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

IMPLEMENTACIÓN DE LEAPFROG TRIEJOIN SOBRE ESTRUCTURAS DE DATOS COMPACTAS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERA
CIVIL EN COMPUTACIÓN

DANIELA CAMPOS FISCHER

PROFESOR GUÍA:
GONZALO NAVARRO BADINO

PROFESOR GUÍA 2:
DIEGO ARROYUELO BILLIARDI

SANTIAGO DE CHILE
2022

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE: Ingeniera Civil en Computación
POR: DANIELA CAMPOS FISCHER
FECHA: 2022
PROFESOR GUÍA 1: Gonzalo Navarro Badino
PROFESOR GUÍA 2: Diego Arroyuelo Billiardi

IMPLEMENTACIÓN DE LEAPFROG TRIEJOIN SOBRE ESTRUCTURAS DE DATOS COMPACTAS

Las operaciones de join en bases de datos resultan costosas, ya sea en el tiempo que toman, o en el espacio utilizado por los índices que resuelven dichas consultas.

En el último tiempo se han introducido los algoritmos Worst Case Optimal para joins. Estos algoritmos permiten resolver una consulta en tiempo lineal respecto al máximo tamaño posible de la respuesta de dicha consulta en alguna base de datos. El problema de estos algoritmos es que para poder responder de manera tan eficiente deben almacenar los índices en todas las permutaciones posibles de los atributos, y dependiendo de cómo se están indexando estos datos, el espacio ocupado puede llegar a ser muy grande.

En esta memoria se buscó implementar algoritmos Worst Case Optimal basados en estructuras de datos compactas, de modo de mitigar el impacto en el espacio que produce el representar todas las permutaciones posibles. Nos centramos en bases de datos de grafos, que indexan triples y requieren almacenar 6 permutaciones, y en el algoritmo Leapfrog Trie Join (LTJ), el Worst Case Optimal más popular. Se implementaron dos índices, LTJ IV y LTJ WM, ambos en C++. Nuestra implementación del algoritmo LTJ está abstraída de los índices que utiliza, lo que permite que futuros usuarios puedan implementar otra versión de un índice que siga la interfaz provista.

Nuestros índices se compararon con el estado del arte sobre un subconjunto de la Wikidata. El índice LTJ IV tuvo buenos resultados, siendo un 30 % más rápido que los índices clásicos con los que se lo comparó y ocupando sólo unas 3 veces el tamaño original de los datos que se indexaron, asimismo utilizó menos de un tercio del tamaño del menor índice no compacto Worst Case Optimal, y aproximadamente la mitad de los clásicos no Worst Case Optimal. Por otro lado el índice LTJ WM no tuvo tan buenos resultados, siendo más rápido que algunos de los índices comparados, pero en general teniendo tiempos de respuesta de consultas bastante altos. Junto con esto utiliza más espacio que LTJ IV, por lo que no resultó ser una buena solución en comparación.

Se concluye que los nuevos índices compactos son una buena alternativa a los clásicos tanto en espacio como en tiempo de consulta. En cambio, los nuevos índices no resultaron competitivos contra el Ring, un índice comprimido ya existente que es más sofisticado, pero que sólo funciona para tablas de tres atributos. Nuestra implementación es genérica y puede usarse en bases de datos relacionales generales.

Para todos los perritos que se podrán esterilizar

Agradecimientos

Gracias a mi familia por apoyarme y acompañarme en todo este proceso y además por dejarme convertir la casa en un zoológico. A Francisco(*Mano*) por siempre creer en mi y en consecuencia hacerme creer en mi misma. A Almendra, Frida, Kalu, Lanita, Palta, Porota y Tomás, por ser uno de mis pilares de apoyo y darme felicidad a diario. A mis amigas de la U, sin las cuales no hubiera llegado hasta aquí. A mis amigas del colegio por siempre estar ahí y creer en mis habilidades. Al Mati, por acompañarme recorriendo 1000 árboles y ser el mejor compañero que pudiera tener y a su familia por acogerme y alimentarme siempre que tenía que estudiar.

Tabla de Contenido

1. Introducción **1**

2. Estado del Arte **2**
2.1. Joins Naturales . . . 2
2.2. Global Attribute Order (GAO) . . . 2
2.3. Algoritmos Worst Case Optimal . . . 3
2.4. Leapfrog Triejoin . . . 3
2.5. Jena LTJ . . . 6
2.6. Ring . . . 6
2.7. Estructuras de Datos Compactas . . . 6
2.7.1. Bitvectors . . . 7
2.7.2. Representación LOUDS de un árbol . . . 7
2.7.3. Wavelet Trees . . . 8
2.7.4. Arrays . . . 9

3. Problema **11**

4. Solución **12**
4.1. Tries . . . 12
4.1.1. Trie Interface . . . 13
4.1.2. Compact Trie WM . . . 13
4.1.2.1. Métodos Importantes . . . 13
4.1.3. Compact Trie IV . . . 15
4.1.3.1. Métodos Importantes . . . 15
4.2. Configuración . . . 16
4.3. Construcción del Índice . . . 16
4.3.1. Table Indexer . . . 16
4.3.1.1. Métodos Importantes . . . 16
4.3.2. Index . . . 16
4.3.3. Proceso de Construcción . . . 17
4.4. Iteradores . . . 18
4.4.1. Iterator . . . 18
4.4.2. Compact Trie Iterator WM . . . 18
4.4.2.1. Métodos Importantes . . . 18
4.4.3. Compact Trie Iterator IV . . . 19
4.4.3.1. Métodos Importantes . . . 19
4.5. Resolución de Consultas . . . 20
4.5.1. Term . . . 20

4.5.2. Tuple 20
4.5.3. Leapfrog Join 21
4.5.3.1. Métodos Importantes 21
4.5.4. Clase LTJ 21
4.5.4.1. Construcción 22
4.5.4.2. Triejoin 23

5. Validación **25**
5.1. Datos 25
5.2. Consultas 25
5.3. Experimentos 25
5.3.1. Indexación 27
5.3.2. Resultados 27
5.3.3. Testeo en mayores dimensiones 30

6. Conclusiones **31**

Bibliografía **32**

Anexo A. **34**
A.1. Ejemplo indexación y resolución de consultas 34

Índice de Tablas

2.1. Ejemplo de Relación A 5
5.1. Espacio utilizado por Índices implementados 27
5.2. Espacio utilizado en bytes por triple y tiempo promedio de consulta en milisegundo para los sistemas comparados 28

Índice de Ilustraciones

2.1. Ejemplos de Relaciones para LTJ 4
2.2. Trie para la relación A 5
2.3. Representación LOUDS para la estructura de un árbol ordinal 8
2.4. Representación de secuencia S en Wavelet Tree 9
2.5. Representación compacta de un arreglo A 10
4.1. Procesos y clases asociadas 12
4.2. Conversión de Trie Regular a B y S 17
4.3. Ejemplificación del re-ordenamiento por el que pasan las consultas 22
5.1. Patrones de consultas de WGPB [6] 26
5.2. Comparación del tiempo de resolución de consultas(en segundos). Las cajas asociadas a cada índice van desde el 25 % al 75 % de los datos, con la mediana marcada dentro de la caja. Las líneas se extienden desde el mínimo valor al máximo, sin considerar valores atípicos. 29
A.1. Índice A construido a partir de los Datos A 34
A.2. Resolución de Consultas: Instanciación de LTJ 35
A.3. Estado inicial de los iteradores al llamar al método **triejoin** 36
A.4. Se aplica **triejoin_open** sobre todos los iteradores 36
A.5. Avance por constantes 37
A.6. Obtener valor para $?V2$ 37
A.7. Se baja un nivel en los iteradores asociados a $?V2$ 38
A.8. Se baja un nivel en el iterador asociado a $?V3$ 39
A.9. Se encuentra binding para variable $V4$ 39
A.10. Se sube un nivel en el iterador 012 para volver a buscar bindings para $?V3$ 40
A.11. Se avanza al siguiente posible valor para $?V3$ 41
A.12. Se obtiene valor para $?V4$ 41

Capítulo 1

Introducción

La evaluación de joins relacionales es uno de los problemas centrales y exhaustivamente estudiados en los sistemas de bases de datos, debido a lo costoso que resultan en términos de tiempo [1, 2]. En el último tiempo los pair-wise joins, que corresponden a aquellos algoritmos en los que se hace join de pares de relaciones, siendo necesario obtener resultados intermedios para obtener el resultado final, han sido reemplazados por multiway joins sobre un atributo en común, debido a que resultan subóptimos respecto a estos últimos, en los cuales se va avanzando simultáneamente por todas las relaciones con atributos en común. Este tipo de estrategias han permitido el desarrollo de algoritmos Worst Case Optimal [3].

Los algoritmos Worst Case Optimal son la clase de algoritmos de join que pueden resolver en tiempo $O(Q^*)$ una consulta de join cuyo resultado puede ser (en el peor de los casos) de tamaño Q^*.

El problema de estos algoritmos es que utilizan mucho espacio, ya que para poder lograr los tiempos Worst Case Optimal deben guardar varios órdenes de los índices con los que trabajan.

Un algoritmo que ha resultado bastante útil a la hora de resolver join es Leapfrog Triejoin. El enfoque de esta memoria es proveer variantes de la estructuras utilizadas por Leapfrog Triejoin que utilicen menos espacio. El espacio utilizado por este algoritmo va creciendo exponencialmente con la cantidad de dimensiones que se quieren indexar (cuántos atributos tienen las tablas indexadas), por lo tanto disminuir el espacio utilizado por algoritmos Worst Case Optimal podría traducirse en la posibilidad de indexar tablas de más dimensiones o mayor cardinalidad a las posibles hoy en día en memoria principal.

La solución consiste de una implementación del algoritmo Leapfrog Triejoin que utiliza iteradores para poder encontrar la respuesta a la consulta. Estos iteradores recorren tries, una estructura de datos utilizada para almacenar strings o secuencias en forma de un árbol etiquetado [4]. Estos tries indexan las filas de las tablas, en todos los órdenes posibles, por lo que se implementaron tries compactos.

Capítulo 2

Estado del Arte

2.1. Joins Naturales

Dado un conjunto de relaciones que posean al menos un atributo en común, se les puede aplicar join natural y la tabla resultante contendrá todos los atributos presentes en el conjunto de relaciones y los atributos que son compartidos por las tablas aparecerán una vez denotando los valores que son compartidos por ambas tablas [5].

Si consideramos el formato de consulta utilizado por Ring, una estructura para indexar bases de datos de grafos [6], el cual está inspirado en el formato utilizado por las consultas de SPARQL, podemos ver a continuación un ejemplo de una consulta de join natural:

$$?x_1 \; 1 \; ?x_2 \cdot ?x_2 \; ?x_3 \; ?x_4$$

El punto denota el join y los términos que están a cada lado corresponden a la descripción de los triples que buscamos. Para este ejemplo, el primer término de la consulta indica que se buscan todos los triples cuya arista tenga un valor de 1 y luego el segundo término nos indica que queremos todos los triples que estén conectados a los triples cuyo sujeto sea igual al objeto de las aristas encontradas para el primer término de la consulta.

2.2. Global Attribute Order (GAO)

Para responder consultas de manera óptima, se puede calcular el GAO de estas. El GAO de una consulta corresponde a un vector que contiene las variables encontradas en la consulta ordenadas de acuerdo a su prioridad. Esta prioridad luego es utilizada para re-ordenar la consulta de manera que pueda ser evaluada eficientemente. Si consideramos la siguiente consulta:

$$?x_1 \; ?x_2 \; ?x_3 \cdot ?x_3 \; ?x_4 \; ?x_5$$

Y un GAO igual a $[?x_3, ?x_2, ?x_1, ?x_4, ?x_5]$. Con una tabla cuyas columnas son s, p, o.

Para evaluar el primer término de la consulta se debe pasar del orden original (s, p, o) a (o, p, s)

y para evaluar el segundo término se puede mantener el orden original, ya que este calza con la prioridad definida por GAO.

El GAO no es necesario para responder consultas correctamente, pero utilizarlo permite recorrer las variables y datos disponibles de una manera óptima, logrando descartar triples como respuesta del join, y de esta manera se entregan respuestas de una manera más eficiente.

2.3. Algoritmos Worst Case Optimal

Los algoritmos Worst Case Optimal pueden resolver una consulta join en tiempo $O(Q^*)$, con Q^* siendo la cota AGM de dicha consulta [7]. La cota AGM define el máximo número de resultados para una consulta join sobre relaciones con una cantidad determinada de columnas y filas. Por lo tanto, todo algoritmo de join Worst Case Optimal debe enumerar Q^* resultados, tomando tiempo $O(Q^*)$ en el peor caso [1, 2].

Una manera tradicional de evaluar consultas joins es hacerlo con una estrategia pair-wise join, es decir, si una consulta consiste de joins entre varias relaciones, se computarán joins intermedios de pares de relaciones y se repetirá este proceso con los resultados intermedios hasta obtener el resultado final [1]. Este enfoque no es Worst Case Optimal porque con algunas instancias de bases de datos, los resultados intermedios podrían ser más grandes que el resultado final y por lo tanto el tiempo del algoritmo estaría definido por los resultados intermedios, no permitiendo un tiempo Worst Case Optimal.

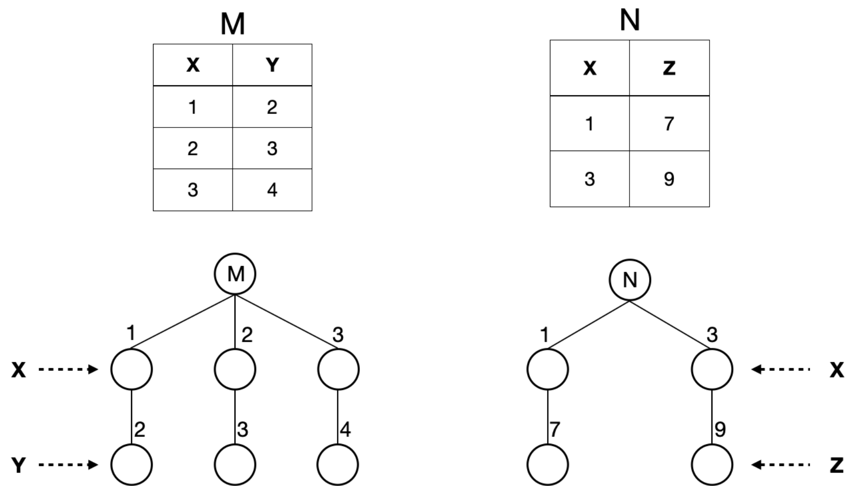
2.4. Leapfrog Triejoin

El algoritmo Leapfrog Triejoin fue presentado como una nueva propuesta para obtener algoritmos Worst Case Optimal. Este realiza join de todos los términos (o relaciones) de una consulta de manera simultánea sin producir resultados intermedios. Esto se lleva a cabo instanciando una variable a la vez y moviéndose por todas las relaciones que poseen dicha variable al mismo tiempo. Para poder responder las consultas de manera óptima, el algoritmo requiere una cantidad exponencial de órdenes de índices respecto a la aridad de las relaciones con las que está trabajando, estos diferentes órdenes son almacenados en tries [2]. Un Trie es una estructura de datos con forma de árbol en la que un camino desde la raíz a una hoja corresponde a una palabra del conjunto que se desea representar [8], en este caso el conjunto a representar es la tabla de datos indexada y la palabra corresponde a una de las filas de dicha tabla.

Como las variables pueden requerir instanciarse en orden arbitrario, los órdenes de índices que Leapfrog Triejoin requiere corresponden a permutaciones del orden original que describe a una relación. Es decir, si la tabla que describe a una relación tiene el orden (x,y,z) será necesario indexar (x,y,z), (x,z,y), (y,x,z), (y,z,x), (z,x,y) y (z,y,x). Esto se debe a que diferentes consultas requerirán diferentes órdenes de atributos para poder ser contestadas de manera óptima, por lo tanto a la hora de la construcción del índice es necesario construir todas las permutaciones.

Leapfrog Triejoin es Worst Case Optimal hasta un factor logarítmico, esto quiere decir que logra

un tiempo de $O(Q^* \log(n))$, donde Q^* es el límite AGM para la consulta y n es la cardinalidad más grande entre las relaciones de la consulta de join [2].



X	Y
1	2
2	3
3	4

X	Z
1	7
3	9

Figura 2.1: Ejemplos de Relaciones para LTJ

A continuación utilizaremos las relaciones descritas en la Figura 2.1 para ejemplificar el funcionamiento de LTJ. Si se desea hacer el join natural entre la relación M y N, se recorrerán sus respectivos tries de la siguiente manera: cómo se puede ver el atributo compartido por ambas relaciones es X, correspondiente al primer nivel de ambos árboles, por lo que se buscará el primer valor que ambos tries compartan en dicho nivel, en este caso el 1, luego se bajará un nivel en ambos tries. En este punto ya no hay más atributos compartidos por lo que sólo se debe asignar a las variables Y y Z los valores que corresponden, que serían 2 y 7 respectivamente, de esta manera encontramos la primera tupla (X,Y,Z) igual a $(1,2,7)$. Luego se sube por ambos árboles hasta llegar al atributo en común X y se repite el proceso de buscar un valor que se encuentre en ambos tries, ahora sin considerar al 1, encontrando al 3. Al igual que la vez anterior se baja por ambos árboles encontrando una segunda tupla (X,Y,Z) igual a $(3,4,9)$, al volver a subir hasta el atributo en común X ya no es posible seguir encontrando resultados por lo que el resultado del join corresponde a las dos tuplas encontradas.

Leapfrog Triejoin utiliza interfaces de iteradores para unificar las representaciones de las relaciones del input. Para representar relaciones n-arias utiliza una interfaz llamada Trie Iterator, donde cada tupla $\{x_1,...,x_n\}$ corresponde a un único camino desde la raíz del trie hasta una hoja. Por ejemplo para la relación de la Tabla 2.1, el trie que la representaría sería el de la Figura 2.2, donde cada uno de los niveles del trie corresponde a todos los valores en la relación de cada uno de sus atributos.

Para poder moverse en paralelo por los tries de cada relación que involucra al join y de esa manera obtener los resultados, el Trie Iterator utiliza un puntero por relación, el que se va moviendo por los valores del trie. Para lograr esto es necesario que el Trie Iterator cuente con ciertas operaciones, que deben cumplir con una complejidad estipulada, ya que de lo contrario el algoritmo no sería Worst Case Optimal. Si consideramos que se está trabajando con una relación de cardinalidad

Tabla 2.1: Ejemplo de Relación A

x	y	z
1	1	4
1	1	5
1	2	6
1	2	7
1	2	8
1	3	2
3	1	2

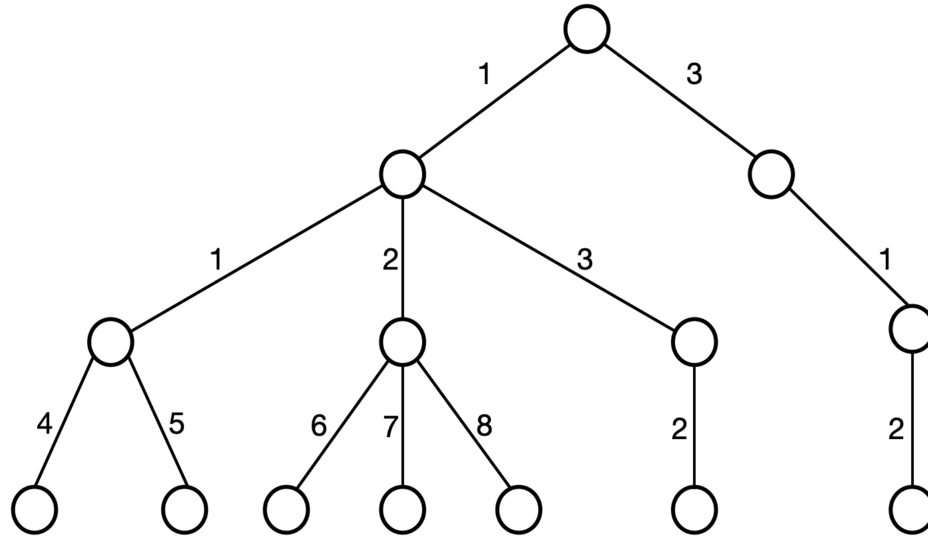


Figura 2.2: Trie para la relación A

n, las operaciones del Trie Iterator son:

- **key():** Retorna la llave en la que está parado el puntero. Por ejemplo, si el puntero se encuentra en el primer hijo de la raíz del trie de la Figura 2.2, **key()** retornará 1. Su complejidad debe ser constante, $O(1)$.
- **next():** Mueve el puntero a la siguiente llave del padre de la llave actual, de no haber una siguiente llave se determina que el puntero está *al final*. Por ejemplo, si el puntero se encuentra en el primer hijo de la raíz del trie de la Figura 2.2, **next()** lo moverá al segundo hijo de la raíz. Su complejidad debe ser logarítmica, $O(\log(n))$.
- **seek(seekKey):** Mueve el puntero a la menor posición que sea mayor o igual a **seekKey** dentro de los hijos del padre actual. De no existir dicha posición se dice que el puntero está *al final*. Por ejemplo, si el puntero está en el primer hijo de la raíz del trie de la Figura 2.2, **seek(2)** nos llevaría al segundo hijo de la raíz ya que su llave es 3 y este es el primer valor mayor o igual a 2 dentro de los hijos de la raíz. Su complejidad debe ser $O(1+\log(n/m))$ con m correspondiendo a m operaciones seek con valores crecientes.
- **atEnd():** Retorna *true* si el puntero se encuentra *al final*.

- **open():** Mueve el puntero a la primera llave del siguiente nivel que es hija del nodo en el que el puntero se encuentra actualmente. Por ejemplo, si el puntero está en la raíz del trie de la Figura 2.2, **open()** llevará al puntero al primer hijo de la raíz, que está asociado a la llave 1. Su complejidad debe ser logarítmica, $O(\log(n))$.

- **up():** Mueve el puntero al padre de la llave actual en el nivel previo. Por ejemplo, si el puntero está en el primer hijo de la raíz del trie de la Figura 2.2, **up()** lo moverá a la raíz. Su complejidad debe ser logarítmica, $O(\log(n))$.

2.5. Jena LTJ

Leapfrog Triejoin es usado en el contexto de bases de datos RDF, ya que las implementaciones disponibles para joins en este tipo de base de datos aún tenían problemas con consultas que incluyeran joins complejos [9]. Es por esto que se propuso un nuevo procedimiento para evaluar consultas SPARQL utilizando Leapfrog Triejoin y este fue implementado en Apache Jena. El resultado fue Jena LTJ, un algoritmo Worst Case Optimal para patrones de grafos básicos, que fue implementado indexando los datos en B+-trees. Esta nueva versión de Jena redujo los tiempos de consultas con joins no triviales en varios órdenes de magnitud, en comparación a Apache Jena sin Leapfrog Triejoin [9].

En [6] se comparó el funcionamiento de varios algoritmos Worst Case Optimal junto con sistemas de bases de datos de grafos, dentro de los que estaba Jena y Jena LTJ. Se utilizó un subgrafo de la Wikidata con alrededor de 932 MB de datos. Jena LTJ resultó ser más costoso en espacio que la mayoría de los índices testeados, pero estuvo dentro de los índices más eficientes a la hora de responder consultas de join [6].

2.6. Ring

Una nueva estructura para indexar bases de datos de grafos llamada Ring [6], utiliza espacio sub-lineal, sobre el necesario para guardar el grafo. Debido a que se utiliza para almacenar grafos, indexa triples, es decir, un conjunto sujeto-predicado-objeto, representando la arista $s \xrightarrow{p} o$ del grafo como el triple (s, p, o). El Ring guarda cada triple como un string cíclico y bidireccional de largo tres y fue implementado utilizando técnicas de indexación de texto. Esto le permite simular los $3! = 6$ órdenes posible con solo una copia del grafo.

El Ring utiliza estructuras compactas para almacenar los triples, por lo que es muy eficiente en su uso de espacio. De momento no existe una implementación de Ring para bases de datos de dimensiones mayores a tres.

2.7. Estructuras de Datos Compactas

Como vimos, en Leapfrog Triejoin y Jena LTJ, los algoritmos que soportan joins Worst Case Optimal usando estructuras de datos convencionales necesitan mucho espacio. Para representar

una relación de d atributos necesitan $d!$ órdenes de índices para poder cumplir tiempos Worst Case Optimal [6]. Por ello es relevante buscar representaciones compactas para estos tries.

Estas estructuras compactas se encuentran implementadas con todas sus operaciones en la librería : SDSL - Succinct Data Structure Library. Cabe destacar que para poder acceder a los valores almacenados en cualquiera de estas estructuras compactas no es necesario descomprimirlas, si no que tienen operaciones implementadas para poder acceder a valores estando en su versión compacta.

Dentro de las estructuras compactas que resultan útiles para simular un trie encontramos a la representación LOUDS de un árbol, los Wavelet Trees y los Arrays. Para comprender el funcionamiento de éstas es necesario tener una noción de qué son los Bitvectors[4].

2.7.1. Bitvectors

Un bitvector es un arreglo de bits B de tamaño n, que soporta las siguientes operaciones:

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

Los bitvectors son parte fundamental y basal de muchas estructuras compactas.

2.7.2. Representación LOUDS de un árbol

Para representar un árbol ordinal de manera compacta, se pueden utilizar $2n$ bits, en vez de ocupar los $O(n)$ punteros que ocupan las representaciones tradicionales. Hay tres tipos de representaciones de árboles compactos, LOUDS, BP y DFUDS. En este trabajo sólo nos enfocaremos en LOUDS debido a su rapidez, simplicidad y debido que su funcionalidad es la necesaria para simular la estructura de un trie.

LOUDS es el acrónimo de Level-Order Unary Degree Sequence. Para representar un árbol de n nodos, se utiliza un bitvector de tamaño $2n+1$. Para instanciar la representación LOUDS de un árbol partimos con un bitvector con sólo dos valores, 10. Luego es necesario recorrer el árbol por niveles, de izquierda a derecha y cada nodo v con c hijos será descrito en el bitvector como 1^c0, es decir un 1 por cada hijo que tenga y un 0 para terminar su descripción [4]. En la Figura 2.3 podemos ver un ejemplo de la representación en LOUDS de un árbol.

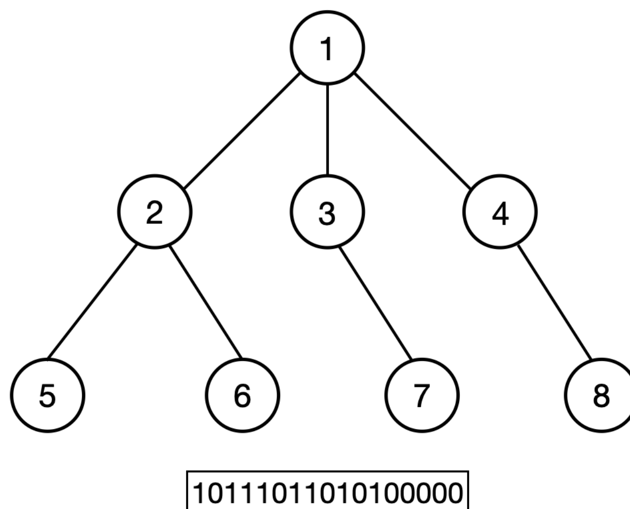


Figura 2.3: Representación LOUDS para la estructura de un árbol ordinal

2.7.3. Wavelet Trees

Los Wavelet Trees son una estructura compacta utilizada para almacenar secuencias de símbolos. Utilizan una partición jerárquica del alfabeto de la secuencia es decir, cada nivel del árbol divide el alfabeto disponible en dos partes de aproximadamente igual tamaño y en los nodos se guarda un bitvector del tamaño de la secuencia indicando con un 1 si un carácter dado está en el alfabeto correspondiente al hijo derecho y un 0 si está en el izquierdo [4].

Si se quiere almacenar la secuencia S y el alfabeto de símbolos $[1,\sigma]$, se dividirá el alfabeto en dos partes que sean aproximadamente iguales, $[1,\lceil\sigma/2\rceil]$ y $[\lceil\sigma/2\rceil+1,\sigma]$, y se creará el bitvector $B_{\langle 1,\sigma\rangle}[1,n]$ donde $B_{\langle 1,\sigma\rangle}[i]=0$ si y sólo si $S[i]\in[1,\lceil\sigma/2\rceil]$ y 1 en el caso contrario. Este proceso se repite recursivamente.

Los Wavelet Trees soportan las siguientes operaciones de manejo de bits:

- $access(i)$: Retorna el símbolo del alfabeto que se encuentra en S[i] en la secuencia original.
- $rank_c(i)$: Retorna la cantidad de caracteres c que hay en $S[1,i]$.
- $select_c(j)$: Retorna la j-ésima ocurrencia de c en S.

En la Figura 2.4 podemos ver la representación de la secuencia “mississipi” utilizando un Wavelet Tree. Se muestran en negro los elementos que efectivamente conforma el Wavelet Tree (los bitvectors) y en gris se ve conceptualmente qué es lo representado por cada nodo. Para responder las operaciones presentadas anteriormente sólo se requieren los bitvectors y conocer el alfabeto con el que se está trabajando.

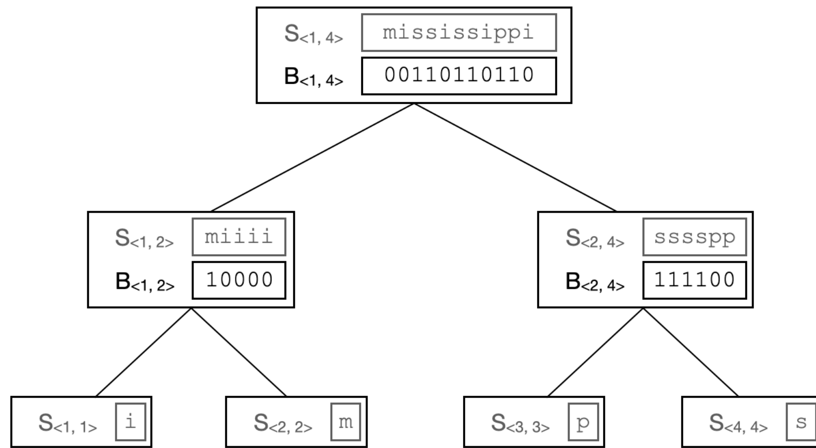


Figura 2.4: Representación de secuencia S en Wavelet Tree

2.7.4. Arrays

Un arreglo $A[1,n]$ es una secuencia de enteros a la cual se le pueden hacer lecturas y escrituras en posiciones arbitrarias. Permiten las siguientes operaciones:

- $read(A,i)$: retorna $A[i]$, para cualquier $1 \leq i \leq n$.
- $write(A,i,x)$: establece $A[i] = x$, para cualquier $1 \leq i \leq n$ y cualquier x.

Si todos los elementos de la secuencia pueden ser almacenados en l bits, es posible almacenarla utilizando tan solo $l \cdot n$ bits y no ocupando la cantidad tradicional de $w \cdot n$ bits, con $w \in \{8, 16, 32, 64\}$ correspondiendo a los tamaños predefinidos para guardar números.

Para almacenar la secuencia se definirá un arreglo de enteros (w bits por elemento) $W[1, \lceil ln/w \rceil]$, que es suficiente para codificar n elementos de l bits. Consideramos los ln bits almacenados en W como un bitvector virtual $B[1, ln]$. Cada elemento $A[i]$ se guardará en $B[(i-1)l+1, il]$. El arreglo W particiona los bits de B en bloques de 32 bits y se lo puede ver como un arreglo de $\lceil ln/w \rceil$ números.

En la Figura 2.5 podemos ver cómo se almacena un arreglo A de $n = 7$ elementos. Cada uno de estos elementos puede ser almacenado en $l = 5$ bits, lo que se ve representado en el bitvector B. Luego W tiene un tamaño de 2 ya que almacena 64 bits, los primeros 32 bits de B pasan a ser el primer número de W y los restantes pasan a ser el segundo.

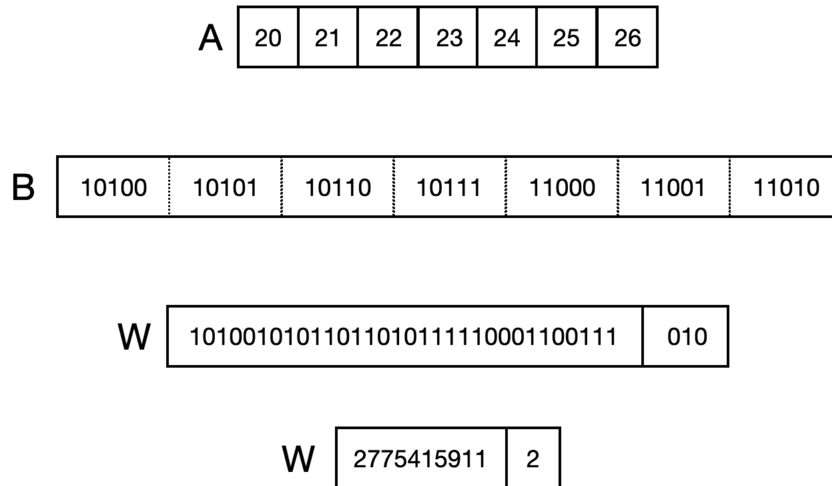


Figura 2.5: Representación compacta de un arreglo A

Capítulo 3

Problema

Los algoritmos Worst Case Optimal son muy eficientes para resolver consultas de join, pero tienden a utilizar mucho espacio, ya que se deben guardar los datos ordenados de diferentes maneras.

Leapfrog Triejoin es un ejemplo de un algoritmo Worst Case Optimal que requiere una cantidad exponencial de órdenes de índices respecto a la aridad de las relaciones con las que está trabajando.

Ya existen ejemplos de implementaciones que utilizan Leapfrog Triejoin, como Jena LTJ y Ring. La primera como una versión que utiliza tries clásicos y la segunda como una versión que no utiliza tries, pero sí estructuras compactas. Jena LTJ responde consultas eficientemente pero utiliza mucho espacio y Ring también responde consultas eficientemente y además utiliza poco espacio pero está limitado a tablas de aridad 3.

Sería valioso contar otra implementación basada en Leapfrog Triejoin que utilice tries pero construidos en base a estructuras compactas. Además sería útil poder indexar tablas de cualquier aridad. De esta manera se podría ocupar como punto de comparación en futuros trabajos. Junto con esto resultaría valioso contar con una versión de Leapfrog Triejoin que utilizara la interfaz para los Trie Iterators descrita en el paper [2], de esta manera, si se quisieran seguir estudiando otras variantes de iteradores sólo bastaría con implementar una versión que herede la interfaz.

La solución debe poder indexar tablas de cualquier dimensión, cargar índices y resolver consultas. Particularmente a la hora de indexar las tablas, se debe poder elegir qué órdenes se desean indexar, así si se conocen los órdenes requeridos para las consultas no se indexan órdenes de más. Además debe proveer una interfaz para los Trie Iterators que pueda ser utilizada en futuros trabajos.

Esta solución sería considerada como una solución lo suficientemente buena si puede indexar tablas de cualquier dimensión razonable y además utiliza menos espacio que Jena LTJ y es competitiva en tiempo con Ring en dimensión 3.

Capítulo 4

Solución

La relación entre las clases y procesos descritos en esta sección pueden ser apreciada en la Figura 4.1

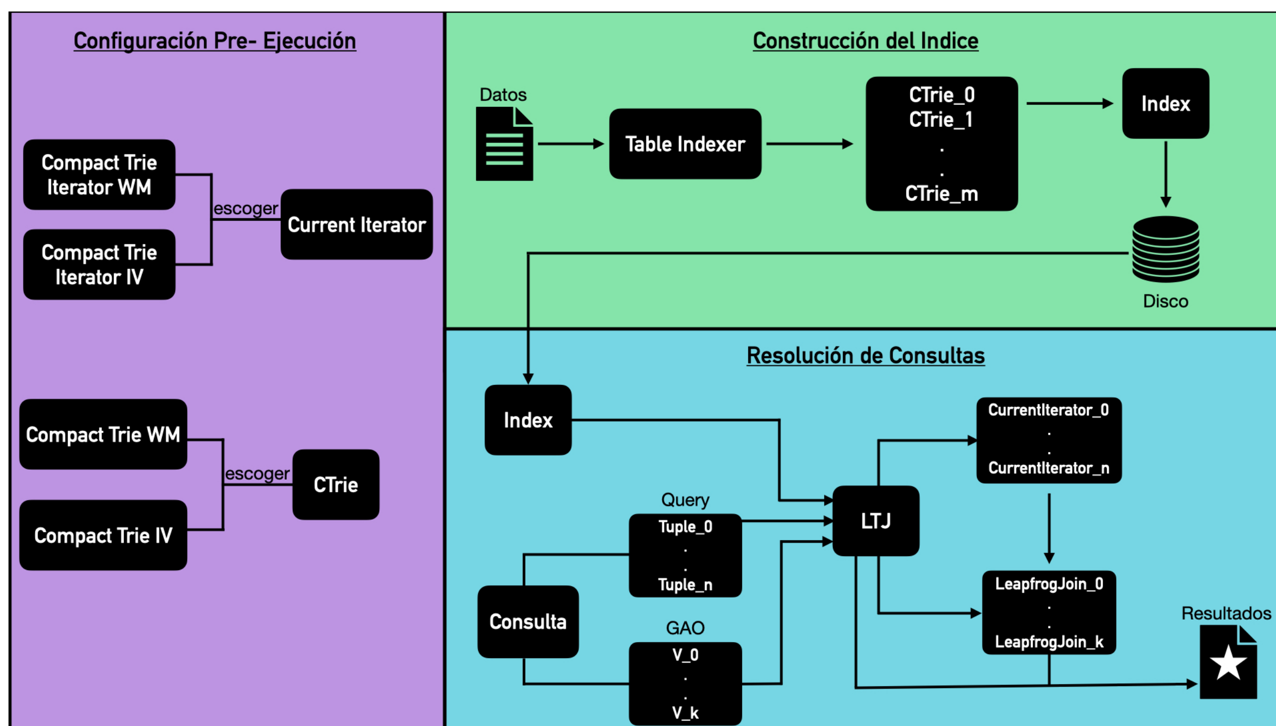


Figura 4.1: Procesos y clases asociadas

4.1. Tries

Los tries son la estructura central de la solución. Estos conforman en gran parte el índice que posteriormente es consultado. Se implementó una interfaz **Trie Interface** y a partir de ésta surgieron dos variantes, **Compact Trie WM** y **Compact Trie IV**. Ambas variantes representan a un trie, guardando la estructura de este en un bitvector, su diferencia radica en la manera de almacenar las llaves del Trie que representan, **Compact Trie WM** las almacena en una Wavelet Matrix y **Compact Trie IV** en un int_vector, que corresponde a un arreglo de enteros compacto.

4.1.1. Trie Interface

Es una clase abstracta que posee los siguientes métodos vituales:

- **size():** Debe retornar el tamaño en bytes utilizado por el trie.
- **child(it,n)**: Recibe dos enteros *it* y *n*. Debe retornar la posición del *n-ésimo* hijo del nodo en la posición *it*.
- **childrenCount(it):** Recibe un entero *it*. Debe retornar la cantidad de hijos que el nodo ubicado en *it* posee.
- **childRank(it):** Recibe un entero *it*. Debe retornar el índice del nodo ubicado en *it* dentro de los hijos de su padre.
- **parent(it):** Recibe un entero *it*. Debe retornar la posición del padre del nodo ubicado en *it*.
- **keyAt(it):** Recibe un entero *it*. Debe retornar la llave asociada al nodo ubicado en *it*.
- **storeToFile(file_name):** Recibe un string *file_name*. Debe pasar el trie a disco, guardándolo en el archivo *file_name*.
- **loadFromFile(file_name):** Recibe un string *file_name*. Debe cargar el trie a memoria desde el archivo *file_name* en el disco.

Estos métodos deben ser implementados por todas las variantes de trie que se implementen.

4.1.2. Compact Trie WM

Esta clase implementa la interfaz descrita en Trie Interface. Está conformada principalmente por dos estructuras: un bitvector *B* que alberga la estructura del árbol en LOUDS y un wavelet tree *wt* que alberga la secuencia de llaves del árbol *S*.

Utiliza la variante wm_int de los wavelet trees disponibles en la librería SDSL. Esta corresponde a una wavelet matrix, la cual fue seleccionada ya que se espera utilizar el índice en datos con alfabetos grandes y esta variante utiliza menos espacio que las otras en este contexto, pero es un poco más lenta.

4.1.2.1. Métodos Importantes

- **Constructor 1:** Recibe un bitvector *b* y un string *s*. El bitvector *b* es almacenado en *B* y el string *s* correspondiente a la secuencia *S* es utilizada para construir la wavelet matrix. Además inicializa las estructuras necesarias para poder aplicar operaciones de **rank** y **select** sobre el bitvector *B*.
- **Constructor 2:** Recibe un string *s*, correspondiente al nombre del archivo de donde hay almacenado un Compact Trie WM. Llama a loadFromFile para restaurar a memoria principal el Compact Trie WM. Además inicializa las estructuras necesarias para poder aplicar operaciones de **rank** y **select** sobre el bitvector *B*.

- **size():** Retorna el tamaño en bytes del Compact Trie WM. Esto comprende el tamaño en bytes del bitvector *B*, la wavelet matrix y las estructuras necesarias para poder aplicar operaciones de **rank** y **select** sobre el bitvector *B*.

- **child(it,n):** Recibe dos enteros *it* y *n*. Retorna la posición del *n*-ésimo hijo del nodo cuya descripción comienza en *it*. Debido a que la estructura del árbol almacenada en el bitvector B, *it* denota un índice en este vector. Para calcular la posición del hijo se utiliza **rank**, que permite obtener la cantidad de unos hasta $it + n$ (índice que en la descripción del nodo padre, identifica al *n*-ésimo hijo), esto permite saber qué nodo es el *n*-ésimo hijo. Luego utilizando **select** se calcula la posición del cero que precede a la descripción del nodo que buscamos. Finalmente se retorna esta posición más uno, que corresponde al inicio de la descripción del *n*-ésimo hijo del nodo posicionado en *it*.

- **childernCount(it):** Recibe un entero *it*. Retorna la cantidad de hijos del nodo cuya descripción comienza en la posición *it* del bitvector *B*. Esto se hace calculando la posición del cero que sucede a la descripción del nodo y luego restándole *it*, esto nos entregará la cantidad de unos en la descripción del nodo, lo que corresponde con la cantidad de hijos de dicho nodo.

- **getPositionInParent(it):** Recibe un entero *it*. Retorna el índice en el bitvector B del bit que identifica al nodo posicionado en *it* en la descripción de su padre. Comienza utilizando **rank** para obtener la cantidad de ceros que anteceden a *it*, lo que nos indica la cantidad de nodos que han sido descritos hasta ese punto. Luego utilizando **select** de unos sobre el valor anterior obtendremos la posición en la descripción de su padre del nodo en *it*.

- **childRank(it):** Recibe un entero *it*. Retorna qué hijo es el nodo cuya descripción comienza en *it* dentro de los hijos de su padre. Primero obtiene la posición dentro de la descripción de su padre del nodo posicionado en *it* utilizando getPositionInParent. Posteriormente a esta posición se le resta la posición del cero que antecede a la descripción del padre, esto nos entrega qué índice tiene el nodo dentro de los hijos de su padre.

- **parent(it):** Recibe un entero *it*. Retorna la posición del padre del nodo cuya descripción comienza en *it* en el bitvector *B*. Para realizarlo utiliza getPositionInParent con el objetivo de obtener el índice del bit que describe al nodo *it* en su padre. Luego calcula la posición del cero que antecede a este índice y le suma uno para posicionarse al inicio de la descripción del padre de *it*.

- **keyAt(it):** Recibe un entero *it*. Retorna el valor de la llave asociada al nodo cuya descripción comienza en *it* en el bitvector *B*. Inicialmente se calcula el índice en la secuencia asociado al nodo utilizando **rank** para obtener cuantos ceros hay antes de *it*, obteniendo la cantidad de nodos que han sido descritos hasta ese punto. Posteriormente a este valor le restamos dos ya que el 0 inicial en *B* no describe ningún nodo y el segundo 0 corresponde a la raíz, que no tiene llave asociada. Así obtenemos la posición dentro de la secuencia de la llave, para obtener la llave simplemente utilizamos la operación **access** de la wavelet matrix.

- **storeToFile(file_name):** Recibe un string file_name, correspondiente al nombre del archivo dónde se desea guardar el Compact Trie WM. Se utiliza el método store_to_file, provisto por la librería SDSL para guardar en disco sus estructuras. El bitvector *B* es guardado en un archivo con el nombre indicado en file_name utilizando la extensión .B y lo mismo se hace para la wavelet matrix, pero ocupando la extensión .WM.

- **loadFromFile(file_name):** Recibe un string file_name, correspondiente al nombre del archivo de donde se debe extraer el Compact Trie WM. Se utiliza el método load_from_file, provisto por la librería SDSL, para cargar del disco el bitvector *B* y la wavelet matrix.

4.1.3. Compact Trie IV

Esta clase implementa la interfaz descrita en Trie Interface. Está conformada principalmente por dos estructuras: un bitvector *B*, que al igual que en la clase Compact Trie WM alberga la estructura del árbol y un int_vector *sequence* para guardar la secuencia de llaves del árbol S. Los int_vector corresponden a arreglos de enteros que pueden ser compactados como fue descrito en la sección 2.5.4.

4.1.3.1. Métodos Importantes

Esta clase comparte varios métodos con la clase Compact Trie WM, particularmente todos los métodos que sólo involucran moverse por la estructura del árbol. Esto se debe a que ambas clases guardan esta estructura en un bitvector *B*. Es por esto que no ahondaremos en los métodos que son equivalentes a los que ya fueron presentados.

- Los métodos **child**, **childrenCount**, **getPositionInParent**, **childRank** y **parent**, junto con el Constructor 2 son análogos a lo encontrado en Compact Trie WM.

- **Constructor 1:** Recibe un bitvector *b* y un string *s*. El bitvector *b* es almacenado en *B* y el string s, correspondiente a la secuencia S es almacenado en el int_vector *sequence*. Además se inicializan todas las estructuras necesarias para poder aplicar operaciones de **rank** y **select** sobre el bitvector *B*.

- **size():** Retorna el tamaño en bytes del Compact Trie IV. Esto comprende el tamaño en bytes del bitvector *B*, el int_vector *sequence* y de las estructuras necesarias para poder aplicar operaciones de **rank** y **select** sobre el bitvector *B*.

- **keyAt(it):** Recibe un entero *it*. El cálculo del índice en la secuencia, de la llave asociada al nodo ubicado en *it* es análogo a lo hecho en Compact Trie. Una vez obtenida la posición a la que hay que acceder en la secuencia se lee el int_vector en dicha posición.

- **binarySearchSeek(value,i,f):** Recibe tres enteros, *value*, *i* y *f*. Se implementa una búsqueda binaria para encontrar el valor *value* dentro de el int_vector *sequence* en el rango de posiciones entre *i* y *f* inclusive. Retorna un par (*value*, *index*) si se encontró el valor *value* en la posición *index* de *sequence*, de lo contrario retorna el par $(0, f+1)$. La variante que utiliza una Wavelet Matrix no tiene un simil a este método, debido a que la implementación de la Wavelet Matrix posee métodos que nos permiten acceder a la misma información de manera eficiente.

- **storeToFile(file_name):**Recibe un string file_name, correspondiente al nombre del archivo dónde se desea guardar el Compact Trie IV. Comprime al int_vector *sequence* utilizando el método bit_compress de la SDSL. Se utiliza el método store_to_file, provisto por la librería SDSL para guardar en disco sus estructuras. El bitvector *B* y el int_vector *sequence* son guardados en archivos con el nombre indicado en file_name utilizando la extensión .B y .IV respectivamente.

- **loadFromFile(file_name):** Recibe un string file_name, correspondiende al nombre del archivo de dónde se debe extraer el Compact Trie IV. Se utiliza el método load_from_file, provisto por la librería SDSL, para cargar del disco el bitvector B y el int_vector *sequence*. No descomprime el int_vector.

4.2. Configuración

Existe un archivo de configuración donde se puede elegir qué variante de la implementación se desea ocupar. Existen las variables globales CurrentIterator y CTrie, las cuales deben ser asignadas a la variante del iterador y el trie compacto que se desea ocupar respectivamente.

4.3. Construcción del Índice

Comenzaremos describiendo la clase que se encarga del procesamiento de los datos y la instanciación del índice, luego entraremos en detalle sobre cómo está implementado dicho índice para finalmente explicar cómo es el proceso de construcción de éste.

4.3.1. Table Indexer

Esta clase es quien se encarga de procesar los datos que se desean indexar y de posteriormente crear el índice.

4.3.1.1. Métodos Importantes

- **readTable(file_name):** Recibe un string file_name. Lee el archivo file_name que contiene la tabla a ser indexada, asegurándose que tenga el formato apropiado, el cual consiste en una primera línea que describe la dimensión de la tabla, una segunda línea que describe los órdenes que desean ser indexados y luego el resto del archivo que contiene las tuplas de la dimensión indicada. Lee todas las tuplas y las guarda en una matriz *table*.
- **createTries():** Para cada uno de los órdenes que deben ser indexados, se construye un trie regular (no una versión compacta) recorriendo la matriz *table* acorde al orden y creando un camino desde la raíz hasta una hoja por cada tupla. Luego este trie es traspasado a forma compacta recorriendo el árbol y armando un bitvector B que almacena la estructura y una secuencia S que almacena las llaves ordenadas por nivel. Finalmente se construye un CTrie entregándole el bitvector B y la secuencia S y éste se agrega al vector de CTries que posteriormente será entregado al índice. En la Figura 4.2 se puede ver cómo el trie conformado por los valores de la Tabla 2.1 pasa a convertirse en un bitvector B y una secuencia S.
- **indexNewTable(file_name):** Recibe un string file_name. Utiliza el método **createTries** para obtener el vector de CTries que conformarán el índice. Finalmente crea un objeto de la clase Index (descrita a continuación) entregandole la dimensión, los órdenes indexados, un vector de CTries con un CTrie por orden indexado y el nombre del archivo file_name.

4.3.2. Index

Esta clase tiene el objetivo de representar al índice completo. Esto quiere decir que alberga los diferentes órdenes indicados en la lectura de los datos y los CTries asociados a estos órdenes.

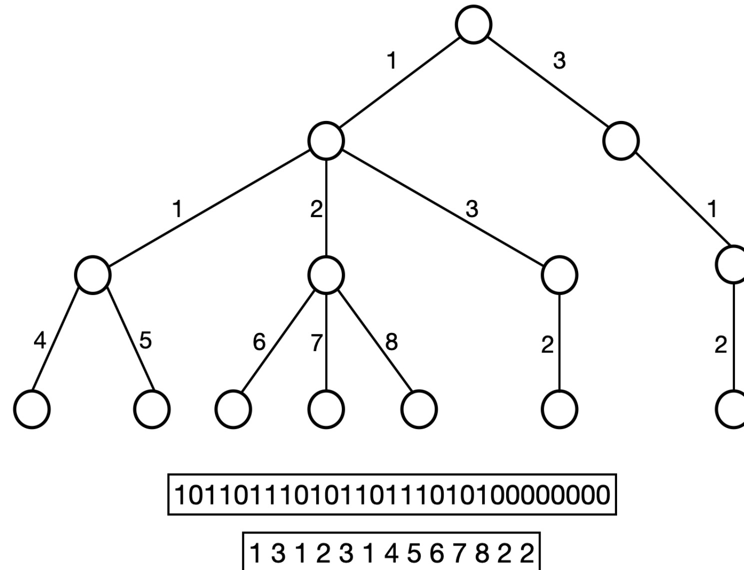


Figura 4.2: Conversión de Trie Regular a B y S

Además alberga la dimensión de la tabla indexada. Posee los siguientes métodos:

- **Constructor 1:** Recibe la dimensión del índice, un vector de strings conteniendo los órdenes indexados, un vector de CTries conteniendo un árbol por orden y el nombre del archivo que contenía la tabla que se está indexando. Almacena los árboles en un mapa, con la llave correspondiendo al orden que dicho trie representa.

- **Constructor 2:** Recibe un string correspondiente al nombre de la carpeta dónde está almacenado el índice. Carga el índice desde dicha carpeta utilizando el método **load**.

- **size():** Calcula el tamaño del índice en bytes.

- **save:** Guarda la representación del índice en una carpeta con el mismo nombre del archivo original que almacenaba la tabla, pero sin la extensión. Se escribe un archivo info.txt que contendrá la dimensión del índice y los órdenes que fueron indexados. Luego para cada CTrie se llama a su método storeToFile que lo guardará con el formato pertinente a la implementación de CTrie que fue elegida.

- **load:** Carga la representación del índice desde su carpeta.

- **getTrie(order):** Recibe un string *order*. Retorna el CTrie cuya llave corresponde a *order* en el mapa que alberga los CTries.

4.3.3. Proceso de Construcción

Para construir el índice a partir de una tabla se instancia un objeto de la clase Table Indexer y luego se utiliza al método readTable del Table Indexer para procesar los datos. Posteriormente se llama al método indexNewTable que crea el índice a partir de los datos obtenidos, para finalmente pasarlo a disco.

4.4. Iteradores

En esta sección describiremos las variantes de iteradores disponibles para ocupar con la solución. Estos iteradores serán los que nos permitan encontrar las respuestas a las consultas.

4.4.1. Iterator

La clase Iterator es una clase abstracta que actúa como una interfaz para los iteradores. Esto nos permite que el proceso de resolución de consultas sea agnóstico respecto a qué iterador está utilizando. Contiene los siguientes métodos virtuales:

- **atEnd():** Debe retornar *true* si el iterador está *al final*. Recordemos que un iterador está *al final* cuando se intenta avanzar al siguiente hijo de un nodo y este no tiene otro hijo.
- **key():** Debe retornar la llave asociada al nodo en el que el iterador está posicionado.
- **seek(seekKey):** Debe mover el iterador a la posición del siguiente nodo cuya llave sea mayor o igual a *seekKey*. De no existir dicha llave, debe posicionar al iterador *al final*.
- **next():** Debe mover el iterador al siguiente nodo, es decir al hermano derecho del nodo actual. De no existir dicho hermano debe posicionar al iterador *al final*.
- **open():** Debe mover el iterador al primer hijo del nodo actual.
- **up():** Debe mover el iterador al padre del nodo actual.
- **getCompactTrie():** Debe retornar un objeto del tipo Trie Interface, el que corresponde al Trie que este iterador está iterando.
- **getDepth():** Debe retornar el nivel en el que está el iterador. La raíz es considerada como el nivel -1 y de ahí en adelante se le va sumando uno a medida que se hacen operaciones **open**.
- **getTuple():** Debe retornar el índice de la tupla a la que este iterador está asociado.

4.4.2. Compact Trie Iterator WM

Este es el iterador que corresponde ocupar cuando se construye el índice a partir de tries de la clase Compact Trie WM.

4.4.2.1. Métodos Importantes

- **Constructor:** Recibe un puntero a un Compact Trie WM y el índice de la tupla para la que fue creado, entraremos más en detalle sobre esto en la sección de resolución de consultas. Inicializa la variable *it* que almacena la posición del iterador en 2, esto se debe a que al momento de la instanciación el iterador debe estar posicionado en la raíz y la descripción del nodo raíz comienza en la posición 2 del bitvector del Compact Trie, ya que los primeros dos valores del bitvector no identifican a ningún nodo. Además inicializa las variables *at_root* como *true*, *at_end* como *false* y *depth* como -1.
- **key():** Si las variables *at_root* o *at_end* son *true* lanza una excepción ya que no se puede obtener la llave. De lo contrario, si es que la llave ya ha sido calculada por otra operación retorna el valor precalculado, de lo contrario llama al método **keyAt** de su Compact Trie WM.

- **atEnd():** Retorna la variable *at_end*.
- **open():** Si es que estamos en la raíz, le asigna a la variable *at_root* el valor *false* debido a que bajaremos un nivel en le trie y por lo tanto ya no se estará en la raíz. Verifica que el nodo actual tenga hijos y de tenerlos le asigna a la variable *parent_it*, que almacena la posición del padre del nodo actual, el valor encontrado en *it*, e *it* pasa a ser el valor encontrado al llamar al método **child(it,1)** del Compact Trie WM ya que queremos bajar al primer hijo de este nodo. *depth* es incrementado en uno debido a que se bajó un nivel.
- **next():** Utilizando el método **childrenCount** del CompactTrie WM obtiene la cantidad de hijos del padre del nodo actual. Si la posición actual del nodo es igual a la cantidad de hijos quiere decir que no hay más hijos por lo que debemos asignarle a *at_end* el valor *true*, de lo contrario se mueve el iterador al siguiente hijo del padre del nodo actual.
- **up():** Decrementa *depth* en uno ya que se subirá un nivel en el trie. *it* pasa a tener el valor de su padre y *at_end* pasa a ser *false*. Si es que *it* tiene un valor igual a 2, *at_root* pasa a ser *true* ya que ésa es la posición de la raiz.
- **seek(seekKey):** Recibe un entero *seekKey*. Obtiene la cantidad de hijos del padre del nodo actual y luego calcula los valores i y f, con i correspondiendo a la posición en la secuencia de llaves de la llave del nodo actual y f la posición en la secuencia de llaves de la llave del último hijo del padre del nodo actual. Luego llama al método **range_next_value_pos(seekKey, i,f)** de la wavelet matrix para encontrar la posición y el valor del primer valor mayor o igual a seekKey entre i y f inclusive en S. Si la posición encontrada es $f+1$ quiere decir que dicho valor no existe, de lo contrario se actualiza el valor del *it* en base a la posición encontrada y se guarda el valor encontrado para no volver a calcular la llave en la que está el iterador.
- **getCompactTrie():** Retorna el Compact Trie WM asociado a este iterador.
- **getDepth():** Retorna la variable *depth*.
- **getTuple():** Retorna la tupla asociada a este iterador.

4.4.3. Compact Trie Iterator IV

Este es el iterador que corresponde ocupar cuando se construya el índice a partir de tries de la clase Compact Trie IV.

4.4.3.1. Métodos Importantes

Debido a que los Compact Trie WM y Compact Trie IV comparten la manera de manejar la estructura del árbol, los iteradores también comparten varios métodos, por lo que no ahondaremos en aquellos métodos que son equivalentes a los ya presentados en la clase Compact Trie Iterator WM.

- Todos los métodos excepto seek son equivalentes en ambas clases.
- **seek(seekKey):** Recibe un entero *seekKey*. Al igual que en Compact Trie Iterator WM se calculan los valores i y f correspondiente al rango de llaves que deben ser consultados. Luego se llama al método **binary_search_seek(seekKey,i,f)** del Compact Trie IV asociado a este iterador para encontrar el valor y la posición de la llave buscada. Si la posición encontrada es

$f+1$ quiere decir que no existe dicha llave, ni otra mayor a esta, en la sección del árbol que estamos buscando y por lo tanto *at_end* pasa a ser *true*, de lo contrario se actualiza la posición del iterador y se guarda la llave encontrada.

4.5. Resolución de Consultas

Para resolver una consulta se requieren tres cosas: la ubicación del índice, la consulta y el vector GAO asociado a la consulta. En la práctica se recibe un archivo con varias consultas en el que la primera línea corresponde a la primera consulta y la segunda al vector GAO de dicha consulta y luego se repite este esquema para el resto de las consultas.

Las consultas corresponden a strings que tienen tuplas de elementos, cuyo tamaño corresponde al de la dimensión de la tabla indexada. Los valores de la tupla pueden ser variables o constantes. Cada tupla es seguida de un punto que identifica que se debe hacer join entre la tuplas involucradas.

El vector GAO nos indica el orden en el que se deben buscar los valores de cada una de las variables de la consulta. Cabe destacar que la construcción del vector GAO está fuera del alcance de la memoria, pues corresponde a un problema de optimización de consultas.

Al recibir una consulta, se reconstruirá el índice en memoria principal y se creará una instancia de la clase LTJ, para luego utilizar su método **triejoin** para resolver la consulta. El índice es construido una sola vez antes de leer las consultas y es utilizado por cada instanciación de la clase LTJ.

Se detallará a continuación algunas clases importantes para la resolución de consultas para luego pasar a explicar el funcionamiento de LTJ que es la clase principal.

4.5.1. Term

Esta clase es utilizada para representar todos los términos encontrados en la consulta. Los términos pueden ser de dos tipos: Variables o Constantes. Cuenta con los métodos:

- **isVariable():** Retorna True si es que el término corresponde a una variable, False si no.
- **getVariable():** Retorna el string correspondiente a la variable que representa.
- **getConstant():** Retorna el entero correspondiente a la constante que representa.

4.5.2. Tuple

Esta es la clase utilizada para representar a cada tupla del join. Su principal elemento es un vector de Terms a los cuales se puede acceder a través del método getTerm(i) que recibe un entero y retorna el término del índice i.

Para una consulta:

$$?x_1\ 1\ ?x_2 \cdot ?x_2\ ?x_3\ ?x_4$$

Se crearán objetos Term para los valores $?x_1$, 1, $?x_2$, $?x_3$, $?x_4$ y dos objetos de la clase Tuple, uno que recibe los términos $?x_1, 1, ?x_2$ y otro que recibe los términos $?x_2, ?x_3, ?x_4$.

4.5.3. Leapfrog Join

Esta clase es instanciada con un vector de iteradores, una variable y una dimensión. Será utilizada para encontrar los resultados para la variable que se le es entregada al inicializarla. Los iteradores que se le entregan son los asociados a esta variable, por lo que estará a cargo de encontrar los resultados y actualizar el nivel en el que se está en cada iterador a medida que sea necesario.

4.5.3.1. Métodos Importantes

- **leapfrog-init():** Si alguno de los iteradores está *al final* actualiza su variable *at_end* a *true*, esto nos indica que no se pueden encontrar más resultados en la zona del árbol en la que estamos. Si ningún iterador está *al final* los ordena de acuerdo a la llave en la que están posicionados.

- **leapfrog-search():** Encuentra un resultados del join para la variable, si es que existe alguno. Es decir encuentra los valores para la variable correspondiente al Leapfrog Join tal que se se cumplan las condiciones de la consulta para los valores evaluados hasta este punto. Esto se lleva a cabo moviendo la posición de los iteradores hasta que todos estén posicionados en la misma llave, en ese momento sabemos que hemos encontrado un binding para la variable y este es almacenado en la variable *key* del Leapfrog Join. Si no es posible encontrar un binding actualiza el *at_end* como *true*.

- **leapfrog-next():** Mueve todos los iteradores a la siguiente instancia de la variable con la que se está trabajando. Llama a **leapfrog-search** para encontrar el siguiente resultado.

- **leapfrog-seek(seekKey):** Recibe un entero *seekKey* y encuentra el primer elemento que es mayor o igual a *seekKey*, si es que éste existe, utilizando el método **seek** de los iteradores.

- **up():** Sube todos los iteradores un nivel utilizando el método **up** de éstos.

- **up(up_indicator):** Recibe un vector de booleanos up_indicator, el cual tiene el tamaño de los iteradores que conformar al Leapfrog Join. Para cada iterador si el índice correspondiente en el up_indicator es *true* sube de nivel, de lo contrario re-posiciona el iterador en el primer hijo del padre del nodo actual.

- **open():** Baja un nivel en todos los iteradores, utilizando el método **open** de estos.

A pesar de que los métodos **up** y **open** están fuera del scope de Leapfrog Join descrito en [2], a la hora de la implementación resultó útil que esta clase pudiera mover los iteradores de manera vertical, ya que en general se desean mover todos los iteradores asociados a una variable al mismo tiempo.

4.5.4. Clase LTJ

Esta clase es la que se utiliza para responder las consultas.

4.5.4.1. Construcción

Para instanciarla se le debe entregar:

- **Un puntero al índice**
- **La consulta:** Esta es representada como un vector de tuplas de la clase Tuple.
- **Vector GAO:** Corresponde a un vector de strings que indica el orden de resolución de la consulta en término de sus variables.
- **Mapeo de Variables a Tuplas:** Se le entrega un mapeo de qué variables se encuentran en qué tuplas.

Para poder responder la consulta en el orden descrito por el vector GAO, es necesario re-ordenarla. El proceso para re-ordenar la consulta consiste en modificar el orden de los términos de cada tupla para que sigan la prioridad descrita por el vector GAO, las constantes tendrán prioridad superior a cualquier variable, de manera que siempre estén al inicio de la consulta, esto se hace para descartar de inmediato las entradas que no poseen las constantes requeridas por la consulta. Para cada tupla de la consulta se hace lo siguiente:

- Se obtienen los índices de las constantes en la tupla y estos son agregados a un string que describe el orden.
- Se obtiene los índices de las variables en la tupla. Estas se van agregando al string que describe el orden, de acuerdo a la prioridad descrita por el vector GAO.
- En este momento el string que describe el orden posee el nuevo ordenamiento de la tupla.

A continuación se ejemplifica cómo se hace la re-ordenación de la consulta.

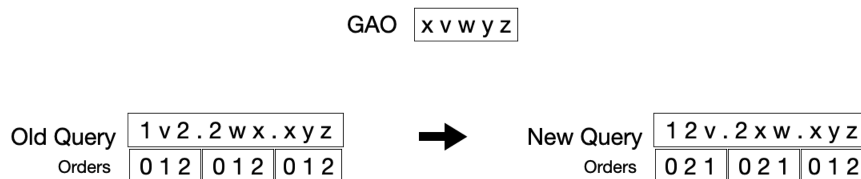


Figura 4.3: Ejemplificación del re-ordenamiento por el que pasan las consultas

Luego por cada string de orden, se creará un iterador, el cual iterará el CTrie del índice que posee el orden indicado. Por lo tanto, se posee un vector de iteradores, teniendo un iterador por tupla de la consulta. Además se almacena un mapping de en qué tuplas está cada variable, esto se utiliza para saber a qué iteradores hay que acudir cuando se quiere obtener un binding para una variable.

Luego se crea una instancia de la clase Leapfrog Join para cada variable, el cual almacenará punteros a los iteradores a los cuales está asociada la variable.

4.5.4.2. Triejoin

Este es el método que resuelve las consultas. En un inicio todos los iteradores se encuentran el la raíz de sus respectivos árboles, el método inicia llamando a triejoin-open(), que baja un nivel en todos los iteradores, de esa manera nos posicionamos para buscar el primer valor (ya sea variable o constante) de cada tupla.

Luego se procede a avanzar en cada tupla por sus constantes. Como fue descrito anteriormente, las constantes fueron movidas al principio de la tupla, de esta manera podemos avanzar en el trie simbólico que los iteradores recorren. Este proceso se lleva a cabo de la siguiente manera para cada tupla y consecuentemente para el iterador que está asociado a ella.

- Se iterará por todos los términos de la tupla
- Para cada término se verifica si es constante o no.
- Si es constante se utiliza la operación **seek** del iterador, a la cual se le entrega el valor de la constante buscada. Esto moverá el iterador hasta que llegue a un valor igual o mayor a la constante.
- Si el valor encontrado es mayor a la constante buscada o el iterador está *al final* esto quiere decir que no existe ningún camino del árbol que resuelva la tupla actual, por lo que se retorna ya que el join no tiene resultado.
- Si el valor encontrado es igual a la constante buscada entonces hacemos open del iterador asociado a la tupla actual y continuamos con el siguiente término de la tupla.

Finalmente se pasa a buscar los bindings de las variables para los cuales se cumple la consulta. Esto se hace mediante el siguiente proceso:

- Inicialmente se define la variable *gao_index* que corresponderá a los índices del vector GAO e indicará la variable con la que se está trabajando en cada momento. El valor inicial de *gao_index* será 0, es decir se parte buscando los bindings para la variable de mayor prioridad.
- Mientras *gao_index* no llegue al final del vector GAO o se llegue a alguno de los casos que indique que no hay más resultados, este proceso se repite.
- Se define la variable *lj* como el Leapfrog Join asociado a la variable en *gao_index*.
- Se llama al método **leapfrog_search** de *lj* para encontrar un binding válido para la variable actual.
- Si después de llamar a **leapfrog_search** *lj* se encuentra *al final*, quiere decir que en la sección de los iteradores en la que se está no hay un binding que cumpla la consulta para la variable actual, por lo que debemos subir por todos los iteradores. Se sube variable por variable, decrementando el valor del *gao_index* hasta encontrar una variable en la que al utilizar el método **leapfrog_next** sus iteradores no terminen *al final*. Si se llega a la variable con *gao_index* igual a 0 y al hacer **leapfrog_next** sus iteradores están *al final* quiere decir que ya no hay más respuestas y por lo tanto debe terminar el programa.

- Si después de llamar a **leapfrog_search** *lj* no se encuentra *al final* quiere decir que se ha encontrado un binding válido para la variable. Si se está en la variable con mayor *gao_index* se llama a **leapfrog_next** de *lj* para poder encontrar nuevos bindings de esta misma variable, de lo contrario se llama a **open** de *lj* para bajar un nivel en todos los iteradores que están asociados a la variable actual y se incrementa *gao_index* en uno para trabajar con la siguiente variable. Se repite el proceso anteriormente descrito para este *gao_index*.

Se puede encontrar un ejemplo del funcionamiento de esta implementación en el Anexo A. Junto con esto el código implementado se encuentra en https://github.com/DaniCampos/LTJ-CompactTries.

Capítulo 5

Validación

En esta sección se detallarán los experimentos que se llevaron a cabo para ver el rendimiento de los índices creados y compararlos con otros índices que resuelven consultas del mismo tipo.

5.1. Datos

Los datos para construir los índices fueron obtenidos de Wikidata [10], la cual tiene dimensión tres. Se utilizó un sub-grafo de esta que contiene $n = 81,426,573$ triples, determinado por una tabla con las columnas sujeto(s), predicado(p) y objeto(o). En los datos había $19,227,372$ sujetos, $2,101$ predicados y $39,894,042$ objetos diferentes [6]. El subgrafo almacenado en disco, en formato de texto, utiliza 1.7 GB y al pasarlo a memoria utiliza 932 MB almacenando los valores en enteros de 32 bits.

5.2. Consultas

Se utilizaron las consultas provistas en Wikidata Graph Pattern Benchmark (WGPB) por Hogan et al. [9], las cuales están hechas pensando en el sub-grafo de la Wikidata utilizado. WGPB provee 17 tipos de patrones de consultas que representan diferentes tipos de conexiones entre los elementos del grafo. Los patrones son los representados en la Figura 5.1.

Para cada patrón hay 50 consultas no vacías.

Para cada una de estas consultas se requería un vector GAO y estos fueron obtenidos del cálculo realizado por Ring, es decir esta implementación utiliza los mismos vectores GAO que los utilizados por Ring al responder estas mismas consultas.

5.3. Experimentos

Los experimentos se corrieron en una máquina Intel(R) Xeon(R) CPU E5-2630 en 2.30GHz, con 6 cores, 15 MB de cache, y 96 GB de RAM. Todas las consultas se corrieron con un límite de 1000 resultados.

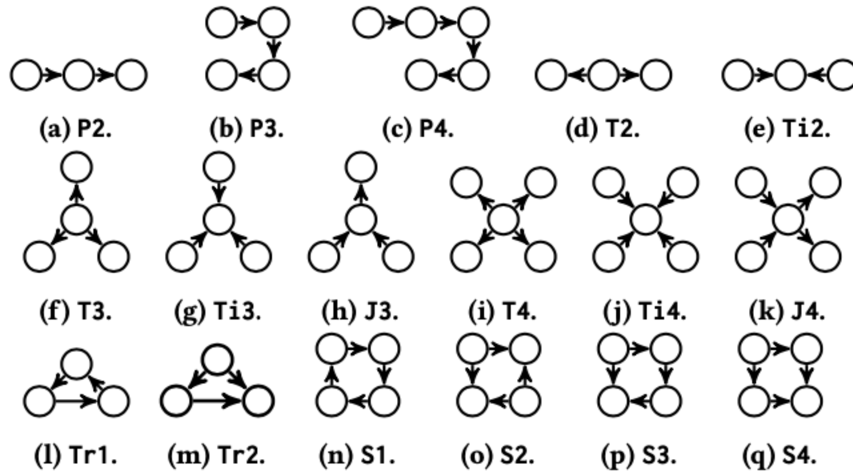


Figura 5.1: Patrones de consultas de WGPB [6]

Se comparó a los índices creados con los siguientes algoritmos de join Worst Case Optimal:

- **Ring y C-Ring:** Índice Ring descrito en Estado del Arte y su versión ocupando bitvectors compactos. Ambos índices corren en memoria principal [6].
- **EmptyHeaded:** Otra variante del algoritmo Leapfrog Triejoin. Los triples son guardados en 6 tries diferentes en memoria principal [11].
- **Qdag:** Índice WCO compacto basado en una representación con quadtrees de grafos que corre en memoria principal [12].

Además se lo comparó con otros sistemas de bases de datos para tener un punto de referencia de su funcionamiento. Esos fueron:

- **Jena LTJ:** Descrito en Estado del Arte Sección 2.3.
- **RDF-3X:** Indexa una tabla de triples en un árbol B+ compacto clusterizado. Se ordena a los triples de manera que las hojas del árbol puedan ser codificadas de manera diferencial. Utiliza un optimizador de consultas basado en pairwise joins [13].
- **Virtuoso:** Es una base de datos de grafos. Provee un índice por columnas de quads con un atributo de grafo (g) adicional. Optimizado para patrones que tienen predicados constantes [14].
- **Blazegraph:** Sistema de base de datos para grafos, alberga el Wikidata Query Service [15].

Los datos del tiempo de consulta para los índices y sistemas de bases de datos anteriormente mencionados fueron obtenidos en el testeo de Ring [6]. No fue necesario correr nuevamente los experimentos debido a que se está ocupando la misma máquina.

5.3.1. Indexación

Se construyó un índice por cada variante implementada. Se llamará LTJ IV y LTJ WM a los índices construidos a partir de los iteradores Compact Trie Iterator IV y Compact Trie Iterator WM respectivamente. LTJ IV fue construido en 2 horas, 49 minutos 4 segundos (10144 segundos) y LTJ WM fue construido en 4 horas, 32 minutos y 52 segundos (16372 segundos). Ambos índices fueron construidos utilizando todos los órdenes posibles para las columnas sujeto(s), predicado(p), objeto(o), es decir *spo*, *sop*, *pos*, *pso*, *osp*, *ops*. El espacio utilizado por los índices se puede ver en la Tabla 5.1, donde la diferencia entre el espacio utilizado por ambos índices, en memoria principal versus en disco, puede ser atribuido a los vectores necesarios para poder realizar operaciones de *rank* y *select* sobre los bitvectors utilizados por ambos índices, los cuales no son almacenados en disco, si no que se instancian junto con los índices

Tabla 5.1: Espacio utilizado por Índices implementados

Espacio utilizado por	LTJ IV(GB)	LTJ WM(GB)
Índice en memoria	3.7	5.3
Índice en disco	3.3	4.8
Construcción del Índice	11.29	9.9

5.3.2. Resultados

Los valores encontrados en los resultados fueron comparados con los entregados por Ring y todos concuerdan, es decir se verificó la correctitud de los resultado entregados por las nuevas variantes.

En la Tabla 5.2 se puede observar la diferencia en el espacio por triple en bytes por los diferentes sistemas que fueron comparados y el tiempo promedio que a estos sistemas les toma resolver una consulta, en milisegundos. LTJ IV y LTJ WM utilizan más espacio que Ring, C-Ring y Qdag, pero menos espacio que el resto de los sistemas. Respecto al tiempo promedio que les toma resolver consultas se observa que LTJ IV resulta bastante exitoso, teniendo un tiempo de respuesta por consulta de 40 msec, que es menor a todos los índices menos Ring. Por otro lado LTJ WM tiene un tiempo promedio de consulta bastante más alto, sólo es más rápido que Qdag, Virtuoso y Blazegraph.

Tabla 5.2: Espacio utilizado en bytes por triple y tiempo promedio de consulta en milisegundo para los sistemas comparados

Sistema(Índice)	Espacio(bytes)	Tiempo(msec)
LTJ IV	45.62	40
LTJ WM	65.17	285
Ring	12.70	31
C-Ring	6.68	97
Qdag	8.86	14873
EmptyHeaded	1809.84	118
Jena LTJ	144.64	59
RDF-3X	107.65	182
Virtuoso	104.49	1135
Blazegraph	99.86	1709

En la Figura 5.2 se puede ver el tiempo de respuesta de cada índice, separados por el tipo de patrón de las consultas contestadas. En el gráfico asociado a cada patrón, el rectángulo perteneciente a cada índice refleja el rango de valores entre el percentil 25 y 75 de los tiempos de respuesta para dicho patrón, con la línea central del rectángulo representando la mediana de los tiempos de consulta y los límites de las líneas correspondiendo a los tiempos máximos y mínimos.

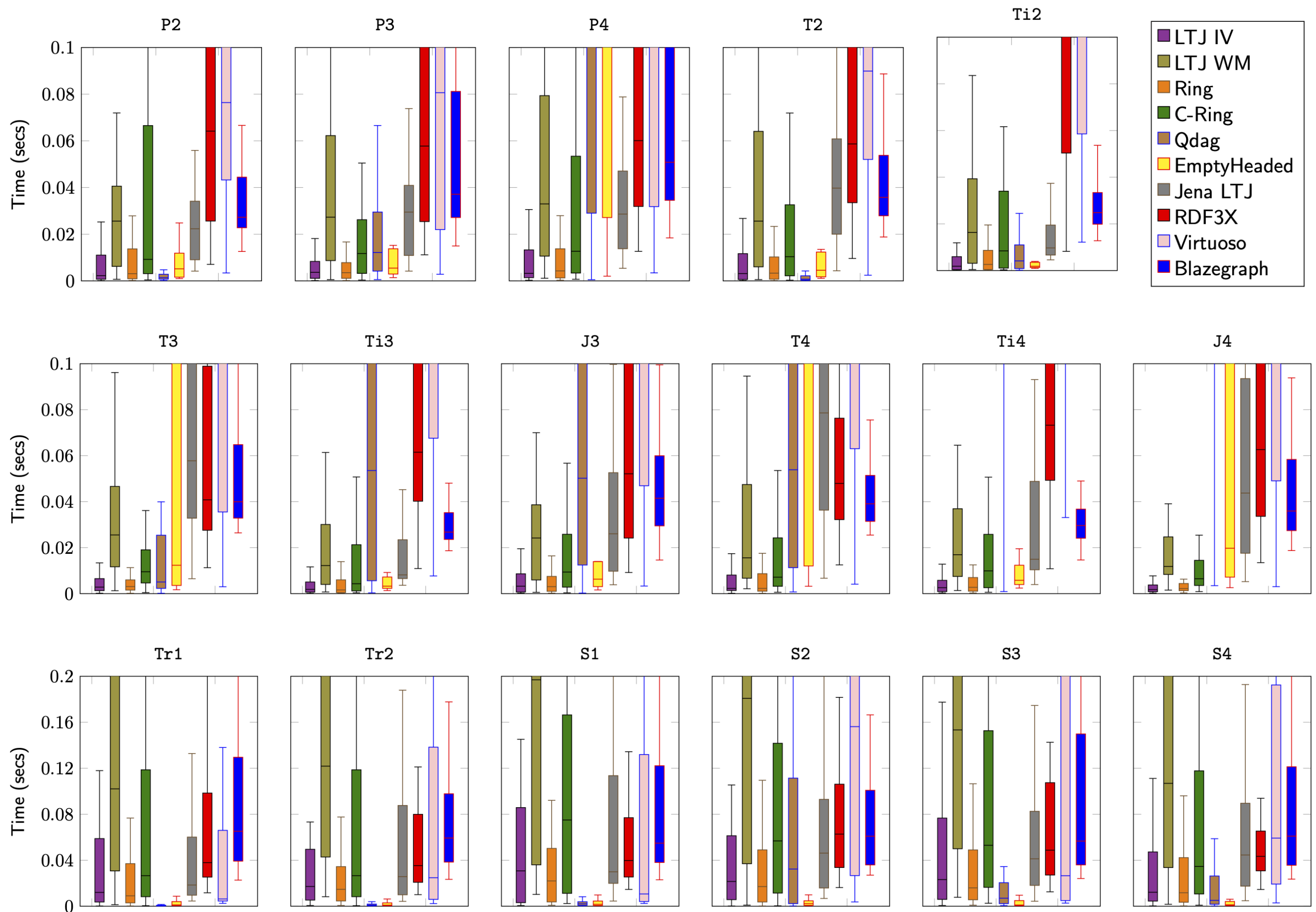


Figura 5.2: Comparación del tiempo de resolución de consultas(en segundos). Las cajas asociadas a cada índice van desde el 25 % al 75 % de los datos, con la mediana marcada dentro de la caja. Las líneas se extienden desde el mínimo valor al máximo, sin considerar valores atípicos.

Al considerar la mediana del tiempo de resolución de consulta de los índices, se observa que LTJ IV obtiene resultados mejores que el resto de los índices para el patrón P4, se comporta de manera similar a Ring en los patrones P3, T2, T3, Ti3, J3, Ti4, J4, Tr1 y S4 y es más lento que Qdags o EmptyHeaded en los patrones P2, T2, Ti2, Tr1, Tr2, S1, S2, S3 y S4. Es más rápido que los índices Jena LTJ, RDF3X, Virtuoso y Blazegraph en todo tipo de patrones. LTJ WM no tiene tan buenos resultados, para la mayoría de las consultas sólo es más rápido que RDF3X, Virtuoso y Blazegraph y para los patrones Tr1, Tr2, S1, S2, S3 y S4 es el índice con peor mediana.

Al evaluar los tiempos de consulta en el peor caso, se observa que LTJ IV es bastante consistente, ningún patrón lleva a tiempos que se salgan de la norma respecto a los otros índices y en general para todos los patrones tiene tiempos competitivos. Por otro lado LTJ WM posee tiempos de consultas bastante altos en el peor de los casos escapándose de los rangos normales en al menos la mitad de los patrones.

5.3.3. Testeo en mayores dimensiones

Para probar la correctitud de la implementación en mayores dimensiones se la testeó con ejemplos que no utilizaban datos reales. Para estos ejemplos se indexaron correctamente los datos y las respuestas a las consultas fueron las esperadas.

Capítulo 6

Conclusiones

El trabajo llevado a cabo en esta memoria consistió inicialmente en proveer un índice que almacenara datos de manera eficiente y pudiera ser utilizado para resolver consultas de join por el algoritmo Leapfrog Triejoin. Posteriormente se implementó otra variante del índice y además se implementó una versión de Leapfrog Triejoin que pudiera ser ocupada por cualquier índice implementado en el futuro.

Se lograron los objetivos planteados y se entregó una implementación de un algoritmo que resulta fácil de usar y podrá ser de gran ayuda a futuros trabajos. Además esta implementación es capaz de indexar tablas de cualquier dimensión, por lo que no está limitada a bases de datos de grafos. En el caso de bases de datos de grafos, la representación compacta de las estructuras del algoritmo Leapfrog Triejoin permitió reducir el espacio utilizado por este a un tercio del original y disminuir el tiempo de consulta en un 30 %.

Este trabajo expone cómo las estructuras compactas pueden ser utilizadas para mejorar el rendimiento de algoritmos y que no siempre la estructura más compleja tendrá los mejores resultados. De las dos versiones implementadas en esta memoria, la más simple LTJ IV, fue la que obtuvo mejores resultados. Es importante mencionar que Ring, que fue uno de los algoritmos que se utilizó para comparar, utiliza estructuras más sofisticadas pero no pierde eficiencia comparada con implementaciones compactas más simples.

El trabajo fue desafiante en varias aristas, en un comienzo la comprensión del algoritmo resultó difícil y posteriormente en la implementación del sistema, hubo varios obstáculos debido a su complejidad y magnitud.

Bibliografía

[1] H. Q. Ngo, C. Ré, and A. Rudra, "Skew strikes back: new developments in the theory of join algorithms," *SIGMOD Rec.*, vol. 42, pp. 5–16, 2013.

[2] T. L. Veldhuizen, "Leapfrog triejoin: a worst-case optimal join algorithm," *CoRR*, vol. abs/1210.0481, 2012.

[3] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, "Worst-case Optimal Join Algorithms," *J. ACM*, vol. 65, no. 3, pp. 16:1–16:40, 2018.

[4] G. Navarro, *Compact Data Structures – A practical approach*. Cambridge University Press, 2016. ISBN 978-1-107-15238-0. 536 pages.

[5] J. G. Raghu Ramakrishnan, *Database Management Systems*. McGraw-Hill Companies, 2nd ed., 2000.

[6] D. Arroyuelo, A. Hogan, G. Navarro, J. L. Reutter, J. Rojas-Ledesma, and A. Soto, "Worst-case optimal graph joins in almost no space," in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021* (G. Li, Z. Li, S. Idreos, and D. Srivastava, eds.), pp. 102–114, ACM, 2021.

[7] A. Atserias, M. Grohe, and D. Marx, "Size Bounds and Query Plans for Relational Joins," *SIAM J. Comput.*, vol. 42, no. 4, pp. 1737–1767, 2013.

[8] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. Computer Science and Information Processing, Reading, Massachusetts: Addison-Wesley, 1st ed., Jan. 1983.

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

[10] D. Vrandeić and M. Krötzsch, "Wikidata: a free collaborative knowledgebase," *Commun. ACM*, vol. 57, pp. 78–85, 2014.

[11] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré, "Emptyheaded: A relational engine for graph processing," *ACM Trans. Database Syst.*, vol. 42, oct 2017.

[12] G. Navarro, J. L. Reutter, and J. Rojas-Ledesma, "Optimal joins using compact data structures," 2019.

[13] T. Neumann and G. Weikum, "The RDF-3X engine for scalable management of RDF data," *VLDB J.*, vol. 19, no. 1, pp. 91–113, 2010.

[14] O.ErlingandI.Mikhailov, *RDF support in the Virtuoso DBMS.In Networked Knowledge – Net-*

worked Media. Springer, 2009.

[15] B. B. Thompson, M. Personick, and M. Cutcher, “The bigdata® rdf graph database,” 2014.

Anexo A

A.1. Ejemplo indexación y resolución de consultas

A continuación se explicará en detalle un ejemplo, para seguir los pasos que la implementación entregada utiliza para indexar y resolver consultas.

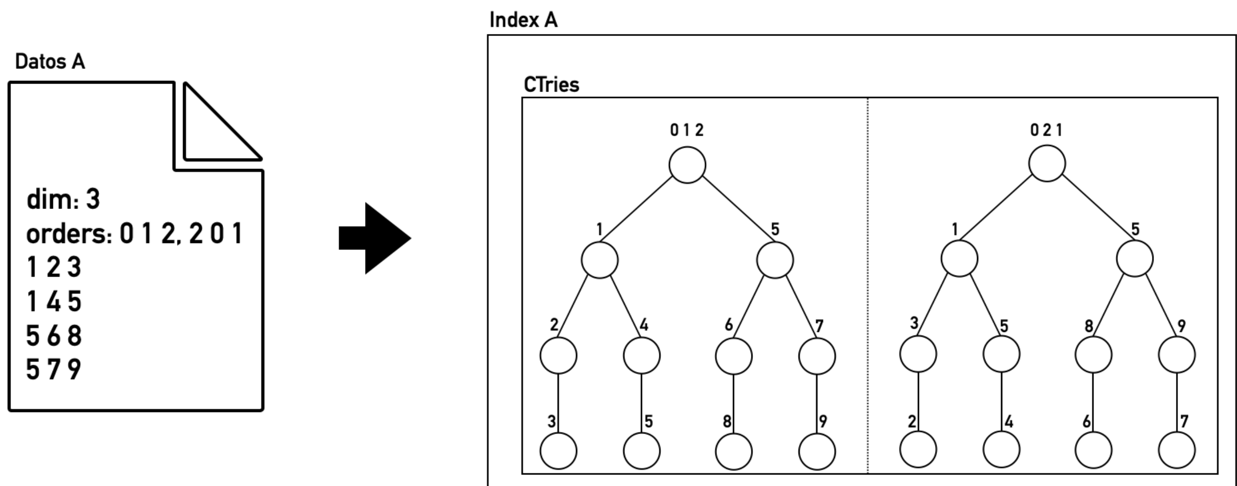


Figura A.1: Índice A construido a partir de los Datos A

En la Figura A.1 se puede ver la conversión de un archivo de datos, con la estructura requerida, a un objeto de la clase Index. Se crea un CTrie (versión compacta de el Trie representado en la figura) por cada orden solicitado en los datos. Posteriormente este índice es almacenado en disco.

En la Figura A.2 se muestra el primer paso para resolver una consulta, la instanciación de un objeto LTJ. Para poder instanciarlo se le entrega una consulta (Query), un vector GAO, el índice A, el cual es cargado desde el disco y un mapeo de que variables aparecen en que tuplas (Mapping). Tras la instanciación LTJ contará con los siguientes elementos:

- **Modified Query:** Una versión modificada de la consulta original (Query). En este caso la constante que hay ya está al principio de su tupla, por lo que no es necesario moverla. Las variables son ordenadas dentro de cada tupla de acuerdo a la prioridad indicada en el vector GAO. En la primera tuple se intercambian los valores de $?V1$ y $?V2$, debido a que $?V2$ está

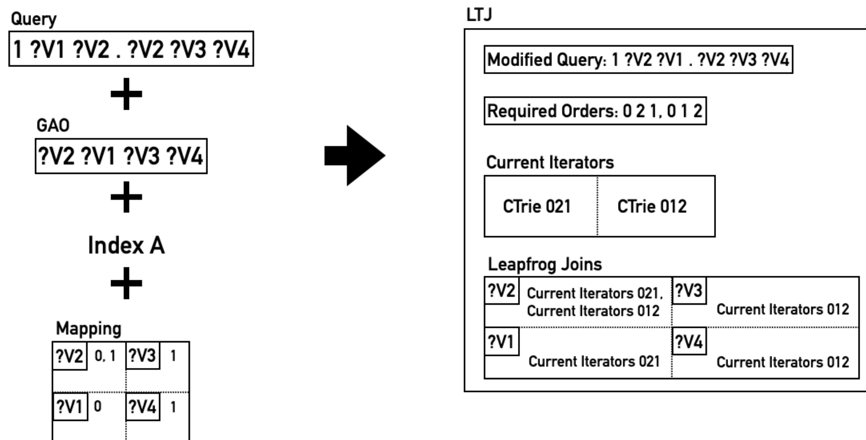


Figura A.2: Resolución de Consultas: Instanciación de LTJ

antes en el vector GAO y en la segunda tupla se mantiene el orden original ya que este sigue el indicado por el vector GAO.

- **Required Orders:** En base a las modificaciones hechas a la consulta se obtienen los órdenes asociados a cada variable. Para la primera tupla se tiene el orden 021, lo que indica que los CTries que se ocuparán tendrán en su primer nivel los datos de la columna 0 de los datos originales, en el segundo la columna 2 y en el tercero los de la columna 1. Para la segunda tupla se tiene el orden 012, lo que indica que se ocupará el mismo orden de la tabla original.

- **Current Iterators:** Contiene un iterador por cada tupla y dicho iterador utiliza el CTrie asociado al orden correspondiente en Required Orders.

- **Leapfrog Joins**: Contiene un objeto de la clase Leapfrog Join por variable de la consulta y cada uno de estos recibe los iteradores asociados a las tuplas que contienen dicha variable. En este caso sólo la variable $?V2$ se encuentra en ambas tuplas, por lo que su Leapfrog Join recibe ambos Current Iterators. El Leapfrog Join de la variable $?V1$ recibe el Current Iterator asociado al orden 021 y los de las variables $?V3$ y $?V4$ el Current Iterator asociado al orden 012.

De aquí en adelante, en los rectángulos punteados se verá la representación conceptual del movimiento de los iteradores sobre los CTries. En la práctica el movimiento se hace por las estructuras compactas que representan a estos CTries.

Tras haber instanciado el LTJ se pasa a llamar al método **triejoin**. En este punto de la ejecución, los iteradores se encuentran en las raíces de sus respectivos CTries. En la Figura A.3 podemos ver en naranjo los nodos en los cuales los iteradores están posicionados.

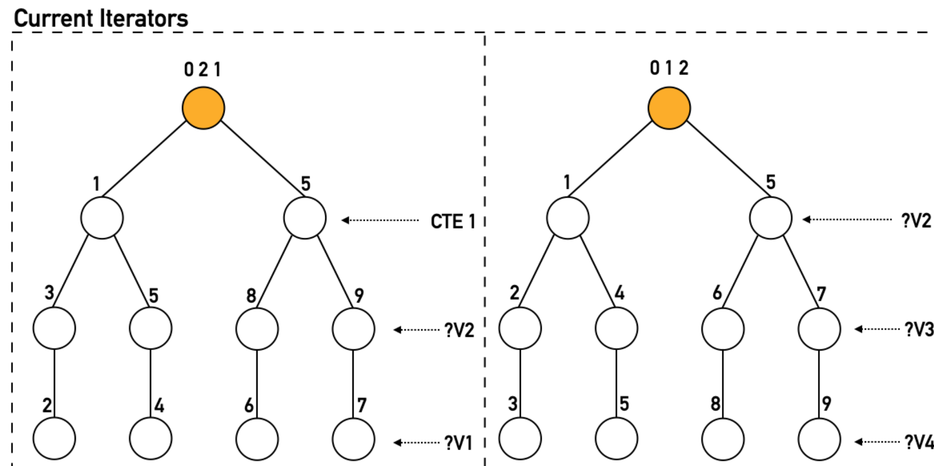


Figura A.3: Estado inicial de los iteradores al llamar al método **triejoin**

El primer paso del método **triejoin** consiste en aplicar el método **triejoin_open** el cual hace que todos los iteradores bajen un nivel. En la Figura A.4 se observa en color rojo los nodos en los cuales quedan posicionados los iteradores después de este paso.

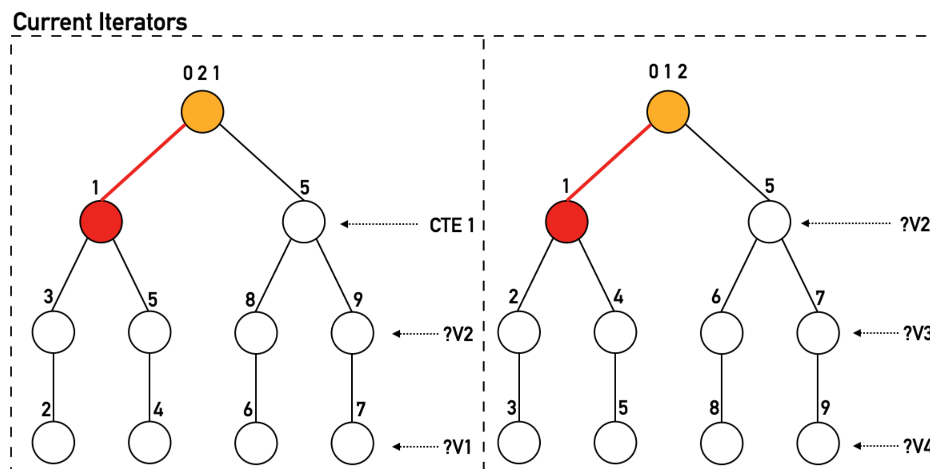


Figura A.4: Se aplica **triejoin_open** sobre todos los iteradores

Ahora se procede a avanzar por las constantes de cada tupla de la consulta (Figura A.5). Sólo la primera tupla, la cual está asociada al iterador con el orden 021, posee una constante, por lo que se bajará un nivel el iterador desde el nodo etiquetado con un 1, el cual corresponde a la constante que buscamos.

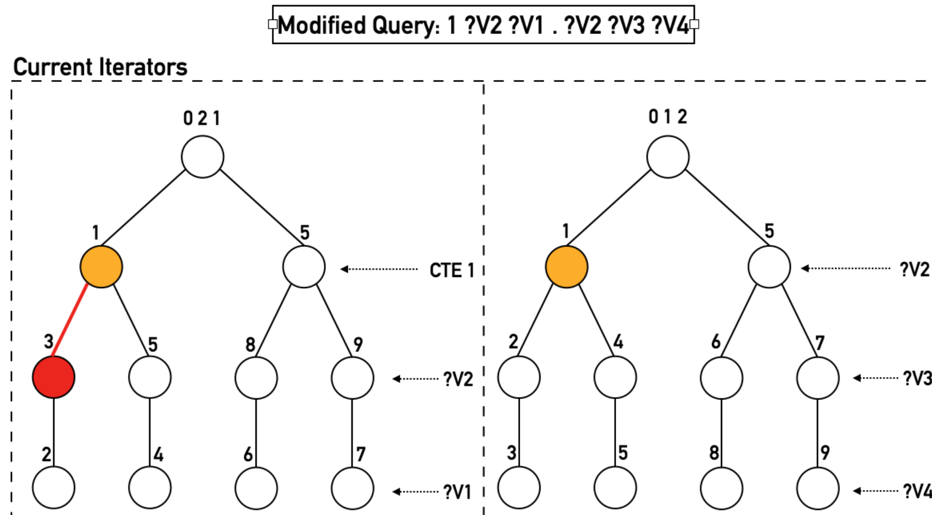


Figura A.5: Avance por constantes

Se procede a buscar los bindings para las variables de la consulta. Esto se hace definiendo a *gao_index* como 0, el cual identifica el índice en el vector GAO de la variable para la cual se está buscando el valor. En este caso la variable es $?V2$ y su Leapfrog Join posee a ambos iteradores 021 y 012, ya que esta variable está en ambas tuplas. Se llama al método **leapfrog_search** del Leapfrog Join el cual buscará un valor que esté presente en ambos iteradores para $?V2$, encontrando al valor 5, por lo que esta será la nueva posición de ambos iteradores, reflejada en los nodos rojos de la Figura A.6.

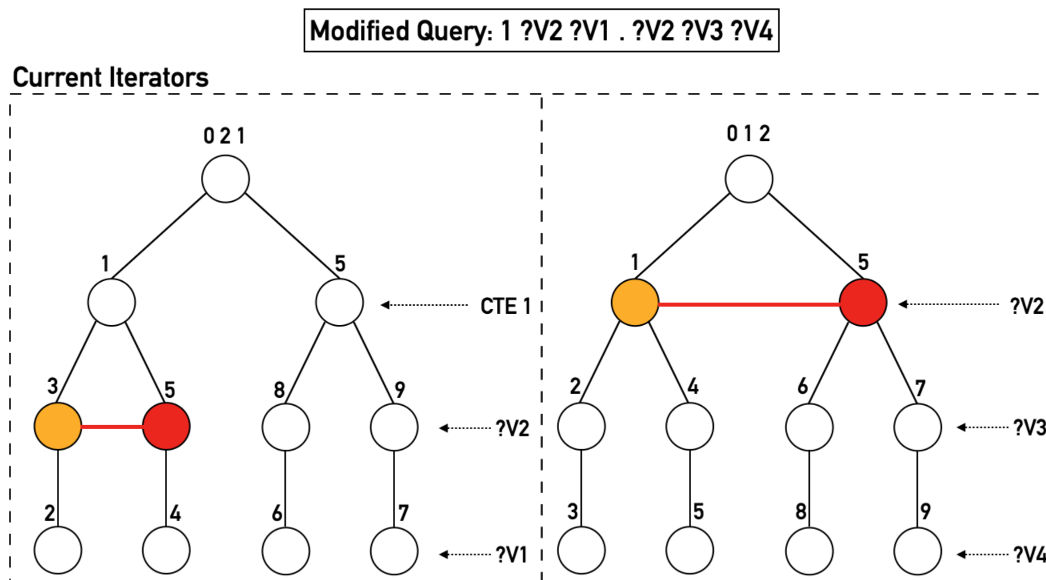


Figura A.6: Obtener valor para $?V2$

Ahora se llama a **open** en el Leapfrog Join de $?V2$ para bajar un nivel en todos sus iteradores y se incrementa *gao_index* en uno, por lo que la nueva variable para la que se busca valor es $?V1$.

Como se ve en la Figura A.7, la variable $?V1$ solo está asociada a la primera tupla de la consulta, por lo que su Leapfrog Join tiene únicamente al iterador 021 y al llamar a **leapfrog_search**, se encontrará el único valor disponible para $?V1$ que es 4.

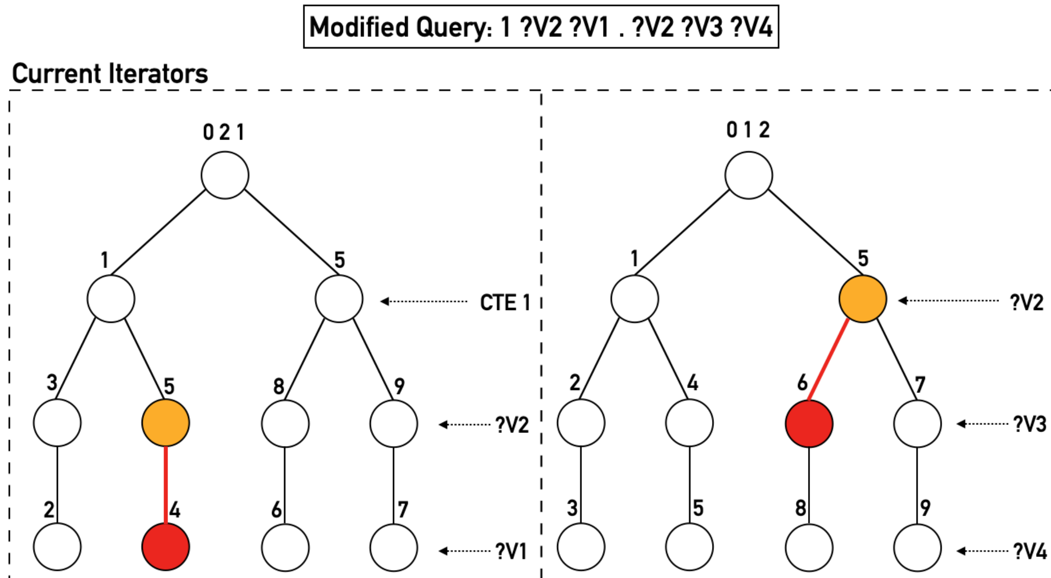


Figura A.7: Se baja un nivel en los iteradores asociados a $?V2$

En la Figura A.8 se pueden ver en rosado, los nodos correspondientes a los bindings encontrados hasta ahora para las variables $?V2$ y $?V1$. El *gao_index* es nuevamente incrementado y se pasa a la variable $?V3$, que sólo se encuentra en la segunda tupla de la consulta y por lo tanto su Leapfrog Join sólo tiene al iterador 012, al hacer **leapfrog_search**, se obtiene el primer valor disponible que corresponde a 6 y se baja un nivel para avanzar a la siguiente variable.

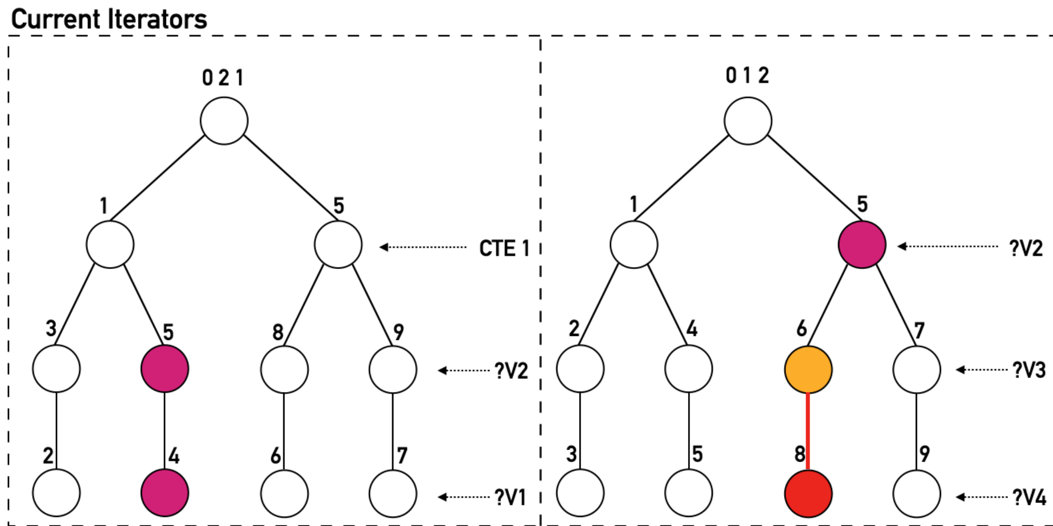


Figura A.8: Se baja un nivel en el iterador asociado a $?V3$

En este momento el **gao_index** pasa a se 3, el cual corresponde a la variable $?V4$. Su Leapfrog Join sólo tiene asociado al iterador 012 y al hacer **leapfrog_search** se encuentra el primer y único valor disponible para $?V4$, dado los bindings anteriores, que es 8.

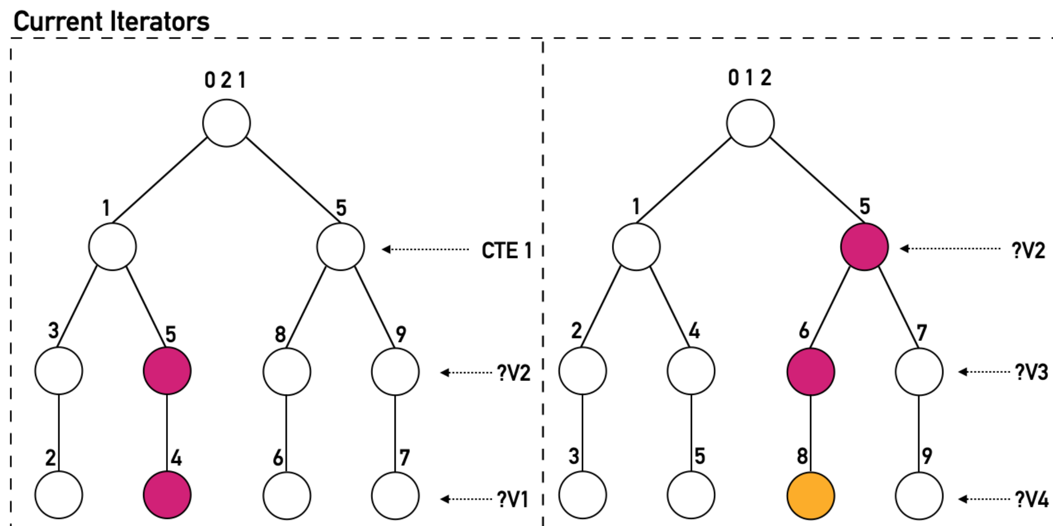


Figura A.9: Se encuentra binding para variable $V4$

Se han encontrado bindings para todas las variables $?V2$, $?V1$, $?V3$ y $?V4$, siendo estos $5,4,6,8$ respectivamente. Dado que el *gao_index* es 3, que es el máximo valor que se podría tener, indicando que ya se ha llegado al último nivel de todos los iteradores, se llama a **leapfrog_next** del Leapfrog Join de la variable $?V4$ para encontrar otras posibles soluciones, pero el nodo asociado al 8, no tiene hermanos por lo que el iterador pasa a estar *al final*, volviéndose necesario subir un nivel y decrementar el *gao_index* en uno para buscar nuevos bindings para $?V3$. La nueva posición del iterador 012 se puede ver en la Figura A.10 en color rojo.

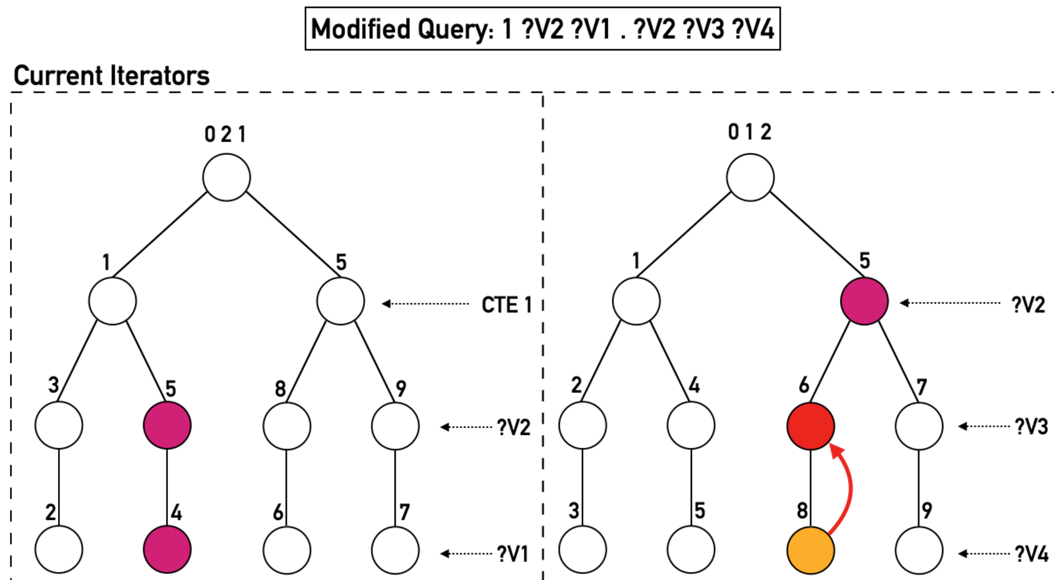


Figura A.10: Se sube un nivel en el iterador 012 para volver a buscar bindings para $?V3$

Se llama al método **leapfrog_next** del Leapfrog Join de la variable $?V3$, obteniendo el valor 7, el cual corresponde al nuevo binding de esta. La nueva posición del iterador se puede ver en rojo en la Figura A.11.

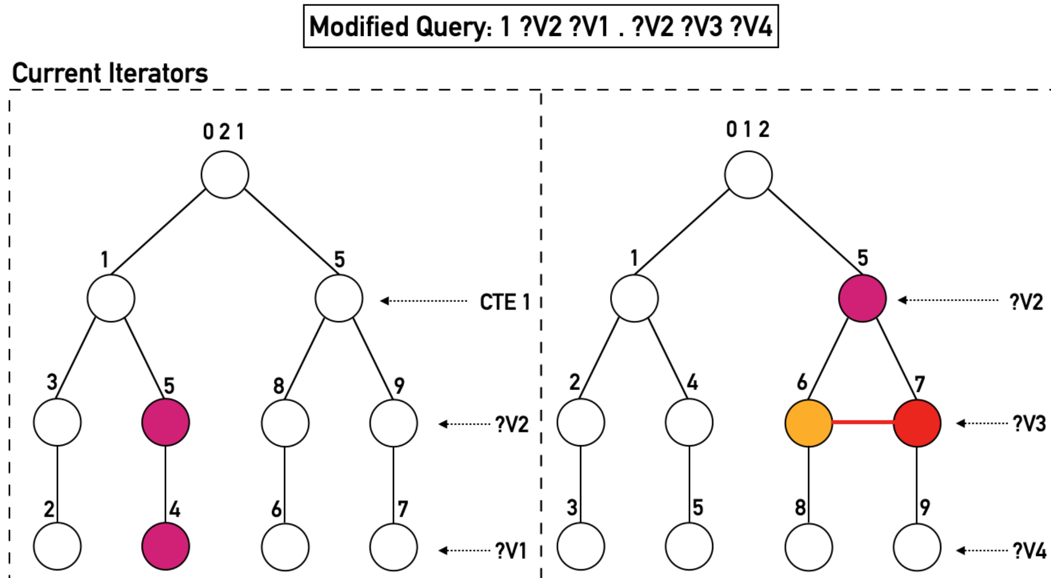


Figura A.11: Se avanza al siguiente posible valor para $?V3$

El *gao_index* es incrementado a 3 y se procede a buscar un nuevo valor para la variable $?V4$. Al hacer **leapfrog_search** en su Leapfrog Join, se encuentra al valor 9. Nuevamente se han encontrado bindings para todas las variables, esta vez los valores para $?V2$, $?V1$, $?V3$ y $?V4$ son 5,4,7,9.

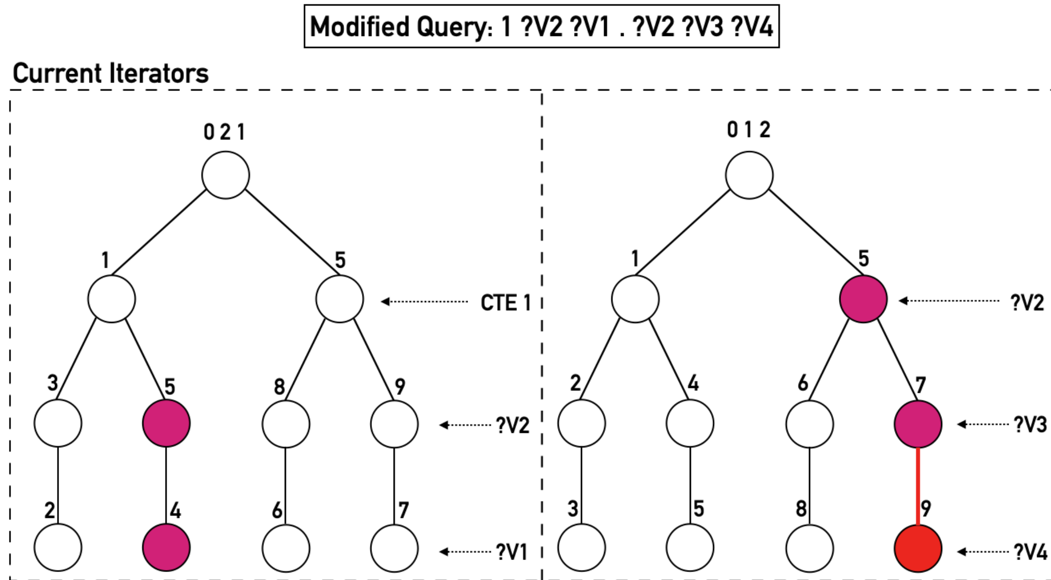


Figura A.12: Se obtiene valor para $?V4$

Se procede a hacer **leapfrog_next** para $?V4$, pero este no tiene ningún hermano a su derecha, por lo que se sube de nivel y se repite este proceso, pero ocurre lo mismo con las variable $?V3$, $?V1$ y $?V2$, esto nos indica que ya no existen más soluciones debido a que se han encontrado todos los posibles bindings para estas variables para esta consulta.