25M es un conjunto de archivos igual al anterior (tiene la misma estructura), contiene 25.000.095 *ratings*, hechos por 162.541 usuarios, sobre 62.423 películas. Los archivos tienen los mismos nombres que en ml10M, y como contienen la misma información, el grafo de ml25M se ve idéntico al de la Figura 33. La clasificación entre atributos densos o esparsos también se mantiene igual.

9.3. Comparación experimental

En los experimentos que se muestran en las siguientes secciones, se ponen a a prueba Attk2DynTree y Attk2DynTree2. La idea es comparar estas dos estructuras en términos de inserción de nodos y aristas (de grafo), en términos de memoria ocupada y tiempo de consultas. Cada set de experimentos se replica en ml100k, ml10M y ml25M. De esta forma se puede apreciar cómo afecta el tamaño del grafo, al comportamiento de las estructuras. La configuración de la máquina en la que corren los experimentos, y de la estructura Blocks, es la misma que aparece en la sección 7.1.

9.3.1. Tiempos de inserciones en Attk2DynTree

Este experimento consiste en insertar todos los nodos y aristas de cada uno de los grafos, midiendo el tiempo total de inserción. En la Figura 34 se muestra el tiempo total de insertar las relaciones del grafo (las aristas de la topología). Se observa que Attk2DynTree2 se demora mucho menos que la versión antigua. En la Figura 35 se muestra el tiempo total de insertar los atributos de los nodos del grafo. Se observa que a medida que aumenta el número de nodos, aumenta la diferencia entre los tiempos de inserción. Por otro lado, se puede observar que la diferencia no es tan grande como en la Figura 34. Esto se debe a que la inserción considera atributos esparsos y densos. La diferencia de tiempos en la Figura 34 se debe en mayor parte a que Blocks inserta más rápido que Dyn-array1 los atributos densos.

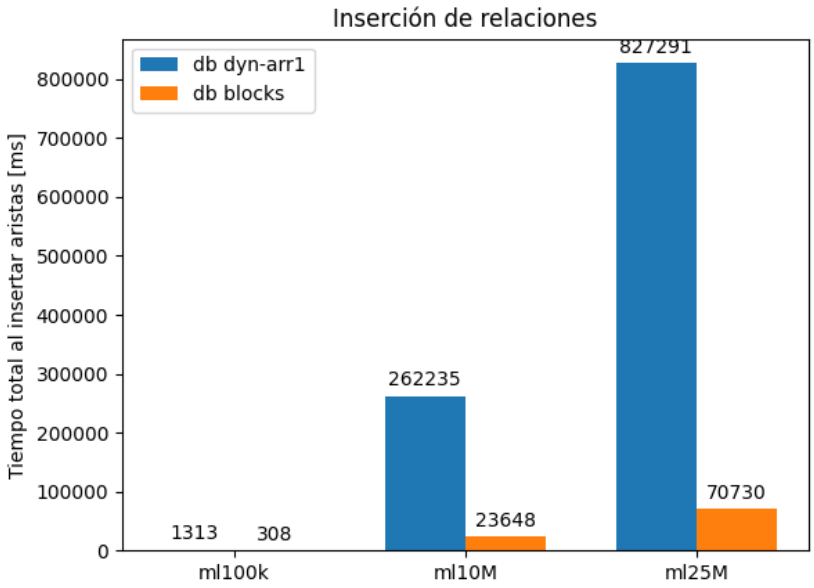

Figura 34: Tiempo total de inserción de aristas en la topología

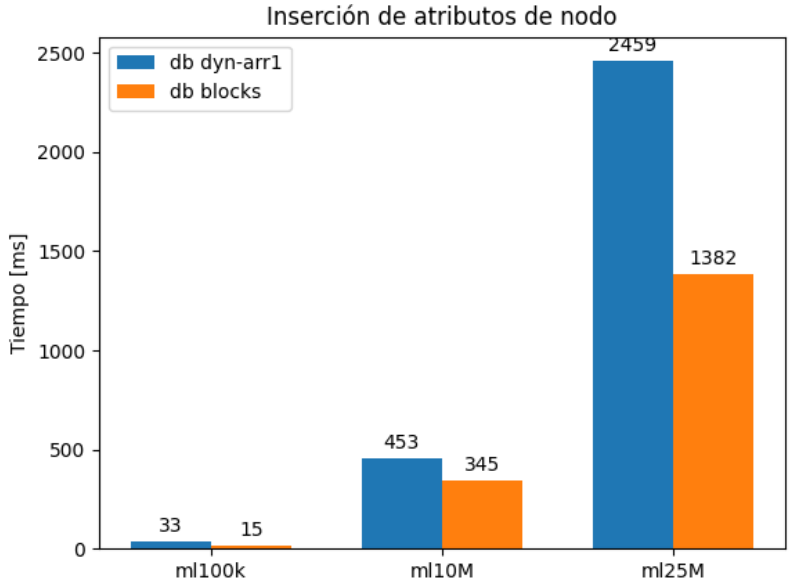

Figura 35: Tiempo total de inserción de los valores de los atributos de los nodos del grafo

En cuanto a la inserción de los atributos de aristas, la Figura 36 muestra un comportamiento bastante parecido a la inserción de atributos de nodos, pero la diferencia de tiempos es mayor, ya que el número de atributos insertados es mayor. Por último, en la Figura 37 se muestran los distintos tamaños de la base de datos, para los distintos grafos. Se observa que Attk2DynTree ocupa menos espacio Attk2DynTree2.

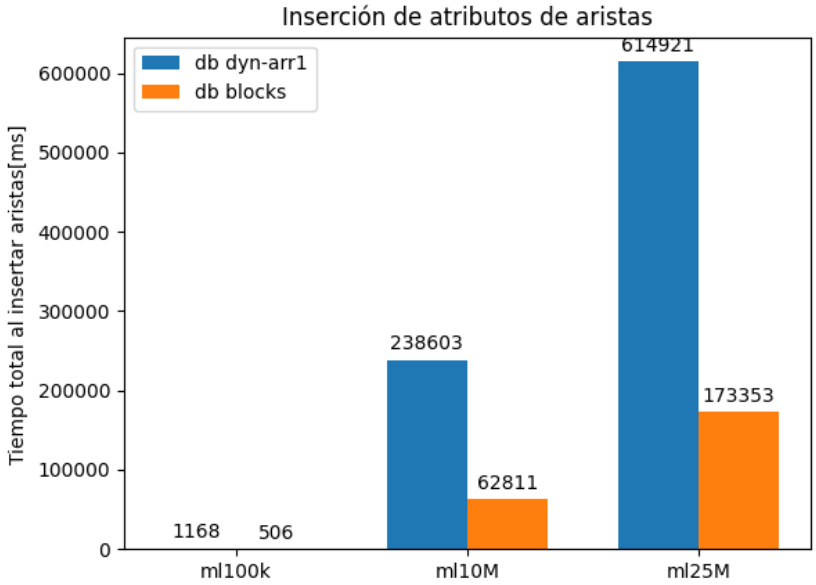

Figura 36: Tiempo total de inserción de los valores de los atributos de las aristas del grafo

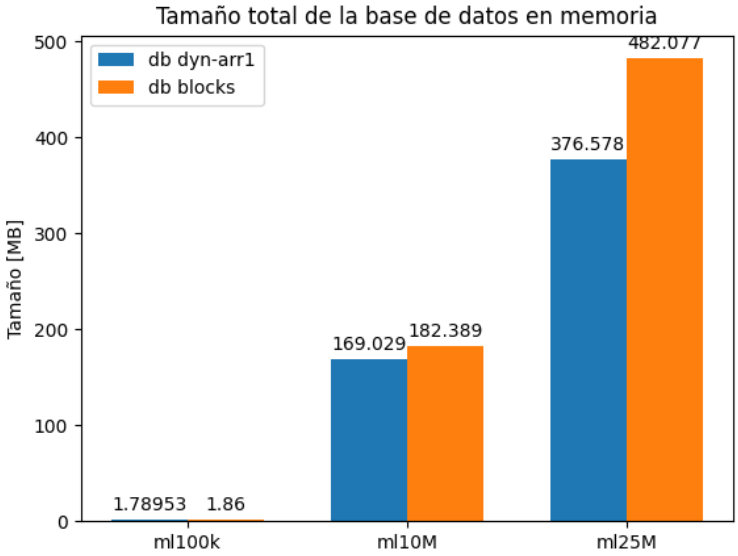

Figura 37: Tamaño total de la base de datos

9.3.2. getNodeAttribute

Una vez que se construyen los distintos grafos, es interesante saber cuánto se demora la estructura en hacer consultas getNodeAttribute(id,att). Se decide que la medición se hará sobre el atributo año de estreno de los nodos película. Se elige este atributo, ya que es denso y aparece en ml100, ml10M y ml20M. Para calcular el tiempo, se mide cuánto se demora un programa que hace la consulta en cuestión, por cada uno de los nodos película. En la Figura 38 se observa que Attk2DynTree2 se demora menos de la mitad del tiempo que demora Attk2DynTree. Esto se corresponde con la Figura 31, que compara las consultas de rango efectuadas por Dyn-array1 y Blocks.

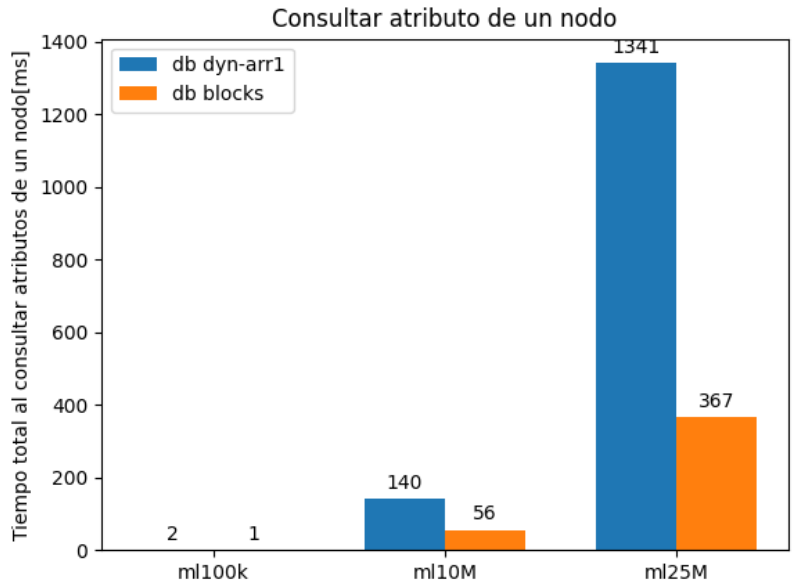

Figura 38: Tiempo total de búsqueda del atributo año de estreno, por id de nodo

9.3.3. getEdgeAttribute

Al igual que en el experimento anterior, luego de que los grafos están cargados en memoria, se quiere saber cuánto demoran las estructuras en responder la consulta getEdgeAttribute(id,att) para el atributo *rate*, de todas las aristas de *rating*. Se decide usar este atributo ya que es denso y aparece en todos los grafos. Para medir el tiempo en cada estructura, se calcula el tiempo total de preguntar por el *rate* de cada arista (una por una). En la Figura 39 se observa que Attk2DynTree2 demora casi 4 veces menos que Attk2DynTree.

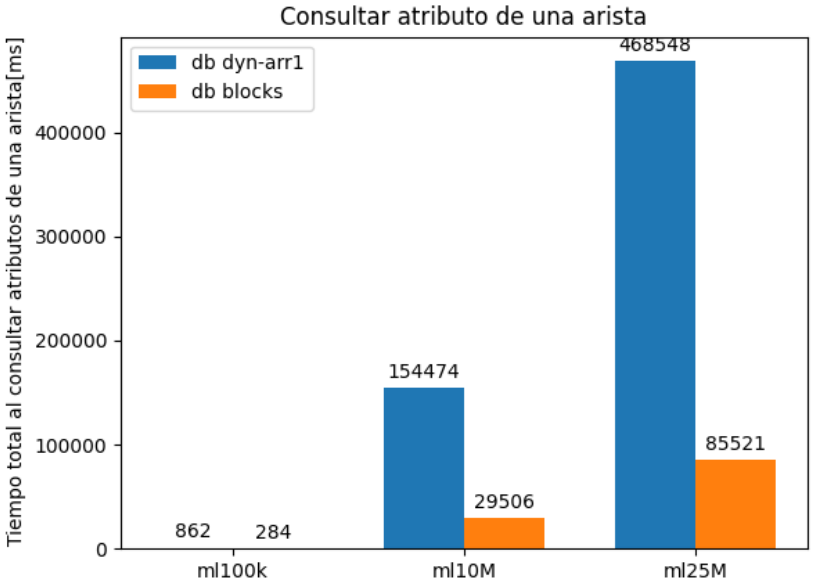

Figura 39: Tiempo total de búsqueda del atributo *rate* , por id de arista

9.3.4. selectNodes

Otra consulta que interesa medir es la que busca todos los nodos de cierto tipo que tienen un valor específico para uno de sus atributos. La medición se hace sobre el atributo año de estreno de los nodos película. Se pregunta por el tiempo total, de recuperar todas las películas que fueron estrenadas entre los años 1915 y 2008 (se hace una consulta por año). En la Figura 40 se muestran los tiempos que demora Attk2DynTree y Attk2DynTree2 en resolver estas consultas. Nuevamente Attk2DynTree2 resuelve esta consulta en menor tiempo.

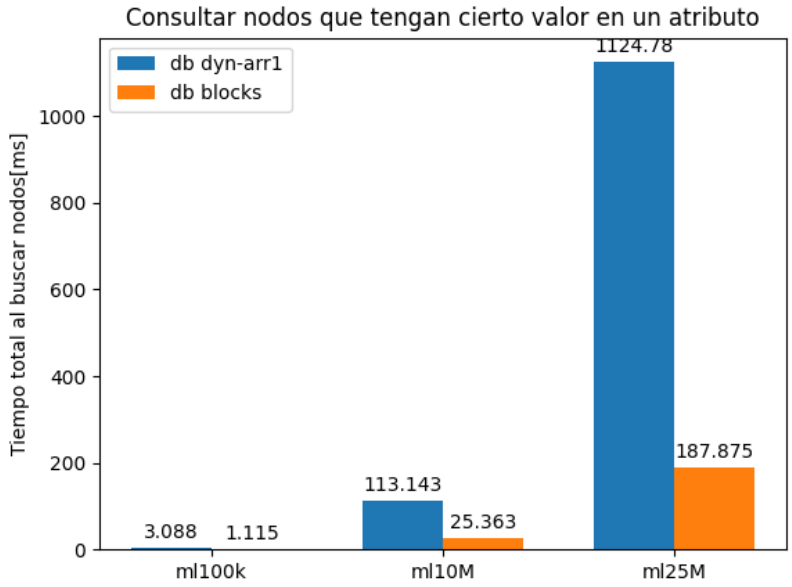

Figura 40: Tiempo total de búsqueda de todos las películas entre los años 1915 y 2008

9.3.5. selectEdges

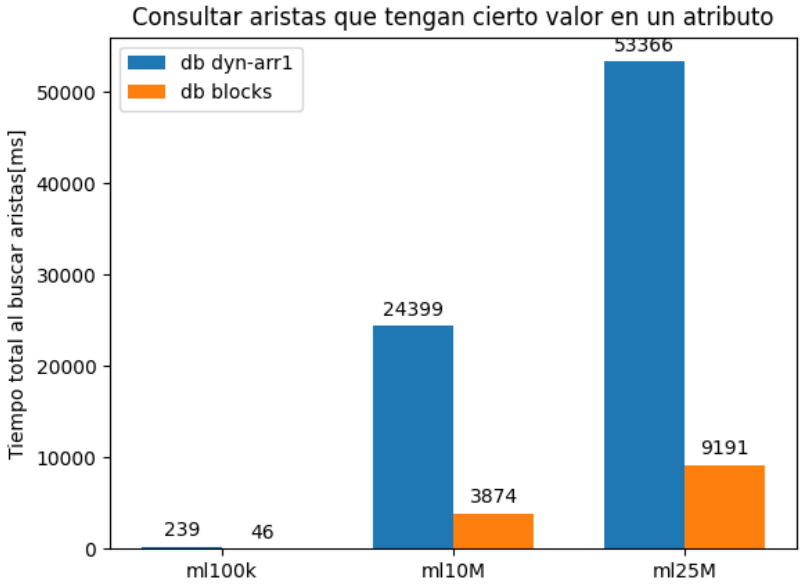

Figura 41: Tiempo total de búsqueda de todos los *rates* iguales a 1.0, 2.0, 3.0, 4.0 o 5.0

Al igual que en el experimento anterior, interesa medir el tiempo de búsqueda de todas las aristas de cierto tipo que poseen un valor específico para uno de sus atributos. La consulta se hace sobre el atributo *rate* de las aristas *rating*. En particular se buscan todas las aristas que poseen *ratings* con valores 1.0, 2.0, 3.0, 4.0 y 5.0. Se mide el tiempo total de realizar todas las búsquedas (se hace una búsqueda por *rate*). En la Figura 41 se muestran los tiempos totales de realizar estas consultas. Nuevamente, Attk2DynTree2 tiene mejores tiempos.

9.3.6. neighbors

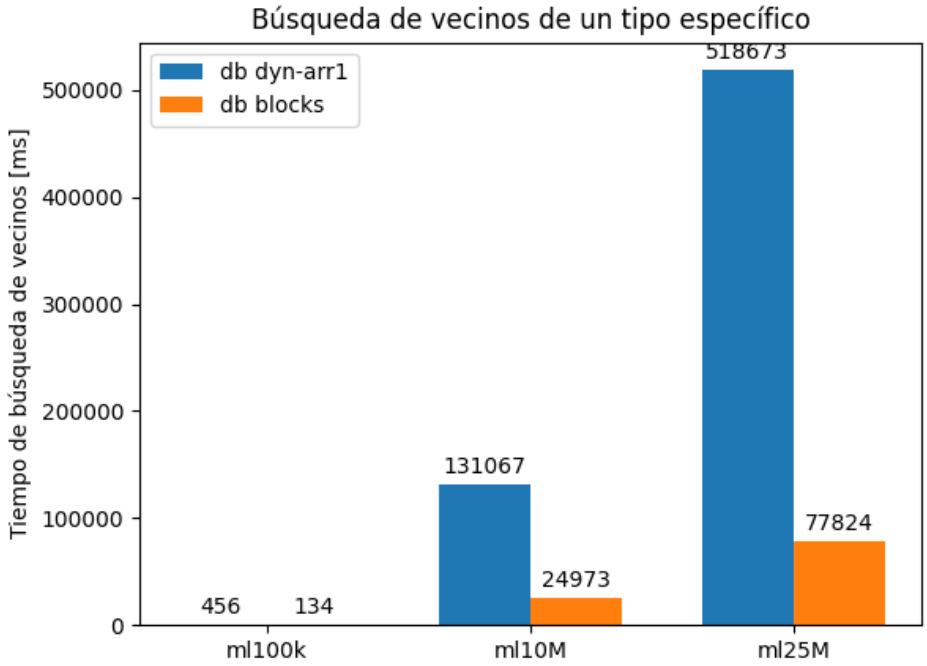

Figura 42: Tiempo total de búsqueda de vecinos (de cierto tipo) de un nodo

La última consulta que se mide es la que busca, para un nodo en particular, todos sus vecinos de cierto tipo. Se decide buscar todos los *rates* que un usuario ha hecho sobre las películas. Se hace esta *query* por todos los usuarios y se mide el tiempo total. En la Figura 42, se observa que Attk2DynTree2 resuelve esta consulta mucho más rápido que Attk2DynTree. Esta es una de las consultas que tiene mayor diferencia de tiempo, y se debe a que la consulta de rango es mucho más "limpia", es decir se hace directamente sobre el k^2-tree, y no pasa por otras capas de datos antes.

9.4. Análisis general de los resultados del Hito 3

De los experimentos anteriores, se puede deducir que la nueva versión de Attk2DynTree es más rápida para insertar información (hasta 20 veces más rápido) y hacer consultas (hasta 3 veces más rápido). Ocupa un poco más de espacio (28 %), pero el espacio aumentado no es tanto, comparado con la mejora de tiempo en las consultas, por lo que vale completamente la pena hacer el cambio a Attk2DynTree2. Sin embargo, no hay que olvidar que la nueva versión, Attk2DynTree2, no soporta múltiples aristas entre dos nodos. Esto es una limitante, ya que en la mayoría de los casos las bases de datos de grafos necesitan esta propiedad. Vemos al menos dos formas de soportar multi aristas, por ejemplo se podría guardar de alguna forma en Blocks, el último nivel del k^2-tree, de una forma que permita hacer los ranks que necesitan los arreglos multi, first y next. O tambien se podría trabajar con un tensor de 3 dimensiones, para que las multi aristas se guarden en la tercera dimensión (esto es usar un k^3-tree).

Por otro lado, en ninguno de los experimentos se pone a prueba la capacidad de borrar información, y es que la primera versión de Attk2DynTree no estaba pensada para borrar. Un punto importante a mencionar es que Ustatic también era buen candidato, pues realiza varias operaciones de manera muy rápida ocupando una buena cantidad de memoria, sin embargo al ser una estructura amortizada, pueden existir casos muy malos que serían inaceptables en un motor de base de datos que se use en producción.

10. Conclusión y trabajo futuro

Se presenta una nueva versión de Attk2DynTree que permite insertar nodos y aristas de grafos con atributos, mucho más rápido y usando solo un poco más de memoria. Además de insertar, esta nueva versión permite realizar consultas sobre los grafos de manera mucho más eficiente (búsqueda de vecinos, recuperar atributos, recuperar todos los nodos/aristas con cierto valor, entre otras). El precio que se paga es dejar de lado la capacidad de soportar multi aristas, pues por el momento Blocks no tiene la capacidad de realizar consultas de rank en su último nivel. Además de la optimización de tiempo, nuestra versión de Attk2DynTree, podría extenderse para soportar borrado de información, al menos para los atributos densos, y las relaciones entre nodos. Esto último es posible, debido a que el nuevo k^2-tree que usa permite borrados.

Como paso intermedio para cumplir con el objetivo principal, se agregan nuevas operaciones al al k^2-tree Blocks, de tal manera que ahora soporta operaciones de borrado de aristas, búsqueda de vecinos y consultas de rango, con eficiencia competitiva con el resto de estructuras. De los experimentos se puede concluir que existen varias formas de representar un k^2-tree dinámico eficientemente. Entre ellos, se decide que Blocks es la mejor opción para integrar en Attk2DynTree, ya que es la estructura más rápida en varias operaciones y la que mejor gestiona su memoria.

En cuanto al trabajo futuro, a lo largo de este informe, se mencionan varias tareas que no se realizan porque se escapan de los objetivos principales, o porque extienden el trabajo de memoria más allá de los limites de tiempo propuestos. Algunas de estas tareas que pueden ser retomadas en trabajos futuros son:

- Probar distintas variables de optimización para borrar aristas en Blocks (seccion 5.7.1).
- Optimizar búsqueda de vecinos y consultas de rango en Blocks, aprovechando la localidad de memoria de DFUDS.
- Extender Blocks para que soporte operaciones de rank en el último nivel.
- Extender Blocks para guardarlo en disco (y cargar a memoria).
- Extender Blocks para que soporte inserción/borrado de nodos de grafo.
- Extender Attk2DynTree2 para que soporte borrado de información.
- Extender Attk2DynTree2 para que soporte multi aristas.

Finalmente, se puede decir que se logra el objetivo general, pues la nueva versión de Attk2DynTree inserta y hace las consultas que involucran k^2-trees de manera más eficiente. Esto se logra gracias a que se cumplen los objetivos específicos. Se extiende Blocks para el borrado de aristas, se hacen experimentos, se concluye que Blocks es el mejor candidato para ser integrado en Attk2DynTree, y finalmente se obtiene una mejor versión de Attk2DynTree.

Bibliografía

[1] Sandra Álvarez-García y col. "Compact and Efficient Representation of General Graph Databases". En: *Knowledge and Information Systems* 60 (2018), págs. 1479-1510. arXiv: 1812.10977 [cs.DS].

[2] Diego Arroyuelo y col. "Faster Dynamic Compressed d-ary Relations". En: *String Processing and Information Retrieval: 26th International Symposium*. 2019, págs. 419-433. DOI: 10.1007/978-3-030-32686-9_30.

[3] Diego Arroyuelo y col. "LZ78 Compression in Low Main Memory Space". En: *International Symposium on String Processing and Information Retrieval*. 2017, págs. 38-50. DOI: 10.1007/978-3-319-67428-5_4.

[4] Paolo Boldi y Sebastiano Vigna. "The WebGraph Framework I: Compression Techniques". En: *Proceedings of the 13th International World Wide Web Conference*. 2004, págs. 595-601.

[5] Paolo Boldi y col. "Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks". En: *Proceedings of the 20th International Conference on World Wide Web*. 2011, págs. 587-596.

[6] Nieves R. Brisaboa y col. "Compressed Representation of Dynamic Binary Relations with Applications". En: *Information Systems* 69 (2017), págs. 106-123. arXiv: 1707.02769 [cs.DS].

[7] Miguel E. Coimbra y col. "On dynamic succinct graph representations". En: *2020 Data Compression Conference*. 2019, págs. 213-222. arXiv: 1911.03195 [cs.DS].

[8] F. Maxwell Harper y Joseph A. Konstan. "The MovieLens Datasets: History and Context". En: *ACM Trans. Interact. Intell. Syst.* 5.4 (dic. de 2015), págs. 1-19.

[9] Borislav Iordanov. "HyperGraphDB: A Generalized Graph Database". En: *Web-Age Information Management*. Springer Berlin Heidelberg, 2010, págs. 25-36.

[10] Veli Mäkinen y Gonzalo Navarro. "Dynamic Entropy-Compressed Sequences and Full-Text Indexes". En: *ACM Trans. Algorithms* 4.3 (2008), págs. 1-38. DOI: 10.1145/1367064.1367072.

[11] Norbert Martínez-Bazan y col. "DEX: High-performance exploration on large graphs for information retrieval". En: *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*. Ene. de 2007, págs. 573-582. DOI: 10.1145/1321440.1321521.

[12] G.M. Morton. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. International Business Machines Company, 1966.

[13] Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. 1st. 2016.

[14] A. Padrol-Sureda y col. "Overlapping Community Search for social networks". En: *26th International IEEE Conference on Data Engineering*. 2010, págs. 992-995.

[15] Nicola Prezza. "A Framework of Dynamic Data Structures for String Processing". En: *16th International Symposium on Experimental Algorithms*. Vol. 75. 2017, 11:1-11:15. DOI: 10.4230/LIPIcs.SEA.2017.11.

[16] Rajeev Raman, Venkatesh Raman y S. Srinivasa Rao. "Succinct Dynamic Data Structures". En: *Algorithms and Data Structures*. Berlin, Heidelberg, 2001, págs. 426-437.

[17] Sriram Raghavan y H. Garcia-Molina. "Representing Web graphs". En: *Proceedings of the 19th International Conference on Data Engineering*. 2003, págs. 405-416.

[18] Wen Sun y col. "SQLGraph: An Efficient Relational-Based Property Graph Store". En: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of*

Data. Melbourne, Victoria, Australia, 2015, págs. 1887-1901. DOI: 10.1145/2723372.2723732.