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

IMPLEMENTACIÓN DE UN LENGUAJE DE CONSULTAS PARA BASES DE DATOS DE GRAFOS UTILIZANDO ESTRUCTURAS DE DATOS COMPRIMIDAS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

NICOLÁS EMILIO LEHMANN MELÉNDEZ

PROFESOR GUÍA:
JORGE PÉREZ ROJAS

MIEMBROS DE LA COMISIÓN:
GONZALO NAVARRO BADINO
AIDAN HOGAN

SANTIAGO DE CHILE
2014

Resumen

En la actualidad existe una clara tendencia por buscar nuevos modelos de bases de datos que se adecuen de mejor manera a las necesidades modernas. Dentro de las alternativas que cuentan con popularidad se encuentran las denominadas bases de datos de grafos, que han adquirido fuerza en conjunto a la reciente revolución de la Web y sus tecnologías asociadas.

La adopción de este modelo aún requiere de un amplio estudio, pues los grafos suponen una complejidad intrínseca que debe ser considerada al momento de diseñar una implementación. Dos desafíos fundamentales que se presentan son los volúmenes de información que deben ser manejados, junto con la elección de un buen lenguaje de consultas que equilibre el nivel de expresividad con su complejidad de evaluación.

En este trabajo se considera un lenguaje de consultas ampliamente estudiado que permite realizar consultas de navegación a partir de expresiones regulares. Así mismo, se presenta un algoritmo de evaluación práctico para resolver estas consultas de manera eficiente. Adicionalmente se utiliza una representación para bases de datos de grafos que explota las características de las estructuras de datos sucintas para mantener grandes volúmenes de información en memoria principal.

El trabajo presenta además una implementación de la propuesta considerando la extensibilidad y orden del código, con el fin de proveer una herramienta de utilidad para nuevas investigaciones. El desarrollo está dividido en dos componentes. Por un lado se proporciona una biblioteca con la implementación de las estructuras sucintas involucradas y por otro un sistema simple de gestión de base de datos que permite la evaluación de consultas del lenguaje escogido.

Adicionalmente, y con el objetivo de comprobar el desempeño de la propuesta, se presenta una evaluación experimental de la implementación, realizando una comparación con algunas alternativas actuales para bases de datos de grafos y obteniendo resultados favorables. Finalmente se termina el trabajo señalando las conclusiones obtenidas del análisis experimental realizado.

A mis padres, a quiénes debo esto y mucho más.

Agradecimientos

Quiero presentar un profundo agradecimiento a mis padres quienes inculcaron en mi lo necesario para poder terminar este trabajo. Además a toda mi familia que sin entender demasiado siempre estuvieron interesados en que esto tuviera un buen término. También quiero agradecer a mis amigos que me mantuvieron sereno entre las largas jornadas de música, cerveza y programación.

Finalmente quiero agradecer al Centro de Investigación para la Web Semántica (CIWS), núcleo en el cuál se ha enmarcado mi trabajo y que me ofreció el apoyo necesario tanto en lo académico como en lo económico. En especial quiero mencionar también a mi profesor guía, quién me ayudó a encontrar un tema que se adecuara a mis gustos y siempre presentó mucho interés por mi trabajo.

Tabla de contenido

1. Introducción **1**
- 1.1. Justificación y Motivación . . . 2
- 1.2. Objetivos . . . 2
- 1.3. Resultados y Organización del Documento . . . 3

2. Antecedentes **4**
- 2.1. Bases de Datos de Grafos . . . 4
  - 2.1.1. Modelos Formales . . . 5
  - 2.1.2. Implementaciones actuales . . . 6
  - 2.1.3. Apache Jena . . . 7
- 2.2. Lenguajes de Consulta . . . 7
  - 2.2.1. Expresiones Regulares . . . 8
- 2.3. Estructuras de Datos Sucintas . . . 9
  - 2.3.1. Secuencias Binarias . . . 10
  - 2.3.2. *Directly Addressable Codes* (DACs) . . . 11
  - 2.3.3. k^2-trees . . . 13

3. Propuesta **16**
- 3.1. Evaluación de consultas . . . 16
- 3.2. Representación de los Grafos . . . 18
  - 3.2.1. Diccionario de Codificación . . . 19
  - 3.2.2. *Vertical Partitioning* . . . 19

4. Implementación **21**
- 4.1. Implementación de libk2tree . . . 21
  - 4.1.1. Tecnologías utilizadas . . . 21
  - 4.1.2. Variantes de la estructura . . . 22
  - 4.1.3. Construcción . . . 23
- 4.2. Motor de evaluación de consultas . . . 24
  - 4.2.1. Tecnologías utilizadas . . . 24
  - 4.2.2. Expresiones regulares . . . 24
  - 4.2.3. Construcción NFA . . . 26
  - 4.2.4. `K2tdbEngine` . . . 26
  - 4.2.5. Diccionario de Codificación . . . 27
  - 4.2.6. `GDBEngine` . . . 27
  - 4.2.7. Construcción . . . 28

4.2.8. Interfaz interactiva 28

5. Evaluación Experimental **30**
5.1. Creación de las Bases de Datos 30
5.2. Consultas de Prueba 32
5.3. Resultados Obtenidos 33
5.4. Bases de Datos de Grafos Reales 36
5.4.1. Construcción 37
5.4.2. Resultados 38

6. Conclusiones y Trabajo Futuro **39**

Bibliografía **41**

Capítulo 1

Introducción

Desde su aparición el la década de los *70*'s, el modelo relacional [11] junto al lenguaje de consultas SQL ha sido sin duda la elección más empleada al momento de requerir bases de datos. Este éxito puede atribuirse en gran parte a la potencia y formalismos matemáticos en los que se basa. No obstante, en el último tiempo ha surgido interés por encontrar nuevos modelos que se adecuen de mejor forma a las necesidades actuales, dado que el modelo relacional ya no resulta completamente apropiado para algunos escenarios.

La idea de usar grafos como bases de datos no es nueva, en la práctica existen muchos casos donde la información tiene esta estructura y la representación resulta entonces natural. Podemos encontrar las primeras nociones al respecto en la década de los *80*'s [15], sin embargo, el interés por el tema fue en declive al pasar de los años debido a la aparición de XML. Esta situación puso la atención de todos en las estructuras de datos de árbol, las cuales resultaban convenientes en una amplia cantidad de contextos. Recientemente las bases de datos de grafos han vuelto a recobrar interés principalmente por la aparición de la Web y sus tecnologías asociadas. Un ejemplo clásico son las redes sociales que se conforman por objetos y relaciones entre estos o la misma Web que corresponde a un conjunto de documentos relacionados entre si por enlaces.

En la actualidad es posible observar muchas implementaciones de bases datos relacionales y a pesar de las diferencias que puedan encontrarse, todas comparten en los puntos esenciales las mismas características. Posiblemente la mayor prueba de esto es el uso universal de SQL como lenguaje de consultas. Ahora bien, en el caso de las bases de datos de grafos las discusiones están todavía abiertas y aún no existen consensos respecto a la forma en que éstas deberían implementarse, ni tampoco sobre qué lenguaje de consultas es adecuado utilizar. El trabajo en el área se encuentra aún sin cerrar y propone desafíos interesantes.

Una de las principales dificultades al enfrentarse con el diseño de una base de datos es el tamaño que estas pueden alcanzar, y hoy en día los ejemplos que existen de bases de datos de grafos ponen en evidencia la necesidad de encontrar soluciones escalables que puedan soportar grandes volúmenes de información. En este trabajo se propone una alternativa que utiliza las denominadas estructuras de datos sucintas para representar grafos de manera eficiente en memoria. La principal características de este tipo de estructuras es poder representar los

datos de forma comprimida, pero permitiendo acceder a ellos directamente. Así, es entonces posible trabajar con grandes volúmenes de información sin necesidad de llevarlos a disco. Hoy en día existen diversas estructuras que permiten comprimir grafos mientras que es posible navegarlos [10], estas estructuras logran su cometido haciendo uso, entre otras cosas, de las propiedades estadísticas de los grafos para disminuir el tamaño utilizado por la representación.

Adicionalmente se aborda la implementación de un lenguaje de consultas sencillo y ampliamente estudiado. Este lenguaje permite realizar consultas con una expresividad aceptable y es de particular interés pues se presenta como la base para otros lenguajes de consultas de mayor complejidad.

1.1. Justificación y Motivación

En la actualidad las bases de datos de grafos han ganado popularidad y su uso va en aumento. Teniendo esto en consideración, es preocupante ver que algunas implementaciones dejan de lado consideraciones importantes como la escalabilidad o la incorporación de un lenguaje de consultas que nivele adecuadamente la expresividad y la complejidad de evaluación. Resulta entonces importante indagar sobre los aspectos prácticos de estos detalles al momento de realizar una implementación.

A su vez, las bases de datos de grafos son de particular importancia pues en la actualidad son utilizadas en la práctica en diversas áreas del conocimiento. En Química por ejemplo, hay modelos que representan átomos como nodos y los enlaces entre ellos como aristas [18]. Por otro lado en biología existe un uso similar, pero para modelar interacción entre aminoácidos [13]. La Web también puede ser representada como un grafo, lo mismo sucede con las redes sociales y con redes de información. En todos estos casos es necesario mantener la información almacenada de una forma estándar y resulta de utilidad tener una manera de consultar los datos que no requiera un entendimiento de la implementación subyacente. Es por esto que resulta interesante avanzar en la implementación de un sistema de bases de datos de grafo que permita ser utilizado de esta forma.

Por otro lado, las soluciones actuales de bases de datos de grafos utilizan implementaciones en memoria secundaria. Un aporte interesante es entonces la incorporación de estructuras de datos sucintas y realizar una implementación que resida en memoria principal para obtener mejores tiempos de ejecución, pues existen muchos casos en que esta solución resulta adecuada.

1.2. Objetivos

El objetivo general de este trabajo es la realización de un avance concreto en el área de bases de datos de grafos, proporcionando una implementación eficiente de un lenguaje de consulta que presente un buen poder expresivo. Junto con esto se pretende también incorporar los avances en estructuras de datos sucintas a la teoría actual de bases de datos de grafos.

Dentro de los objetivos específicos del trabajo podemos encontrar:

- Escoger un lenguaje de consultas que permita ser implementado eficientemente con estructuras de datos sucintas.
- Diseñar un algoritmo práctico que permita evaluar las consultas del lenguaje escogido.
- Implementar un sistema simple de gestión de bases de datos de grafos que permita evaluar consultas del lenguaje anteriormente escogido, utilizando las capacidades de las estructuras de datos sucintas para mantener grandes volúmenes de información en memoria principal y así conseguir buen tiempo de respuesta para las consultas.
- Comparar la implementación con otros proyectos similares, evaluando principalmente la expresividad y eficiencia de los lenguajes de consulta.
- Finalmente la intención es presentar el trabajo como un proyecto de software libre con el objetivo de proporcionar una herramienta útil para el desarrollo de investigaciones. El código debe presentarse además de manera extensible y ordenada para su posterior uso.

1.3. Resultados y Organización del Documento

En términos concretos este trabajo presenta el diseño de un sistema simple de base de datos de grafos que apuesta por las estructuras de datos sucintas para manejar grandes volúmenes de información. Adicionalmente se aborda la implementación de un lenguaje de consultas sencillo y ampliamente estudiado. Los antecedentes teóricos necesarios para entender la propuesta son revisados en el Capítulo 2, mientras que la propuesta específica es examinada en el Capítulo 3.

El trabajo consiste además en una implementación de la propuesta cuyos detalles de diseño y organización son abordados en el Capítulo 4. Es importante notar que todo el código producido está distribuido de forma totalmente libre y considera la extensibilidad y el orden como componentes principales.

Finalmente el Capítulo 5 presenta un análisis experimental del rendimiento de la implementación en comparación con algunas alternativas existentes. Los resultados permiten concluir que la propuesta es comparable con sistemas de alto nivel de desarrollo y en algunos escenarios supera con creces el desempeño de las alternativas evaluadas.

Capítulo 2

Antecedentes

En este capítulo se discuten los antecedentes y conceptos teóricos necesarios para poder entender este trabajo. Junto con esto se presentan además las definiciones formales utilizadas en el resto del documento. Finalmente se introducen también las implementaciones actuales con las que se compara este trabajo.

2.1. Bases de Datos de Grafos

Una base de datos de grafos (GDB) es una estructura que permite almacenar información modelándola mediante un grafo. En términos simples una GDB almacena información sobre objetos y las relaciones que existen entre ellos. Cada objeto es representado como un nodo, mientras que las aristas hacen referencia a las relaciones existentes entre estos. Por ejemplo, una GDB podría mantener información sobre grupos musicales y los integrantes que las han conformado, en ésta cada músico y grupo sería representado con un nodo y la relación *"haber sido integrante"* estaría dada por las aristas entre los nodos. Ahora bien, podría ser necesario agregar más información al grafo, en el caso del ejemplo podría ser necesario anotar también cuáles son los estilos musicales asociados a cada banda. Esto generaría un nuevo tipo de relación entre los objetos y para modelar esta diferencia las aristas se etiquetan con el fin de que sea posible distinguirlas.

La Figura 2.1 muestra gráficamente una GDB de ejemplo que guarda información de grupos musicales y los músicos que los han integrado. En ésta podemos ver, por ejemplo, que *"Greg Lake"* tocó en las bandas *"King Crimson"* y *"Emeron, Lake and Palmer"*, además es posible darse cuenta que *"King Crimson"*, *"The Nice"* y *"Emerson, Lake and Palmer"* son bandas asociadas al estilo *"Rock Progresivo"*. Notar que la dirección de las relaciones tiene importancia, en este caso la dirección de la relación *"integrante"* indica que son los músicos quienes integran las bandas y no al contrario. No obstante, cada relación siempre sugiere la existencia de una relación inversa, para el ejemplo, la relación *"integrante"* expone la existencia de una relación que indica que una banda *"ha tenido como integrante"* a un músico. Este hecho resulta de importancia al analizar los lenguaje de consultas y será examinado posteriormente.

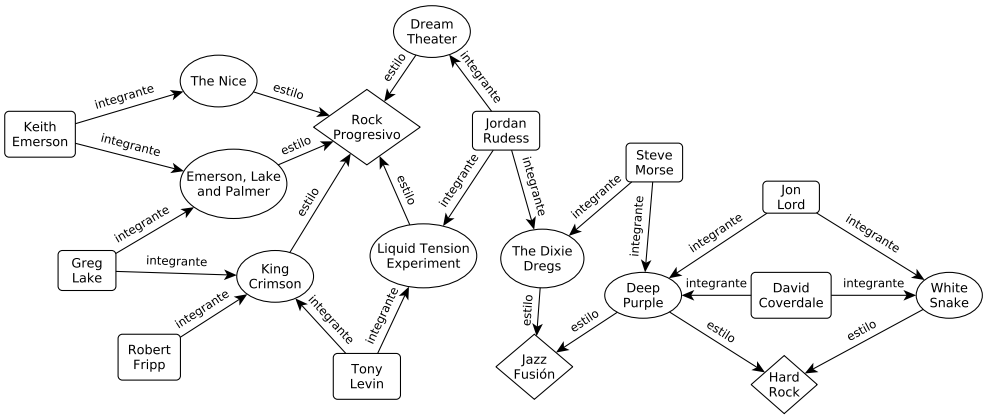


Figura 2.1: Ejemplo de base de datos de grafo

Notar que por simplicidad en el ejemplo de la Figura 2.1 se denota cada objeto utilizando el nombre real con el que se conoce a cada banda o músico. Debido a coincidencias de nombres, en la práctica esto no permite identificar de manera única cada objeto y es necesario la incorporación de identificadores únicos. El nombre real o cualquier otra información extra puede anotarse, dependiendo del modelo, mediante nuevas relaciones o con la utilización de atributos en los nodos.

Un aspecto importante a considerar en la implementación de una GDB es su tamaño. En las aplicaciones actuales la información que se utiliza contiene cientos o miles de millones de relaciones entre objetos, lo que supone un desafío al momento de realizar un implementación que permita trabajar de manera práctica con los datos. Este problema es una de las principales consideraciones de este trabajo.

2.1.1. Modelos Formales

Muchos modelos de bases de datos de grafo han sido propuestos [2], pero todos conservan la definición matemática básica de grafo: un conjunto finito de nodos o vértices que representan las entidades atómicas, y un conjunto de arcos o aristas que representa las relaciones entre los datos. Como se mencionó anteriormente, una extensión natural es etiquetar las aristas con el objetivo de representar la diferencia entre los tipos de relaciones. Este concepto básico de base datos de grafo es posible de extender, agregando diferentes características y capacidades expresivas. Por ejemplo, es posible agregar atributos a los nodos, permitir que las etiquetas de las aristas cumplan también el rol de vértices o definir diferentes tipos de nodos. Por otro lado, algunos modelos extienden el concepto de arco al de *hyperedge*, permitiendo conectar varios nodos a través de una sola "arista".

En este trabajo se considera la definición intuitiva original que se presentó, pues tiene características lo suficientemente expresivas y se presenta como punto base en el estudio de alternativas que añaden más complejidad. El modelo utilizado está compuesto principalmente de un conjunto de nodos y un conjunto de aristas etiquetadas, destacando que estas últimas tienen dirección. A continuación se muestra la formalización de la definición.

Definición 2.1 (Base de datos de grafos) *Sea $\mathcal{V}$ un conjunto de identificadores y Σ un alfabeto finito de etiquetas. Una base de datos de grafos (GDB) G sobre Σ es una par (V, E) donde:*

- *$V \subseteq \mathcal{V}$ es un conjunto finito de identificadores, llamado los nodos de G.*
- *$E \subseteq V \times \Sigma \times V$ es el conjunto de arcos etiquetados en Σ.*

La interpretación de un arco $(u, c, v) \in E$, corresponde a la existencia de una arista con etiqueta c desde el nodo u hacia el nodo v.

2.1.2. Implementaciones actuales

En los últimos años ha habido un incremento en la cantidad de gestores de bases de datos de grafos cada uno con diferentes beneficios y desventajas. Por sus características, en este trabajo fueron escogidos algunas alternativas con el fin de compararlas con la implementación propuesta.

Sparksee

Anteriormente conocido como *DEX*, *Sparksee* promete ser una biblioteca de alto rendimiento para gestionar grafos de gran tamaño (mil millones de objetos). La implementación soporta grafos dirigidos, etiquetados y con atributos en los nodos. Desarrollado como una biblioteca en `C++` con enlace a otros lenguajes, no provee un lenguaje de consulta propio, pero permite responder algunas consultas de navegación mediante la API que posee. Es de licencia comercial aunque permite una descarga gratuita con fines académicos.

Neo4j

Neo4j es una base de datos de grafos que provee su propio lenguaje de consultas. Puede utilizarse como una aplicación *standalone* o a través de una API en Java que presenta una rica interfaz. El proyecto está disponible bajo una licencia personal, gratuita y sin restricciones, pero a su vez se proveen versiones comerciales con soporte por la empresa.

Virtuoso Universal Server

En palabras de sus creadores, *Virtuoso* es un innovador servidor de datos multi-modelo para personas y empresas ágiles. En su concepción el proyecto no corresponde a una base de datos de grafos, pero soporta el modelo RDF utilizando una base de datos relacional para su implementación. Una característica interesante es que implementa las especificaciones de SPARQL1.1 y en particular soporta el tipo de consultas evaluadas en en este trabajo. Es de licencia comercial, pero provee una descarga gratuita con fines de evaluación.

2.1.3. Apache Jena

Jena es un *framework* libre y de código abierto escrito en Java para la construcción de aplicaciones sobre la web semántica. Mediante una serie de de APIs, el *framework* permite interactuar y procesar datos RDF. Al igual que *Virtuoso*, *Jena* implementa las consultas consideradas en este trabajo a través de SPARQL1.1.

2.2. Lenguajes de Consulta

Una consideración importante al momento de analizar una base de datos es el lenguaje de consultas utilizado para obtener información de ella. El uso de estos es indiscutiblemente recomendado pues permite abstraer los detalles de implementación, proveyendo una manera cómoda de acceder a la información de manera eficiente. Por ejemplo, el modelo relacional debe gran parte de su popularidad a la enorme capacidad expresiva que provee SQL, lenguaje que permite realizar consultas complejas obteniendo los resultados de manera rápida.

En el caso de las bases de datos de grafos, no es claro qué lenguaje de consultas utilizar y la elección de uno requiere realizar consideraciones complejas. Por un lado, se quiere tener un lenguaje de consultas que presente un alto nivel de expresividad, pero en general el aumento de esto acarrea también un aumento en el tiempo necesario para evaluar las consultas. Elegir un lenguaje muy expresivo podría generar la existencia de consultas intratables desde el punto de vista computacional y por lo tanto es necesario encontrar un equilibrio. Esta tarea no es sencilla dada la complejidad inherente que supone responder consultas sobre grafos.

A lo largo del tiempo se han propuesto diversas alternativas de lenguajes de consultas para grafos [19], cada una con sus beneficios y desventajas. Entre los intentos de estandarización, mención especial requiere el caso de SPARQL[1], especificación elaborada por la W3C como lenguaje de consultas para bases de datos RDF. SPARQL ha empezado a ganar popularidad y determinar una especificación que alcance el equilibrio adecuado entre expresividad y complejidad de evaluación es de suma importancia. Así mismo el estudio de alternativas para lenguajes de uso más general es también un punto delicado.

En cierto sentido, las consultas más naturales que pueden hacerse sobre un grafo son las que buscan encontrar patrones sobre éste. Considere por ejemplo la base de datos de la Figura 2.1 y suponga que desea encontrar todas las bandas que están asociadas a la vez a los estilos *Rock Progresivo* y *Hard Rock*. Obviando el formato de los identifiadores, en SPARQL esta consulta se realizaría de la siguiente forma:

```
SELECT ?x
WHERE {
  ?x estilo rock_progresivo.
  ?x estilo hard_rock.
}
```

[1] `http://www.w3.org/TR/rdf-sparql-query/`

Aquí `?x` representa una variable y la idea de la consulta es encontrar todos los nodos que cumplen el patrón si son reemplazados por `?x`. Para analizar otro ejemplo, considere una consulta que requerire encontrar todos los músicos que han tocado en una banda asociada a los estilos *Rock Progresivo* y *Hard Rock*.

```
SELECT ?x ?y
WHERE {
  ?x estilo rock_progresivo.
  ?x estilo hard_rock.
  ?y integrante ?x.
}
```

Esta vez la consulta tiene dos variables y la respuesta corresponderá a todos los pares músico-banda que cumplen el patrón. De esta forma es posible realizar consultas más complejas que búsquen encontrar subgrafos que cumplen cierto patrón.

Las consultas de este tipo corresponden a consultas conjuntivas y en el caso general tienen una alta complejidad, pues la evaluación de éstas es escencialmente el problema de isomorfismo de subgrafos, el cual es conocido por ser NP-completo.

A pesar de su expresividad las consultas conjuntivas no permiten realizar algunas consultas interesante sobre grafos, como por ejemplo, determinar si dos nodos están conectados por un camino de largo arbitrario que cumple cierta propiedad. Este trabajo se centra en el estudio de un lenguaje sencillo que permite realizar este tipo de consultas de navegación. La idea básica es usar expresiones regulares para representar caminos de aristas etiquetadas en el grafo. El análisis de esta alternativa es interesante pues permite consultas relativamente expresivas y es la base para otros lenguajes de consultas más complejos. Por ejemplo, SPARQL1.1 soporta este tipo de consultas mediante lo que denominan *property paths*[2].

2.2.1. Expresiones Regulares

Las consultas de este trabajo utilizan la definición usual de expresiones regulares añadiendo un operador inverso que permite capturar la noción de relación inversa en los arcos etiquetados del grafo. A continuación se muestra la definición formal.

Definición 2.2 (Expresión regular) *Sea Σ un alfabeto finito y c^- un operador inverso para cada $c \in \Sigma$. Una expresión regular sobre Σ sigue la sintaxis de la siguiente gramática.*

$$exp \;:=\; \varepsilon \mid c\ (c \in \Sigma) \mid c^-\ (c \in \Sigma) \mid exp/exp \mid exp^* \mid exp + exp \tag{2.1}$$

De manera intuitiva una expresión regular especifica pares de nodos que están conectados por un camino de arcos etiquetados que satisface cierta condición regular. Es decir, cada expresión regular exp sobre un alfabeto Σ define una relación binaria $[\![exp]\!]_G$ cuando es

[2] `http://www.w3.org/TR/sparql11-query/#propertypaths`

evaluada en una base de datos de grafos G sobre Σ. Esta relación binaria puede ser definida de manera inductiva de la siguiente forma

Definición 2.3 (Relación de evaluación) *Sean n, n_1 y n_2 expresiones regulares arbitrarias sobre un alfabeto Σ, $G = (V, E)$ una GDB sobre Σ y c un símbolo cualquiera en Σ. La relación $[\![\cdot]\!]_G$ se define inductivamente de la siguiente forma.*

$$\begin{aligned}
[\![\varepsilon]\!]_G &= \{(u,u) \mid u \in V\} \\
[\![c]\!]_G &= \{(u,v) \mid (u,c,v) \in E\} \\
[\![c^-]\!]_G &= \{(u,v) \mid (v,c,u) \in E\} \\
[\![n_1/n_2]\!]_G &= [\![n_1]\!]_G \circ [\![n_2]\!]_G \\
[\![n_1 + n_2]\!]_G &= [\![n_1]\!]_G \cup [\![n_2]\!]_G \\
[\![n^*]\!]_G &= [\![\varepsilon]\!]_G \cup [\![n]\!]_G \cup [\![n/n]\!]_G \cup [\![n/n/n]\!]_G \cup \ldots
\end{aligned}$$

Aquí el símbolo $\circ$ denota la composición usual de relación binarias, esto es

$$[\![n_1]\!]_G \circ [\![n_2]\!]_G = \{(u,v) \mid \exists w, (u,w) \in [\![n_1]\!]_G \text{ y } (w,v) \in [\![n_2]\!]_G\}$$

Ejemplo Sea G el grafo de la Figura 2.1, la siguiente expresión regular pone en relación todos los pares (u, v) tal que u y v han sido integrantes de la misma banda.[3]

$$n_1 = \texttt{integrante}/\texttt{integrante}^-$$

Por ejemplo, ("Greg Lake", "Robert Fripp") y ("Keith Emerson", "Greg Lake") están en $[\![n_1]\!]_G$.

Considere ahora la siguiente expresión, que pone en relación todos los pares (u, v) tales que u y v están conectados por un camino de músicos que han sido integrantes de la misma banda.

$$n_2 = (\texttt{integrante}/\texttt{integrante}^-)^*$$

Por ejemplo, el par ("Keith Emerson", "Tony Levin") pertenecen a $[\![n_2]\!]_G$. Las expresiones de este estilo, es decir, que definen caminos de largo arbitrario, son de particular importancia y dotan al lenguaje de la expresividad requerida para abordar casos de uso complejos.

2.3. Estructuras de Datos Sucintas

Las estructuras de datos sucintas buscan representar información usando la menor cantidad de memoria posible. En muchos casos logran alcanzar el espacio teórico mínimo para representar la información original, pero otorgando al mismo tiempo acceso directo a los datos. De esta forma es posible trabajar de manera directa con la información comprimida, permitiendo mantenerla en niveles más rápido de memoria y así lograr un mejor desempeño. En esta sección se revisan los conceptos sobre estructuras de datos sucintas involucrados en este trabajo.

[3]El lector atento podrá haberse percatado que la forma en que se define la expresión también pone en relación a cada nodo que haya sido integrante de una banda con sigo mismo.

2.3.1. Secuencias Binarias

Las secuencias binarias (*bitstrings*) son la base para muchas estructuras de datos sucintas. Una secuencia binaria $\mathcal{B}[1, n]$ almacena una secuencia de n bits y permite de manera eficiente realizar las siguientes operaciones:

- $\texttt{rank}_a(\mathcal{B}, i)$ cuenta las ocurrencias del bit a en $\mathcal{B}[1, i]$.
- $\texttt{select}_a(\mathcal{B}, i)$ encuentra la posición de la i-ésima ocurrencia del bit a en $\mathcal{B}$.
- $\texttt{access}(\mathcal{B}, i)$ devuelve el bit en la posición i-ésima, $\mathcal{B}[i]$.

Realizar estas operaciones de manera eficiente es fundamental para su posterior uso, objetivo que debe ser logrado utilizando el menor espacio posible. En teoría, estas operaciones pueden resolverse en tiempo constante y usando sólo $n+o(n)$ bits de memoria total: n bits para almacenar la secuencia y $o(n)$ bits para las estructuras que permiten resolver las consultas de manera eficiente.

La solución clásica para resolver las consultas de `rank` en tiempo constante es relativamente sencilla. Primero se divide la secuencia en bloques de largo $b = \lfloor \log(n)/2 \rfloor$, los cuales son a su vez agrupados en superbloques de tamaño $s = b\lfloor \log n \rfloor$.

Para cada superbloque j $(0 \leq j < \lceil n/s \rceil)$ se precomputa la cantidad de 1s hasta el comienzo del superbloque y la información es guardada en un arreglo R_s, de forma que $R_s[j] = \texttt{rank}_1(\mathcal{B}, j \cdot s)$. El arreglo R_s guarda potencialmente enteros de tamaño n y es de largo $\lceil n/s \rceil$, por lo tanto ocupa $O(n/\log n)$ bits.

Para cada bloque k $(0 \leq k < \lceil n/b \rceil)$ ubicado en el superbloque $j = \lfloor k/\lfloor \log n \rfloor \rfloor$, se guarda en un arreglo R_b la cantidad de 1s desde el comienzo del superbloque, de forma que $R_b[k] = \texttt{rank}_1(\mathcal{B}, k \cdot b) - \texttt{rank}_1(\mathcal{B}, j \cdot s)$. Estos números pueden tener en el peor caso tamaño s y por lo tanto el arreglo R_b ocupa $O(n \log\log n/\log n)$ bits de memoria.

Finalmente, para todas las secuencia de bits $\mathcal{S}$ de largo b y cada posición i en $\mathcal{S}$ se precomputa una tabla R_p, de forma que $R_p[\mathcal{S}, i] = \texttt{rank}_1(\mathcal{S}, i)$. Esta estructura requiere $O(2^b \cdot b \cdot \log b) = O(\sqrt{n} \log n \log\log n)$ bits.

La suma de todas las estructuras requiere $O(n/\log n + n \log\log n/\log n + \sqrt{n} \log n \log\log n) = o(n)$ bits de espacio y puede computar las consultas de `rank` en tiempo constante de la siguiente forma,

$$\texttt{rank}_1(\mathcal{B}, i) = R_s[\lfloor i/s \rfloor] + R_b[\lfloor i/b \rfloor] + R_p[\mathcal{B}[\lfloor i/b \rfloor \cdot b + 1, \ldots, \lfloor i/b \rfloor \cdot b + b], i \bmod b]$$

Esta estrategia puede ser implementada en la práctica sin mucho esfuerzo, pero a pesar de las garantías teóricas de espacio el *overhead* puede ser demasiado alto. Por ejemplo, para $n = 2^{30}$ los $o(n)$ bits de espacio extra corresponden a aproximadamente un 66 % de n, lo cual en la práctica no es para nada despreciable. Teniendo en consideración cosas como la alineación de memoria, la localidad de referencia o la cantidad de accesos a memoria, puede lograrse una implementación más eficiente.

Para la realización de este trabajo se considera una implementación que utiliza un 5 % de espacio extra por sobre la secuencia original y proporciona respuestas rápidas para las consultas [12]. La propuesta sugiere usar sólo un nivel de bloques, R_s, de tamaño $s = 32 \cdot k$. De esta forma los bloques quedan alineados al tamaño de la palabra. Las respuestas a los *ranks* parciales para cada bloque son guardadas en enteros de 32-bits. Para una implementación en una arquitectura de 64-bits los *ranks* parciales para cada bloque deben ser guardados en enteros de 64-bits, esto duplica el tamaño necesario para guardar el muestreo, pero en cambio es posible aumentar el tamaño de los bloques a $s = 64 \cdot k$ manteniendo el mismo *overhead*. Las consultas se resuelven obteniendo el valor en el bloque correspondiente y después contando secuencialmente la cantidad de 1s entre bloques. Esta última operación puede ser realizada de manera eficiente haciendo uso de la operación `popcount`, la cual está implementada en los compiladores más utilizados e incluso integrada en algunas arquitecturas. Esta operación permite contar la cantidad de unos en un entero y dependiendo de la arquitectura puede resolverse en el mismo tiempo para enteros de 32 y 64 bits.

Finalmente, en teoría es posible resolver la operación `select` en tiempo constante, pero la implementación de la estrategia es demasiado compleja y produciría un *overhead* demasiado grande. En la práctica una solución sencilla y relativamente eficiente es evaluar las consultas de `select` haciendo búsqueda binaria en la operación `rank`. Esto supone un aumento en el tiempo de ejecución en relación a la operación `rank` que debe ser considerado, no obstante la operación `select` no es ocupada en las estructuras involucradas en este trabajo.

2.3.2. *Directly Addressable Codes* (DACs)

La asignación de códigos de largo variable es uno de los principios fundamentales en compresión. En esta estrategia se codifican los símbolos más frecuentes de una secuencia usando códigos de menor longitud, mientras que para los símbolos menos frecuentes se utilizan códigos de mayor largo. La secuencia se reemplaza posteriormente por la concatenación de la codificación correspondiente a cada símbolo. Si bien está representación puede lograr una alta compresión, dificulta el acceso aleatorio a símbolos en la secuencia, pues es necesario recorrer toda la secuencia para saber la posición donde comienza un código. Brisaboa, Ladra y Navarro [8], proponen una solución práctica para este problema presentando una estrategia que denominan *Directly Addressable Codes* (DACs).

La propuesta parte con una secuencia de códigos de largo variable $\mathcal{C} = \mathcal{C}_1, \mathcal{C}_2 \ldots, \mathcal{C}_n$. Cada código es posteriormente dividido en fragmentos de b bits, de forma que el código $\mathcal{C}_i$ está compuesto por una secuencia de $r_i = \lceil |\mathcal{C}_i|/b \rceil$ fragmentos, $\mathcal{C}_{i,r_i}, \ldots, \mathcal{C}_{i,2}, \mathcal{C}_{i,1}$. La secuencia puede ser vista como una concatenación de estos fragmentos, los cuales son reordenados en una estructura de varios niveles. En el primer nivel, la secuencia $\mathcal{A}_1$ contendrá los $n_1 = n$ fragmentos menos significativos de cada código, $\mathcal{A}_1 = \mathcal{C}_{1,1}, \mathcal{C}_{2,1}, \ldots, \mathcal{C}_{n,1}$. En el segundo nivel, la secuencia $\mathcal{A}_2$ contendrá los siguientes fragmentos menos significativos de los n_2 códigos que están compuestos por más de un fragmento. Posteriormente se procede de manera similar con $\mathcal{A}_3$ y así sucesivamente. Siendo m el tamaño en bits del código más largo, se necesitarán $L = \lceil m/b \rceil$ niveles con secuencias $\mathcal{A}_k$.

Para permitir reconstruir los códigos se agrega en cada nivel una secuencia binaria $\mathcal{B}_k$

que indica para cada código si existe un fragmento en el siguiente nivel. El bit i-ésimo de $\mathcal{B}_k$ contendrá un 1 si el código correspondiente a $\mathcal{A}_k[i]$ posee un fragmento en $\mathcal{A}_{k+1}$ o un 0 si está compuesto solo por k fragmentos. Para encontrar la posición del fragmento en $\mathcal{A}_{k+1}$ es necesario hacer consultas de `rank` sobre los bitmaps. En la figura 2.2 se muestra una secuencia de ejemplo y la reordenación de los fragmentos junto a los bitmaps de cada nivel.

$\mathcal{C}_{1,2}\mathcal{C}_{1,1}$	$\mathcal{C}_{2,1}$	$\mathcal{C}_{3,3}\mathcal{C}_{3,2}\mathcal{C}_{3,1}$	$\mathcal{C}_{4,2}\mathcal{C}_{4,1}$	$\mathcal{C}_{5,4}\mathcal{C}_{5,3}\mathcal{C}_{5,2}\mathcal{C}_{5,1}$

$\mathcal{A}_1$	$\mathcal{C}_{1,1}$	$\mathcal{C}_{2,1}$	$\mathcal{C}_{3,1}$	$\mathcal{C}_{4,1}$	$\mathcal{C}_{5,1}$
$\mathcal{B}_1$	1	0	1	1	1

$\mathcal{A}_2$	$\mathcal{C}_{1,2}$	$\mathcal{C}_{3,2}$	$\mathcal{C}_{4,2}$	$\mathcal{C}_{5,2}$
$\mathcal{B}_2$	0	1	0	1

$\mathcal{A}_3$	$\mathcal{C}_{3,3}$	$\mathcal{C}_{5,3}$
$\mathcal{B}_3$	0	1

$\mathcal{A}_4$	$\mathcal{C}_{5,3}$
$\mathcal{B}_4$	0

Figura 2.2: Ejemplo de reorganización de los fragmento en los códigos.

Con esta representación es posible obtener directamente los elementos de la secuencia codificada siguiendo un proceso sencillo. Para acceder al elemento en la posición $i = i_1$ se obtienen sus fragmentos por nivel. En primer lugar se revisa el valor de $\mathcal{B}_1[i_1]$, si este valor es 0 significa que el código está compuesto sólo por un fragmento y la reconstrucción termina ahí, siendo $\mathcal{C}[i] = \mathcal{A}[i]$. En caso contrario ($\mathcal{B}_1[i_1] = 1$), el código posee un fragmento en el siguiente nivel, el cual se encuentra en la posición $i_2 = \texttt{rank}(\mathcal{B}_1, i_1)$. Del mismo modo que en la etapa anterior, si $B_2[i_2] = 0$ entonces ya se encontró el último fragmento y $\mathcal{C}[i] = \mathcal{A}[i_2] \cdot 2^b + A[i_1]$. En caso contrario se prosigue de manera iterativa hasta alcanzar un nivel k donde $\mathcal{B}_k[i_k] = 0$ y el código es reconstruido completamente.

Acceder a un código en una secuencia comprimida con DACs toma en el peor caso tiempo $O(L = \lceil m/b \rceil)$, donde m es el tamaño en bits del código más largo. No obstante para los elementos con códigos más cortos el tiempo es menor y estos resultan ser los más frecuentes.

Hasta aquí la propuesta considera un largo fijo para los fragmentos de todos los niveles, sin embargo, es posible escoger un valor b_k distinto para cada nivel. La propuesta presenta una forma de encontrar la manera óptima de escoger los b_k y L de forma que se logre la mayor compresión posible. Esos valores óptimos pueden ser encontrados utilizando un algoritmo de programación dinámica que considera el siguiente subproblema: organizar de manera óptima los códigos que poseen t o más número de bits, sin considerar los t bits menos significativos. La respuesta al problema completo se obtiene con $t = 0$ y se resuelve de manera trivial para $t = m$. En un caso intermedio se puede probar para cada $t < i \leq m$ agrupar desde el bit t-ésimo hasta el $(i-1)$-ésimo en un fragmento y ubicándolo en el nivel actual. Posteriormente se debe resolver el subproblema para $t' = i$ usando los siguientes niveles. El tamaño requerido por elemento codificado corresponde a $i - t$ bits para la secuencia $\mathcal{A}_k$ más $1 + \varepsilon$ bits para $\mathcal{B}_k$, donde ε corresponde al *overhead* requerido por la estructura de `rank`.

En la práctica la implementación del algoritmo considera que las posiciones i y t deben ser escogidas de manera que estén alineadas según el tamaño de un byte. De esta forma los accesos a memoria pueden realizarse de manera eficiente.

2.3.3. k^2-trees

Un k^2-tree es una estructura de datos sucinta para la representación de grafos [10]. En ésta se modela un grafo de n nodos a partir de su matriz de adyacencia $\{a_{i,j}\}$ de tamaño $n \times n$. En la matriz cada fila y columna representa un nodo del grafo y la celda $a_{i,j}$ contiene un 1 si existe una arista entre los nodos representados en la fila i y la columna j, en caso contrario la celda contendrá un 0. La idea principal en un k^2-tree es explotar las propiedades de dispersión y *clustering* para representar la matriz de adyacencia de manera compacta. Fundamentalmente la estrategia explota la existencia de grandes áreas "vacías" (que sólo contienen 0s), representándolas en pocos bits.

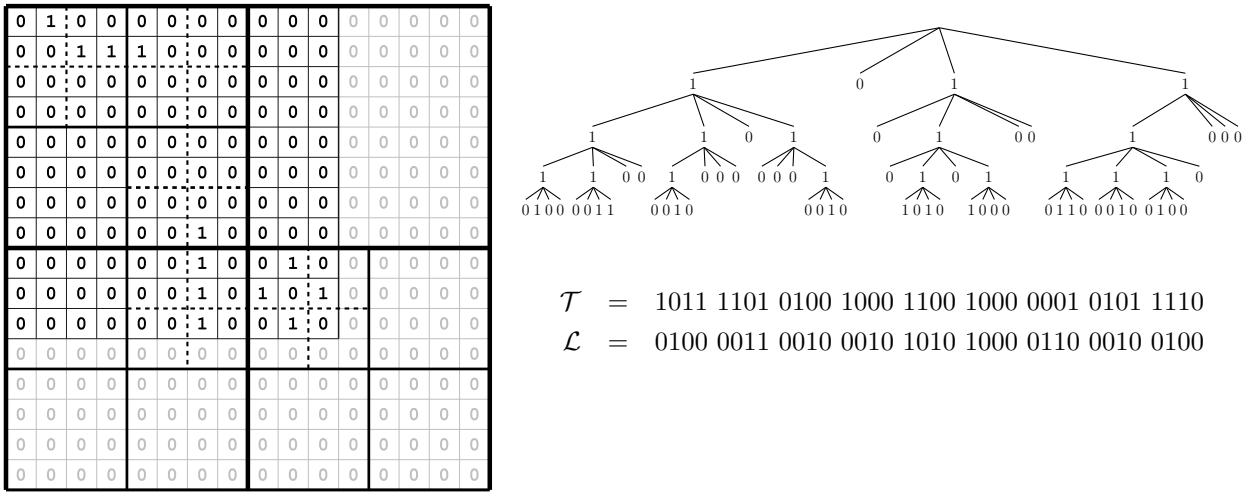
La matriz de adyacencia es representada con un árbol de aridad k^2 y altura $h = \lceil \log_k n \rceil$. Cada nodo del árbol contiene un solo bit de información, 1 para los nodos internos y 0 para las hojas. Esto es así excepto para el último nivel en el cual pueden haber hojas con un 1 y donde se representan los valores de celdas en la matriz original. Todos los nodos internos (con valor 1 y no en el último nivel) tienen exactamente k^2 hijos, mientras que las hojas (nodos con valor 0 o en el último nivel) tienen 0 hijos.

Se asume en primer lugar que n es una potencia de k, en caso contrario la matriz se extiende agregando 0s hasta alcanzar el tamaño requerido. Como los k^2-trees representan grandes zonas de 0s en poco espacio, esta operación no genera un *overhead* demasiado grande. La estructura del árbol parte con la raíz de éste representando la matriz completa, ésta se divide posteriormente en k^2 submatrices de tamaño n^2/k^2, es decir, k filas y k columnas de submatrices. Cada uno de los k^2 hijos de la raíz representa una de las submatrices y contiene un **1** si la submatriz está compuesta de al menos una celda con un 1 o un **0** si ésta contiene solamente 0s. La descomposición continua de manera recursiva para cada hijo con valor 1 hasta alcanzar una matriz llena de 0s o hasta llegar al último nivel, donde las celdas de la matriz son representadas directamente.

Representación

La estructura del árbol es representada de manera compacta utilizando dos secuencias de bits: $\mathcal{T}$ y $\mathcal{L}$. $\mathcal{T}$ almacena todos los bits excepto los que se encuentran en el último nivel. Los bits son ubicados siguiendo un recorrido por nivel del árbol. Esta representación permite navegar el árbol realizando operaciones de `rank`. Por otro lado $\mathcal{L}$ almacena los bits del último nivel. Se realiza esta separación pues en el último nivel no son necesarias las operaciones de `rank` y puede utilizarse un bitmap plano.

Los k^2-trees proveen varias operaciones de navegación sobre sobre el grafo. Particularmente, dado un nodo v, un k^2-tree permite obtener todos los nodos apuntados por v (vecinos



$\mathcal{T}$ = 1011 1101 0100 1000 1100 1000 0001 0101 1110

$\mathcal{L}$ = 0100 0011 0010 0010 1010 1000 0110 0010 0100

Figura 2.3: Ejemplo de un k^2-tree para matriz de 11×11 y $k = 2$ extraído de [10]

directos) y todos los nodos que apuntan a v (vecinos inversos). Adicionalmente es posible obtener todos las conexiones dentro de una submatriz (*range queries*) y comprobar el valor de una celda determinada. Las consultas de vecinos y vecinos inversos son de particular importancia para este trabajo.

Para obtener los vecinos de un nodo v_i se deben encontrar todos los 1s en la fila i-ésima. Esta operación se puede implementar eficientemente recorriendo el árbol desde arriba hacia abajo. Para esto hay que visitar en cada nodo del árbol todos los hijos con valor 1 que se encuentren dentro de la fila correspondiente. De manera simétrica, responder los vecinos inversos de un nodo v_i se logra encontrando los 1s en la columna i-ésima. La operación fundamental para realizar el recorrido es determinar la posición de los hijos de un nodo. Si $\mathcal{T}$ está indexado desde 0, entonces el hijo i-ésimo de un nodo $\mathcal{T}[x]$ se encuentra en la posición $\texttt{child}_i(x) = \texttt{rank}(\mathcal{T}, x) \cdot k^2 + i - 1$ $(1 \leq i \leq k^2)$.

Variante híbrida

Brisaboa, Ladra y Navarro [10] proponen utilizar una aridad k_l distinta para cada nivel y así mejorar el tiempo de las operaciones sin perjudicar la compresión. La idea es mantener valores mas grandes de k_l para los primeros niveles, donde es menos probable encontrarse con submatrices vacías, y utilizar valores más pequeños en los últimos niveles para no perjudicar la compresión.

En términos específicos la estrategia propone mantener una secuencia $\mathcal{T}_l$ para cada nivel. La estructura y navegación del árbol debe ser modificada de acorde a esto. En primer lugar, el hijo i-ésimo de un nodo $\mathcal{T}_l[x]$ se encuentra ahora en $\mathcal{T}_{l+1}$ en la posición $\texttt{child}_i^l(x) = \texttt{rank}(\mathcal{T}_l, x - 1) \cdot k_l^2 + i - 1$ $(1 \leq i \leq k_l)$. En segundo lugar, hay que extender el tamaño de la matriz a $n' = \prod_{l=0}^{h-1} k_l$, que juega el rol de k^h en el caso homogéneo.

Compresión del último nivel

En un k^2-tree el último nivel se almacena en el bitmap $\mathcal{L}$, el cual representa submatrices de tamaño $k_L \times k_L$ y cada bit equivale a una celda de la matriz original ($k_L = k_{h-1}$). Usar valores pequeños de k para los últimos niveles mejora la compresión pero a su vez incrementa el tiempo de navegación.

Una alternativa para mantener la compresión sin perjudicar el tiempo de navegación, es considerar el último nivel como una secuencia de matrices de tamaño $k_L \times k_L$ y comprimirla. Esto permite tener valores mayores para k_L sin perder compresión. La técnica procede armando un vocabulario con las submatrices de tamaño $k_L \times k_L$ representadas en el último nivel, el cual se ordena por frecuencia. El bitmap $\mathcal{L}$ es posteriormente representado con punteros a posiciones en el vocabulario, la secuencia resultante cumple que los números menores son más frecuentes y por tanto una codificación de largos variables que favorezca a estos es recomendable. La secuencia $\mathcal{L}$ se transforma finalmente en la concatenación de estos códigos de largo variable y para mantener una navegación rápida se utilizan DACs asegurando el acceso directo.

Partición del primer nivel

El análisis de la variante híbrida sugiere que es buena idea utilizar un valor muy grande para k_0. Brisaboa et al. proponen ir un paso más allá y construir un k^2-tree separado para cada una de las k_0^2 submatrices del primer nivel. Esto provee una serie de ventajas que mejoran la navegación y además supone un beneficio para la construcción pues cada subárbol del primer nivel puede ser construido de manera independiente.

Construcción

Brisaboa et al. proponen diversas estrategias para la construcción de un k^2-tree. En este trabajo se considera sólo la alternativa que construye primero un árbol tradicional con punteros, el cual es posteriormente navegado por nivel para construir la representación final. Esta estrategia requiere de mayor memoria para funcionar, pero a su vez permite una construcción rápida y no requiere asunciones especiales respecto al almacenamiento previo del grafo.

Capítulo 3

Propuesta

En este trabajo se propone una estrategia para evaluar consultas que utilizan expresiones regulares, además de una representación compacta de la base de datos de grafo utilizando k^2-trees. En este capítulo se detallan estas estrategias además de darse una definición específica de las consultas soportadas por la propuesta.

3.1. Evaluación de consultas

Dada una base de datos G y una expresión regular exp, calcular $[\![exp]\!]_G$ requiere en el peor caso tiempo cuadrático en el tamaño del grafo y para bases de datos muy grandes este cálculo puede tornarse impracticable. No obstante, suele ser necesario realizar consultas más simples que esto, por ejemplo, determinar si un par de nodos (u, v) está conectado por un camino o dado un nodo u listar todos los vértices que están conectados con u mediante un camino específico. Este tipo de consultas busca pares en $[\![exp]\!]_G$ donde al menos uno de los nodos ha sido fijado. Se verá que estas consultas pueden ser evaluadas eficientemente.

En este trabajo se estudian particularmente dos problemas de evaluación sobre expresiones regulares. El primero es el problema de decisión estándar considerado al momento de estudiar lenguajes de consultas y puede definirse formalmente como sigue. Notar que este problema recibe el par (u, v) como input.

PROBLEMA	:	Problema de decisión para expresiones regulares.
INPUT	:	Una base de datos de grafo G, una expresión regular exp y un par de nodos (u, v).
PREGUNTA	:	¿ $(u, v) \in [\![exp]\!]_G$?

Adicionalmente se considera el siguiente problema de computación que recibe un nodo como input y lista todos los vértices conectados con él mediante un camino específico.

PROBLEMA	:	Problema de computación para expresiones regulares.
INPUT	:	Una base de datos de grafo G, un nodo u y una expresión regular exp
OUTPUT	:	Listar todos los elementos v tales que $(u, v) \in [\![exp]\!]_G$

Es posible demostrar que ambos problemas pueden ser resueltos de manera eficiente siguiendo una estrategia similar, en términos precisos se mostrará que existen algoritmos que resuelven los problemas en tiempo $O(|G| \cdot |exp|)$, donde $|G|$ representa el tamaño de la base de datos de grafos y $|exp|$ el tamaño de la expresión regular.

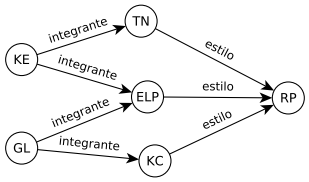
La estrategia es ampliamente conocida y se basa en la construcción de un autómata no determinista con transiciones épsilon (ε-NFA), $\mathcal{A}_{exp}$, que acepta el mismo lenguaje que la expresión regular [16]. La idea continúa considerando el grafo G cómo un NFA y calculando el autómata producto $G \times \mathcal{A}_{exp}$. A partir de aquí el punto clave consiste en observar que si q_0 y q_f son respectivamente el estado inicial y final de $\mathcal{A}_{exp}$, y además para un nodo u existe un vértice v tal que desde (u, q_0) se puede alcanzar el estado (v, q_f) en $G \times \mathcal{A}_{exp}$, entonces $(u, v) \in [\![exp]\!]_G$. Para encontrar todos los elementos u que cumplen lo anterior basta realizar un recorrido de $G \times \mathcal{A}_{exp}$ desde (u, q_0) y listar todos los estados alcanzables que corresponden a un estado final en $\mathcal{A}_{exp}$.

Ejemplo En la Figura 3.1 se muestra el autómata producto correspondiente a la expresión `integrante/integrante`$^-$ y un grafo simple. Podemos notar que el estado (KE, q_0) está conectado mediante un camino con (GL, q_2), y por lo tanto el par (KE, GL) pertenece a $[\![$`integrante/integrante`$^-]\!]_G$.

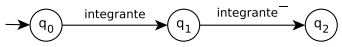
Dicho esto, la resolución de los problemas descritos anteriormente se reduce simplemente a realizar un recorrido lineal sobre un grafo. No obstante, en la práctica no es factible materializar el autómata producto, pero éste puede ser recorrido de manera implícita desde la representación de G y A_{exp}, notando que el autómata producto captura la noción de dos computaciones paralelas. Los Algoritmos 1 y 2 muestran la manera de resolver los problemas de decisión y computación respectivamente. En estos se utiliza en primer lugar la definición tradicional de autómata no determinista: un conjunto de estados Q, un estado inicial q_0, una función de transición $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$ y un conjunto de estados finales F. Además se asume la existencia de la función VECINOS_G definida de la siguiente forma.

$$
\begin{aligned}
\text{VECINOS}_G(u, c) &= \{v \in V \mid (u, c, v) \in E\} \\
\text{VECINOS}_G(u, c^-) &= \{v \in V \mid (v, c, u) \in E\} \\
\text{VECINOS}_G(u, \varepsilon) &= \{u\}
\end{aligned}
$$

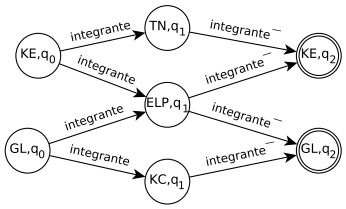
Es claro que la manera de representar los grafos repercute en la eficiencia con que se puede implementar esta función, en especial para el caso en que se recuperan los vecinos inversos, donde representaciones tradicionales tendrían que duplicar el espacio para almacenar el grafo transpuesto. La correctitud de los algoritmos recae en el análisis anterior y el hecho que estos realizan un recorrido en anchura clásico por el autómata producto.



(a) Grafo de ejemplo, G



(b) $\mathcal{A}_{exp}$ para `integrante`/`integrante`$^-$



(c) $G \times \mathcal{A}_{exp}$

Figura 3.1: Ejemplo de un autómata producto.

Algoritmo 1: $\texttt{ExistsPath}^G_{\mathcal{A}_{exp}}(s, e)$

$Q, S \leftarrow \varnothing$
PUSH$(Q, (s, q_0))$
while $Q \neq \varnothing$ **do**
  $(u, q) \leftarrow$ FRONT(Q)
  POP(Q)
  if $q \in F$ **and** $u = e$ **then**
    return true
  for all (r, c) tal que $r \in \delta(q, c)$ **do**
    for all $v \in$ VECINOS$_G(u, c)$ **do**
      if $(v, r) \notin S$ **then**
        $S \leftarrow S \cup \{(v, r)\}$
        PUSH$(Q, (v, r))$
return false

Algoritmo 2: $\texttt{Compute}^G_{\mathcal{A}_{exp}}(s)$

$Q, S \leftarrow \varnothing$
PUSH$(Q, (s, q_0))$
while $Q \neq \varnothing$ **do**
  $(u, q) \leftarrow$ FRONT(Q)
  POP(Q)
  if $q \in F$ **then**
    OUTPUT(u)
  for all (r, c) tal que $r \in \delta(q, c)$ **do**
    for all $v \in$ VECINOS$_G(u, c)$ **do**
      if $(v, r) \notin S$ **then**
        $S \leftarrow S \cup \{(v, r)\}$
        PUSH$(Q, (v, r))$

3.2. Representación de los Grafos

Los algoritmos presentados en la sección anterior dejan abierta la forma de representación del grafo, aunque es claro que la eficiencia de los mismos depende en gran medida de esto. La propuesta de este trabajo es representar los grafos de manera compacta utilizando k^2-trees, permitiendo mantener la información en memoria principal y proveyendo un acceso rápido tanto a los vecinos como los vecinos inversos de un nodo.

3.2.1. Diccionario de Codificación

Considere una base de datos $G = (V, E)$ sobre un alfabeto Σ. La representación propuesta requiere que cada nodo se identifique con un entero único, específicamente, cada nodo es representado con un entero en el rango $[0, |V| - 1]$. De la misma forma cada etiqueta en Σ es identificada con un entero en el rango $[0, |\Sigma| - 1]$. Esta representación permite establecer una correspondencia directa entre los nodos, y las columnas y filas de la matriz de adyacencia.

En la práctica los nodos son representados mediante una cadena de caracteres, por ejemplo, en RDF se utiliza un IRI para identificar cada objeto, esto requiere que se realice una codificación previa de los nodos, mapeando cada identificador con un entero en el rango indicado. Existen diversas formas de representar estos diccionarios de forma comprimida [9] y la forma en que se implemente es un problema ortogonal al trabajo de esta memoria y los procesos relacionados no son evaluados en el desempeño de la propuesta.

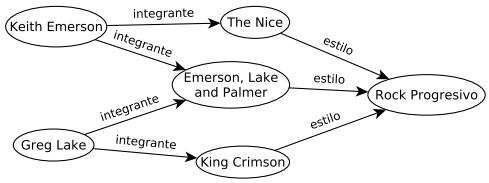
3.2.2. *Vertical Partitioning*

La representación sigue con una estrategia conocida y denominada *vertical partitioning* [1]. Este esquema considera de forma independiente los subgrafos inducidos al considerar aristas de solo una etiqueta y cada subgrafo inducido por el conjunto de arcos con etiqueta $c \in \Sigma$ es representado con un k^2-tree independiente.

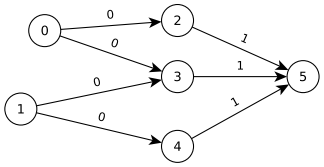
Dada la codificación de las etiquetas en Σ, cada subgrafo puede ser indexado con el identificador correspondiente. De esta forma la función VECINOS_G utilizada en los algoritmos 1 y 2 es implementada simplemente recorriendo los vecinos o vecinos inversos de un nodo en el árbol correspondiente a la etiqueta especificada.

En este trabajo cada k^2-tree usa una versión híbrida configurada con $k = 4$ para los primeros 5 niveles y $k = 2$ para el resto. El último nivel de cada árbol guarda una secuencia de submatrices de tamaño 8×8 y es comprimido utilizando DACs.

La Figura 3.2 muestra un ejemplo de representación para un grafo simple. Primero se codifican los nodos y aristas para obtener un grafo como el de la Figura 3.2b. Posteriormente se construye la matriz de adyacencia para cada subgrafo inducido. En la Figura 3.2c se muestran en orden la matriz de adyacencia para las aristas 0 (integrante) y 1 (estilo). Finalmente un k^2-tree es construido para cada una de estas matrices.



(a) Grafo original.



Codificación nodos	
`Keith Emerson`	0
`Greg Lake`	1
`The Nice`	2
`Emerson, Lake and Palmer`	3
`King Crimson`	4
`Rock Progresivo`	5

Codificación aristas	
`integrante`	0
`estilo`	1

(b) Codificación.

0	0	1	1	0	0
0	0	0	1	1	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	0	1
0	0	0	0	0	1
0	0	0	0	0	0

(c) Matriz de adyacencia para los grafos inducidos.

Figura 3.2: Ejemplo de representación propuesta para grafos.

Capítulo 4

Implementación

La implementación de este trabajo se dividió en dos componentes principales. Por un lado se elaboró una biblioteca que provee las funcionalidades de un k^2-tree y por otro se implementó un sistema de evaluación de consultas. En este capítulo se detalla la estructura general y clases principales existentes en el código.

4.1. Implementación de libk2tree

Con el fin de proveer una herramienta que permita apoyar la realización de nuevas aplicaciones, se desarrolló una biblioteca con una implementación de k^2-trees de manera totalmente independiente[1]. El código está distribuido en su totalidad bajo una licencia de código libre y está permitido cualquier uso o modificación de éste. En este capítulo se discuten las principales consideraciones y decisiones en el diseño e implementación del código.

Se optó por implementar estas funcionalidades desde cero principalmente debido a que las implementaciones existentes no presentan códigos limpios y extensibles al nivel requerido por los objetivos de esta memoria. Así mismo, la reimplementación permitió realizar el desarrolló considerando arquitecturas de 64 bits, lo cuál es de suma importancia para permitir indexar estructuras de gran tamaño.

4.1.1. Tecnologías utilizadas

La biblioteca tiene como objetivo principal lograr implementar las estructuras de forma eficiente tanto en tiempo como memoria, pero a su vez proveer un código legible, extensible y *mantenible*. Con esto en consideración, el lenguaje seleccionado fue `C++` debido a sus características que permiten la manipulación consciente de la memoria y la posibilidad de utilizar *templates* para lograr patrones de diseño complejos que no comprometen la eficiencia en tiempo de ejecución.

[1] `https://github.com/nilehmann/libk2tree`

El código hace uso de algunas de las bibliotecas provistas por Boost[2] y utiliza funcionalidades de C++11. Adicionalmente se requiere el uso de libcds[3], una biblioteca que provee algunas estructuras de datos sucintas y de la cual se obtiene la implementación de secuencias binarias con capacidad de `rank`.

Finalmente, el código sólo utiliza funcionalidades estándares del lenguaje además de librerías multiplataforma y por lo tanto no debería tener problemas de portabilidad. No obstante sólo fue probado en un entorno linux de 64 bits, utilizando los compiladores `gcc4.9` y `clang3.5`.

4.1.2. Variantes de la estructura

Brisaboa et al. [10] proponen distintas variantes para mejorar el comportamiento de la estructura pero que mantienen la esencia de ésta. Considerando todas estas variantes, y con el fin de evitar la duplicación, se decidió factorizar el código en clases bases que implementan las funcionalidades para recorrer y representar las partes comunes a más de una variante.

Variante híbrida

La variante híbrida permite en teoría, escoger un valor distinto de k para cada nivel, no obstante en la implementación se decidió permitir un valor k_1 para los primeros **`k1_levels`** niveles y un valor k_2 para el resto. Adicionalmente se puede proporcionar la aridad k_L para el penúltimo nivel con el fin de configurar el tamaño del vocabulario en las hojas para su posterior compresión. La implementación permite cualquier combinación de valores y el tamaño M de la matriz es extendido para que sea igual a la multiplicación del k en cada nivel. Para determinar el número de niveles con aridad k_2 se busca el menor entero x que satisface $k_1^{\texttt{k1\_levels}} \cdot k_2^x \cdot k_L \geq M$.

Sin importar si se decide comprimir o no el último nivel, en ambos casos el recorrido de los nodos internos se realiza de la misma forma. Esta funcionalidad común fue implementada en la clase **`base_hybrid<Hybrid>`**, la cual provee todo lo necesario para representar y recorrer los niveles internos, delegando a instancias concretas la responsabilidad de visitar los nodos del último nivel. La delegación se hace utilizando una estrategia conocida como *curiously recurring template pattern*[4].

La representación de los nodos internos utiliza un solo *bitstring* $\mathcal{T}$ con la concatenación de las secuencias $\mathcal{T}_l$ de cada nivel y por lo tanto la forma de encontrar la posición de los hijos de un nodo debe ser modificada de acorde a esto. Con esta representación el hijo i-ésimo de un nodo $\mathcal{T}[x]$ ubicado en el nivel l, se encuentra en la posición $\texttt{child}_i^l(x) = (\texttt{rank}(\mathcal{T}, x-1) - \texttt{rank}_{l-1}) \cdot k_l^2 + \texttt{offset}_{l+1} + i - 1$ $(1 \leq i \leq k_l)$. Donde $\texttt{rank}_l$ corresponde a la cantidad de 1's acumulados hasta el nivel l y $\texttt{offset}_l$ a la posición en $\mathcal{T}$ donde comienza el nivel l.

La clase **`base_hybrid<Hybrid>`** implementa los algoritmos **`CheckLink`**, **`DirectLinks`**, **`InverseLinks`** y **`Range`** realizando un recorrido por nivel en vez de usar una estrategia recursiva. El recorrido se hace de manera natural hasta llegar a las hojas, donde la responsabilidad de recorrer los bits en el último nivel es delegada a una implementación concreta.

[2] `http://www.boost.org/`
[3] `https://github.com/fclaude/libcds`
[4] `http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern`

La clase **HybridK2Tree** extiende **base_hybrid<HybridK2Tree>** y representa el último nivel con un bitmap plano. El acceso a las hojas corresponde en este caso simplemente a verificar una posición en el bitmap.

Compresión de las hojas

La clase **CompressedHybrid** extiende **base_hybrid<CompressedHybrid>** y representa el último nivel de manera comprimida utilizando una implementación de DAC's externa[5]. La clase guarda el vocabulario de las hojas, el cual puede ser común a varias instancias.

Para poder comprimir las hojas es necesario construir primero un k^2-tree sin compresión y posteriormente una instancia del mismo árbol pero con las hojas comprimidas puede ser obtenida llamando al método **CompressLeaves**. Como las instancias son inmutables los árboles pueden compartir la secuencia $\mathcal{T}$ y sólo es necesario crear la representación comprimida de las hojas.

División del primer nivel

Esta versión utiliza una de las variantes híbridas para representar las submatrices y puede recibir como parámetro la aridad del primer nivel o el tamaño de cada submatriz. La implementación simplemente guarda una matriz de k^2-trees y llama al método correspondiente del árbol asociado a cada submatriz necesaria. La clase **base_partition<K2Tree>** implementa esta funcionalidad dejando abierta la variante usada para representar las submatrices. Esta variante debe ser provista como un parámetro de plantilla.

La clase **K2TreePartition** extiende de **base_partition<HybridK2Tree>** fijando **HybridK2Tree** como la representación de las submatrices. Llamando al método **CompressLeaves** se puede obtener una instancia de **CompressedPartition** que usa **CompressedHybrid** para representar las submatrices, utilizando un vocabulario compartido para todos los subárboles.

4.1.3. Construcción

La construcción de los árboles se realiza en una clase independiente de forma que sea posible soportar las distintas alternativas. Actualmente sólo está implementada la variante que primero construye un árbol tradicional con punteros y a partir de éste arma los bitmaps necesarios para la representación.

De manera específica se puede encontrar implementada la clase **K2TreeBuilder** que permite construir variantes híbridas. Además **K2TreePartitionBuilder** utiliza la clase anterior para construir de forma separada un k^2-tree para cada submatriz en el caso de la variante que divide el primer nivel. Ambas clases construyen la variante correspondiente sin hacer compresión en las hojas.

[5] `http://lbd.udc.es/research/DACS/`

4.2. Motor de evaluación de consultas

La segunda parte de este desarrollo contempla la implementación de los algoritmos de evaluación discutidos en el capítulo anterior. Todo el código de este componente está a libre disposición y distribuido en un repositorio en Github[6] bajo el nombre de *k2tdb*. En esta sección se discuten las consideraciones generales de la implementación.

4.2.1. Tecnologías utilizadas

Este componente está desarrollado en su totalidad en `C++` y utiliza características de la versión 11. La implementación hace uso de la biblioteca libk2tree desarrollada en este mismo trabajo, además de algunas componentes de Boost[7]. Adicionalmente se utiliza la biblioteca Kyoto Cabinet[8] que implementa rutinas para la gestión de bases de datos.

De la misma forma que en libk2tree, todo el desarrollo fue llevado a cabo utilizando componentes estándares del lenguaje y bibliotecas multiplataforma, pero sólo fue probado en un entorno linux de 64 bits usando los compiladores `gcc` y `clang`.

4.2.2. Expresiones regulares

Para representar las expresiones regulares se utilizan sus árboles de *parsing*, de esta forma es posible aplicar procesos recursivos que permitan trabajar con la expresión. La sintaxis específica considerada debe ser cambiada respecto a la definición revisada anteriormente de modo que se utilicen sólo caracteres ascii. La siguiente gramática especifica la definición.

$$exp \;:=\; c\ (c \in \Sigma) \mid c\ \texttt{\^{}}\ (c \in \Sigma) \mid exp\ \texttt{/}\ exp \mid exp\ \texttt{+}\ exp \mid exp\ \texttt{*}$$

Aquí los símbolos `^`, `/`, `+` y `*` se refieren a su significado literal, tal cual sería ingresado en un teclado. Esta definición tiene la misma interpretación dada anteriormente, pero utiliza `^` para la operación inverso y `*` para la clausura de kleene.

Para evitar ambigüedad en la precedencia de operadores y facilitar la construcción de la representación, se utiliza una versión desambiguada de la gramática que establece de manera explicita la precedencia de los operadores.

$$\begin{aligned} exp &:= concat \mid exp\ \texttt{+}\ concat \\ concat &:= kleene \mid concat\ \texttt{/}\ kleene \\ kleene &:= inv \mid inv\ \texttt{*} \\ inv &:= atom \mid atom\ \texttt{\^{}} \\ atom &:= c\ (c \in \Sigma) \mid \texttt{(}\ exp\ \texttt{)} \end{aligned}$$

Para guardar el árbol de *parsing* se utiliza un tipo de nodo distinto para cada operación (concatenación, alternancia, inverso y estrella de kleene) además de uno adicional para los símbolos.

[6] `https://github.com/nilehmann/k2tdb`
[7] `http://www.boost.org/`
[8] `http://fallabs.com/kyotocabinet/`

La concatenación y alternancia son considerados operadores multiarios y guardan una referencia a dos o más subexpresiones. Por otro lado la estrella de kleene e inverso corresponden a operadores unarios y sólo guardan una referencia a la subexpresión a la que se aplican.

Para guardar la estructura de tipos heterogénea del árbol, se utiliza la biblioteca **`variant`** de boost, la cual permite manipular un conjunto de tipos diversos de manera uniforme[9]. Para soportar una construcción más eficiente, fue necesario realizar una modificación a la clase **`recursive_wrapper<T>`** de modo que ésta aceptara un estado vacío y proveyera un *Move Constructor* con garantías de no excepción[10].

Ejemplo Volviendo a la base de datos de la Figura 2.1 y recordando que $\Sigma = \{\texttt{estilo}, \texttt{integrante}\}$, una posible expresión regular válida sería

```
(integrante/integranteˆ)* + estilo/estiloˆ + integrante
```

La Figura 4.1 muestra el árbol de *parsing* correspondiente a esta expresión.

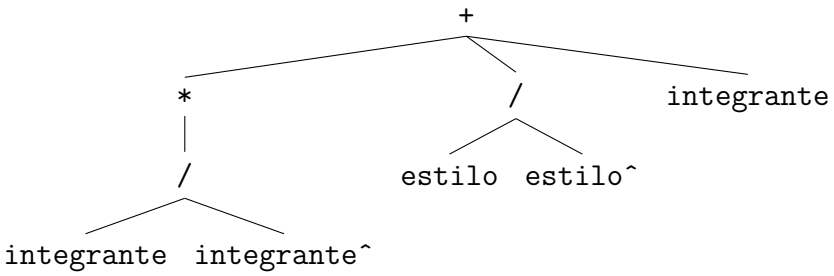


Figura 4.1: Árbol de *parsing* para la expresión `(integrante/integranteˆ)*+estilo/estiloˆ+integrante`

La definición de tipos considera la existencia de expresiones sobre distintos alfabetos, por ejemplo, en secciones posteriores se menciona la utilización de expresiones sobre strings o sobre enteros. Esta definición se hace de manera genérica utilizando *templates* como se muestra a continuación.

```
template<typename T>
using RegExp = boost::variant<
  T,
  boost::recursive_wrapper<concat<T>>,
  boost::recursive_wrapper<alternation<T>>,
  boost::recursive_wrapper<kleene<T>>,
  boost::recursive_wrapper<converse<T>>>;
```

Finalmente, la implementación provee un *parser* de expresiones regulares construido utilizando GNU Bison[11] y GNU Flex[12].

[9] `http://www.boost.org/doc/libs/1_55_0/doc/html/variant.html`

[10] Para una discusión más detallada del problema revisar `http://lists.boost.org/Archives/boost/2013/01/200014.php`

[11] `http://www.gnu.org/software/bison/`

[12] `http://flex.sourceforge.net/`

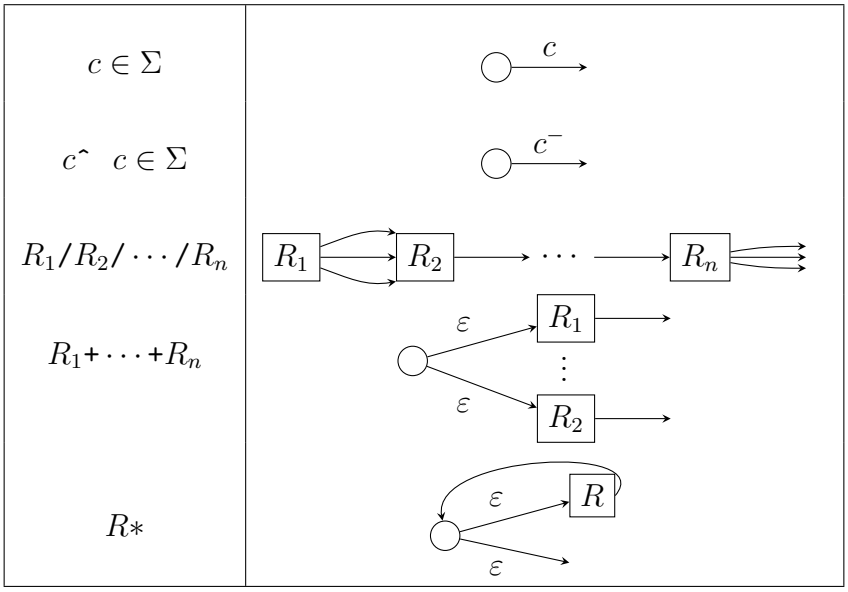


Figura 4.2: Construcción recursiva de un NFA a partir de una expresión regular.

4.2.3. Construcción NFA

Antes de poder evaluar una consulta es necesario construir un autómata finito no determinista que acepte el mismo lenguaje que la expresión regular. En la implementación la construcción se hace de manera recursiva y genera parcialmente autómatas con un estado inicial y varias transiciones de salida que no se conectan con ningún estado. Estas transiciones son conectadas posteriormente con los estados iniciales de los autómatas correspondientes a otras subexpresiones. Finalmente se conectan todas las transiciones de salida de la construcción de la expresión completa con un único estado final. De esta forma se busca minimizar lo más posible la cantidad de estados y transiciones del autómata.

La Figura 4.2 ilustra el proceso recursivo utilizado para construir los autómatas a partir de una expresión. En la figura se muestra el autómata parcial que debe ser construido dependiendo del tipo de la subexpresión.

La funcionalidad para representar y recorrer el autómata está implementada en la clase **`NFA`** cuyo constructor recibe como parámetro una expresión regular sobre enteros e internamente realiza la construcción descrita. El autómata se representa usando una lista de adyacencia donde por cada estado se guarda una lista de pares que contiene los estados vecinos y la etiqueta de la transición.

4.2.4. `K2tdbEngine`

La clase **`K2tdbEngine<K2Tree>`** implementa la representación del grafo utilizando k^2-trees. La estructura interna es sencilla y simplemente guarda un arreglo de k^2-trees, donde cada árbol corresponde a la representación del grafo inducido al considerar aristas de sólo una etiqueta. La clase recibe como parámetro de plantilla el tipo de k^2-tree que se utilizará, realizando una implementación genérica y permitiendo utilizar cualquier variante.

La clase provee dos métodos fundamentales: **`ExistsPath`** y **`Compute`**. El método **`ExistsPath`**

recibe como parámetros dos enteros identificadores de nodos y una expresión regular sobre enteros identificadores de aristas. El método ejecuta el Algoritmo 1 para determinar si los dos nodos identificados en los parámetros están unidos por un camino que cumple la expresión regular especificada. Por otra parte, el método **Compute** recibe un entero identificador de nodo y una expresión regular, e implementa el Algoritmo 2 para encontrar todos los nodos conectados por un camino que cumple la expresión regular.

La implementación es en ambos casos bastante directa, pero requiere algunas consideraciones. En primer lugar la cola es implementada con la clase **queue** provista por la STL[13], utilizando los parámetros por defecto. Adicionalmente el conjunto que guarda los estados ya visitados fue implementado con la clase **unordered_set** provista también por la STL[14]. Esta clase implementa una tabla de hash con encadenamiento separado y la función de hash utilizada está construida combinando los enteros que representan el nodo en el grafo y el estado en el NFA a partir de la función **hash_combine** provista por Boost[15].

4.2.5. Diccionario de Codificación

Si bien los procesos de codificación de identificadores no son evaluados para determinar el desempeño de la implementación, se desarrolló esta funcionalidad para permitir usar la aplicación con mayor facilidad.

La implementación del diccionario utiliza una estrategia sencilla que ataca el problema con una tabla de hash en memoria secundaria. Específicamente se utilizó la clase **HashDB** provista por la biblioteca Kyoto Cabinet. El diccionario utiliza una tabla de hash para guardar la codificación de cada string identificador, además de la tabla inversa para poder convertir los números enteros codificados al string original.

Esta funcionalidad se encuentra en la clase **DictionaryEncoding** la cual provee tres métodos fundamentales. El método **Encode** recibe un string como parámetro y devuelve un entero correspondiente a la codificación de éste. Por otro lado el método **Decode** recibe un entero y retorna el string que corresponde a esa codificación. Finalmente el método **Add** agrega un string al diccionario asignándole como codificación el siguiente entero disponible.

4.2.6. GDBEngine

La clase **GDBEngine** abstrae la codificación de los identificadores y la implementación interna de los grafos, entregando una manera limpia de interactuar con la base de datos. Al igual que **K2tdbEngine**, esta clase provee los métodos **ExistsPath** y **Compute** pero con la diferencia que los identificadores que recibe corresponden a los strings originales correspondientes a los nodos. Adicionalmente la expresión regular está formada sobre los strings originales que identifican las etiquetas. Internamente se realiza la codificación y decodificación para poder interactuar con la clase **K2tdbEngine**.

[13] http://en.cppreference.com/w/cpp/container/queue
[14] http://en.cppreference.com/w/cpp/container/unordered_set
[15] http://www.boost.org/doc/libs/1_35_0/doc/html/boost/hash_combine_id241013.html

4.2.7. Construcción

Este trabajo provee además dos formas de construir una base de datos a partir de un archivo que contiene la información. Los formatos soportados son ntriples y graphdata.

Ntriples es una formato de archivo que permite especificar datos en RDF de manera simple[16]. En la construcción se trabaja con cada identificador simplemente como una cadena de caracteres sin hacer consideraciones especiales sobre *blank nodes* o literales. Esta construcción se encuentra implementada en la clase `NtLoader`.

Graphdata es un formato no estándar utilizado en *GDBenchmark*[17]. En este se permite especificar grafos que representan redes sociales. La funcionalidad para construir grafos desde graphdata esta implementada en la clase `GDLoader`.

4.2.8. Interfaz interactiva

Adicionalmente se desarrolló una aplicación que permite mediante una línea de comandos realizar consultas interactivas a una base de datos. La sintaxis de las operaciones que se pueden realizar se detalla a continuación.

$$
\begin{aligned}
command &:= query1 \mid query2 \mid assign \\
assign &:= var \texttt{ = } id \\
query1 &:= val \texttt{ [} exp \texttt{] } val \\
query2 &:= val \texttt{ [} exp \texttt{] } \\
val &:= id \mid \texttt{:}var
\end{aligned}
$$

Aquí *exp* se refiere a una expresión regular con la sintaxis descrita anteriormente en este capítulo, *id* corresponde a algún identificador de nodo y *var* a un nombre de variable. Como puede esperarse *query*1 representa las consultas decisionales, mientras que *query*2 se refiere a las consultas de computación.

La Figura 4.3 muestra un ejemplo de una sesión de la interfaz de linea de comandos. En ésta hay cargada una base de datos "tipo RDF" del grafo de la Figura 2.1. En la figura puede apreciarse que los dos primeros comandos son definiciones de variables, mientras que el tercero es una consulta decisional que pregunta si *Keith Emerson* y *Greg Lake* tocaron alguna vez en la misma banda. Notar que para hacer uso de las variables se debe anteponer el símbolo `:` de forma que el identificador sea reconocido como tal. Las siguientes dos consultas son de computación e imprimen todos los resultados de manera consecutiva.

[16] `http://www.w3.org/TR/n-triples/`
[17] `http://campuscurico.utalca.cl/~rangles/gdbench/`

Figura 4.3: Ejemplo de interfaz interactiva.

Capítulo 5

Evaluación Experimental

En este capítulo se presenta un análisis experimental de la implementación, mostrando los tiempos de evaluación para distintas consultas. Los resultados son comparados con el desempeño de otras 2 implementaciones: *Sparksee* y *Neo4j*. Todas las pruebas fueron evaluadas en un computador con las siguientes características:

- Procesador Intel® Core™ i7-2600K@3.40GHz (4 cores - 2 threads/core)
- Memoria RAM 8GB DDR3@1300MHz
- Disco Duro WD® 500 GB - 7200 RPM
- Sistema operativo ArchLinux (kernel 3.16.1).

La comparación se restringe sólo a *Sparksee* y *Neo4j*, pues estudios preliminares con grafos de aproximadamente 1M de nodos permitieron determinar que tanto *Jena* como *Virtuoso* no pueden evaluar las consultas de mayor complejidad consideradas en este trabajo. Ambas implementaciones agotan rápidamente la memoria aún siendo ejecutadas sin restricciones en el tamaño del heap. Este comportamiento puede atribuirse a que las implementaciones de *property paths* están construidas de tal forma que la evaluación de las consultas cuenta la cantidad de caminos. Es sabido que esta estrategia produce consultas que son imposibles de tratar [4]. Esta semántica va en contra de la última especificación de SPARQL y sugiere un problema de fondo en las alternativas.

Para el caso de *Sparksee* y *Neo4j* es importante mencionar que ambas alternativas son incomparables en términos de compresión con *k2tdb*, pues las dos soluciones son dinámicas y en disco, no obstante la comparación es presentada aquí para contrastar nuestra alternativa con opciones reales que se usen en la práctica.

5.1. Creación de las Bases de Datos

La evaluación del sistema siguió un esquema similar al utilizado en el trabajo de Angles, Prat y Domínguez [3], en el cual proponen un *benchmark* para la evaluación de bases de datos de grafos basándose en redes sociales. La propuesta provee un generador de grafos sintéticos con características

similares a la de las redes sociales y que entrega además un conjunto de consultas de bajo nivel para su posterior evaluación[1]. La implementación fue levemente modificada para satisfacer de mejor manera las necesidades de este trabajo.

Los grafos utilizados siguen un esquema simple que considera dos entidades: personas y páginas web. La Figura 5.1 muestra el esquema utilizado, donde puede apreciarse que las personas están relacionadas entre si por la arista bidireccional `friend` y con las páginas web por el arco `like`. Adicionalmente cada persona pose como atributos un pid (identificador de persona), un nombre y otros dos campos opcionales. Por otro lado las páginas web poseen un wpid (identificador de página web), una URL y una fecha de creación opcional.

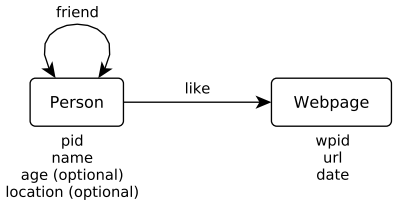


Figura 5.1: Esquema de los grafos de redes sociales.

Para evaluar el desempeño en distintos escenarios de consumo de memoria, se construyeron grafos del orden de 10M, 20M, 30M y 40M de nodos. Posteriormente estos fueron cargados en cada una de las alternativas. En la Tabla 5.1 se muestra la memoria ocupada por las representaciones en cada caso, además de la cantidad de aristas correspondientes a los grafos. A continuación se muestra la experiencia observada en cada caso.

k2tdb

Para nuestra implementación los datos fueron cargados utilizando la funcionalidad provista en la clase `GDLoader` y considerando el tamaño de los datos la inserción fue razonablemente rápida, tardando al rededor de 2 hrs para el grafo más grande.

La implementación alcanzó una representación utilizando muy poco espacio incluso considerando el tamaño ocupado por el diccionario. Para *k2tdb* la Tabla 5.1 muestra tanto el espacio utilizado solo por la representación del grafo como también el tamaño total considerando el diccionario. Puede observarse que el diccionario ocupa aproximadamente un 95 % del espacio total, lo que indica que hay que poner particular atención en esto para generar una alternativa escalable.

Por otro lado la representación del grafo puede extenderse a grafos más grandes teniendo en consideración la memoria necesaria para poder construirlos y que los bitmaps de la representación no alcancen largos demasiado grandes. Ambos problemas pueden solucionarse utilizando la variante que particiona el primer nivel.

[1] `http://campuscurico.utalca.cl/~rangles/gdbench/`

Sparksee

En este caso la construcción de los grafos utilizó la implementación provista por *GDBenchmark*[2] y tardó considerablemente más que la construcción de k2tdb, demorándose al rededor de 5hrs para el grafo de 30M de nodos. Esta implementación utiliza la capacidad de *Sparksee* para tener atributos en los nodos y guarda los nombres de las personas de esta forma. Por otro lado no fue posible cargar el grafo de 40M pues excedía el límite de mil millones de objetos impuesto por la implementación. Finalmente la representación alcanzó tamaños aceptables, pero bastante mayores que los conseguidos con *k2tdb*.

Neo4j

Para *Neo4j* también se utilizó la implementación de *GDBenchmark*[3] y en este caso el tiempo de inserción resultó ser muy grande, demorándose al rededor de 2 días para el grafo de 30M de nodos. El grafo de 40M tampoco fue posible de construir. Finalmente la representación en este caso alcanzó tamaños excesivos.

Nodos	Aristas	k2tdb	k2tdb+dic	Sparksee	Neo4j
10M	156 176 931	81 MB	1.9 GB	7.8 GB	11 GB
20M	326 216 807	168 MB	3.3 GB	17 GB	95 GB
30M	501 489 165	286 MB	5.8 GB	25 GB	148 GB
40M	680 159 502	364 MB	7.5 GB	-	-

Tabla 5.1: Espacio ocupado por cada representación.

5.2. Consultas de Prueba

Siguiendo la propuesta de Angles et al. la evaluación realiza las mismas consultas de navegación. La Tabla 5.2 muestra la descripción de éstas junto a la consulta asociada. Las primeras cuatro formulan consultas sencillas que sólo requieren navegar un paso en el grafo obteniendo los vecinos directos o inversos de un nodo. Las siguientes tres consultas tienen una mayor complejidad pero siguen definiendo caminos de largo fijo. Finalmente la última consulta es de particular interés pues define un camino de largo arbitrario.

El generador utilizado produce además de un grafo, casos de prueba para ser usados en la experimentación. El programa fue ligeramente modificado respecto a la versión de *GDBenchmark* para producir 4 archivos en vez de un sólo XML. Los archivos contienen respectivamente una lista de ID's de personas (usado en las consultas Q3, Q4, Q5, Q6, Q7), una lista de nombres de personas (usado en la consulta Q1), una lista de ID's de páginas (usado en la consulta Q2) y una lista de pares de ID's (id1,id2) tal que la persona id1 está conectada por un camino de amistad con la persona id2 (usado para la consulta Q8).

Para cada consulta se presenta la ejecución de 10000 instancias y se reporta el tiempo promedio de tres ejecuciones consecutivas. Para capturar todos los beneficios de las implementaciones y simular

[2] `http://campuscurico.utalca.cl/~rangles/gdbench/GDBenchDex.zip`
[3] `http://campuscurico.utalca.cl/~rangles/gdbench/GDBenchNeo4j.zip`

lo más posible un escenario real, en cada caso se realizó un *calentamiento* previo donde se ejecutan las mismas consultas. Finalmente se muestran también algunos resultados de ejecuciones en *frío*.

Q	Descripción	Consulta
1	Obtener todas las personas con nombre N.	`N[name⁻]`
2	Obtener todas las personas a las que les gusta una página W.	`W[like⁻]`
3	Obtener las páginas que le gustan a una persona P.	`P[like]`
4	Obtener el nombre de una persona P.	`P[name]`
5	Obtener los amigos de los amigos de una persona P.	`P[friend/friend]`
6	Obtener las páginas que le gustan a los amigos de una persona P.	`P[friend/like]`
7	Obtener las personas que les gusta alguna página que le gusta a una persona P.	`P[like/like⁻]`
8	¿Existe un camino de amistad que conecta las personas P1 y P2?	`P1[friend*]P2`

Tabla 5.2: Consultas de evaluación

5.3. Resultados Obtenidos

La Figura 5.2 muestra los tiempos promedios de ejecución de cada consulta, es decir, el tiempo total dividido 10000. Lo primero que salta la vista es el notorio bajo desempeño de *Neo4j*. En todas las consultas los resultados son varios ordenes de magnitud mayores que en los otros dos casos e incluso, debido a limitaciones de memoria, fue imposible realizar las consultas Q6, Q7 y Q8. Además, a pesar de lograrse construir el grafo de 30M de nodos, fue imposible realizar las consultas para este tamaño.

Para las otras dos implementaciones las diferencias son más complejas y cada alternativa obtiene un rendimiento superior en algunos casos. En primer lugar puede observarse que en las consultas Q1 y Q4 *Sparksee* domina por sobre *k2tdb*, esto se debe a que ambas involucran la resolución de nombres de personas, que en el caso de *Sparksee* son almacenadas como atributos de nodos, por lo cual el problema puede atacarse con técnicas conocidas. Este hecho también explica por qué en ambas consultas el tiempo de ejecución en *Sparksee* se mantiene prácticamente constante para todos los tamaños.

Las consultas Q2 y Q3 comprometen solo aristas de relación entre personas y páginas web, que en este caso forman un conjunto pequeño del total de arcos (sólo un 20 %), y por lo tanto se esclarece por qué *k2tdb* obtiene tiempos de ejecución menores, ya que los k^2-trees están diseñados para funcionar bien en grafos dispersos.

Ahora bien, las consultas Q5 y Q7 representan caminos mas complejos y en *Sparksee* fueron implementados utilizando la API que provee. En estas consultas *k2tdb* obtiene rendimientos superiores y se deja notar la importancia de tener un buen lenguaje de consultas para que un usuario no deba preocuparse de los detalles subyacentes de la implementación.

Otro punto importante es el notorio bajo rendimiento de *k2tdb* en comparación con *Sparksee* en la consulta Q8, este comportamiento puede explicarse por el bajo nivel de optimización de *k2tdb* para mantener en memoria los objetos resultantes de la evaluación de las consultas. Por su parte *Sparksee* debe tener esto en consideración utilizando mejores formas para mantener conjuntos de objetos en memoria.

Sólo en el caso de *k2tdb* fue posible construir el grafo de 40M nodos lo cual evidencia una ventaja en la propuesta. Los resultados muestran que en el rango escogido el crecimiento en los tiempos de evaluación es bajo y que en algunos casos se mantienen prácticamente constante, no obstante el rango de tamaños evaluados es demasiado pequeño para obtener conclusiones significativas.

Ahora bien, en la Figura 5.3 se muestran los resultados para algunas consultas representativas en el escenario donde no se ejecuta un *calentamiento* previo. Es posible notar que en todos los casos con caminos de largo fijo, nuestra propuesta supera por varios ordenes de magnitud el desempeño de *Sparksee* y que los tiempos de este son considerablemente mayores que para el escenario anterior, mientras que *k2tdb* no se ve notoriamente perjudicado. Para el caso de la consulta Q8, nuestra propuesta tiene mejor rendimiento que *Sparksee* pero los valores se encuentran en el mismo orden de magnitud. Estos resultados inducen a pensar que *Sparksee* obtiene su alto rendimiento mediante mecanismos de *caching* y que nuestra propuesta podría mejorar de manera sustancial utilizando estrategias similares.

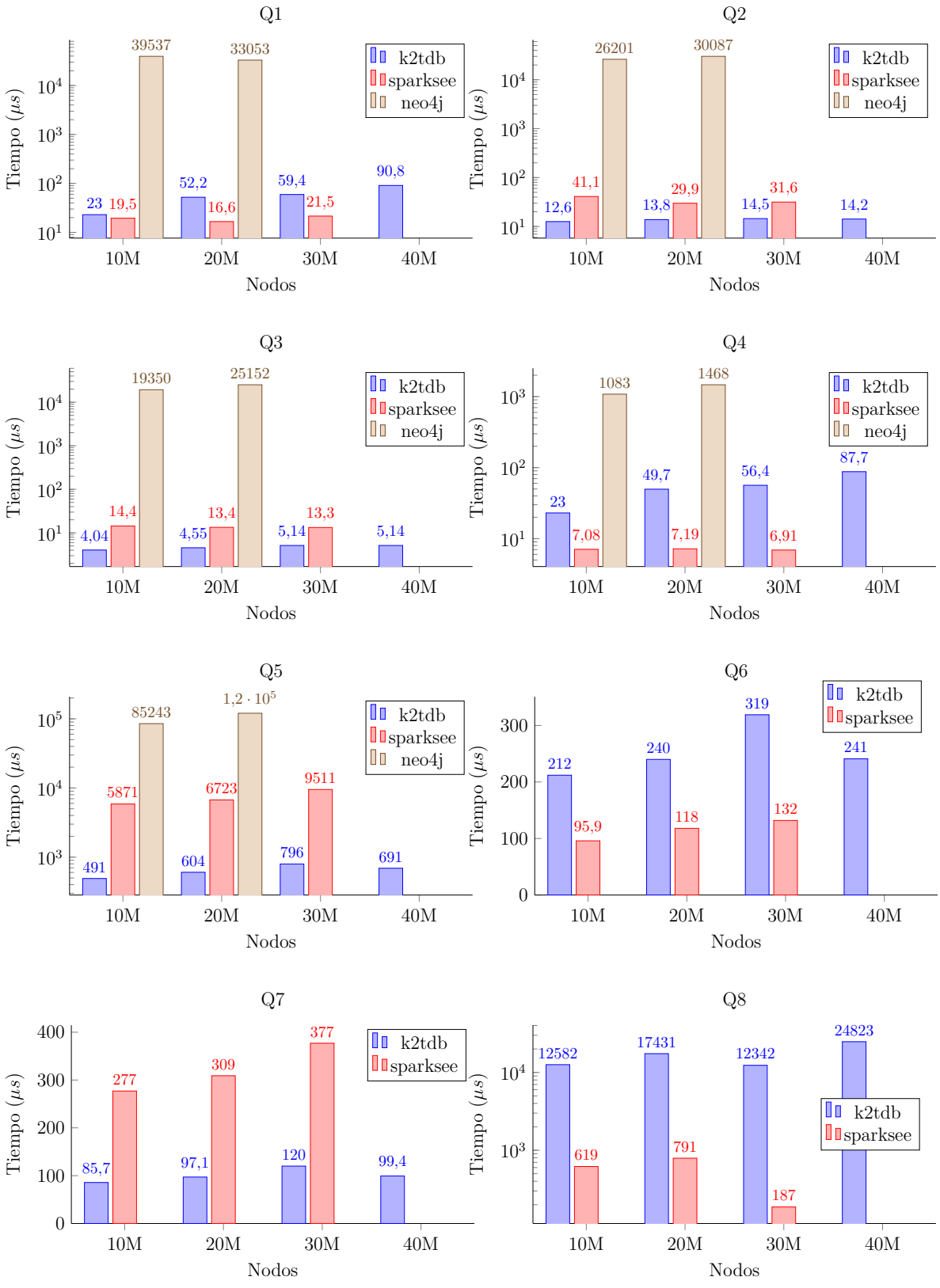


Figura 5.2: Tiempo de respuesta promedio (en microsegundos) para las consultas.

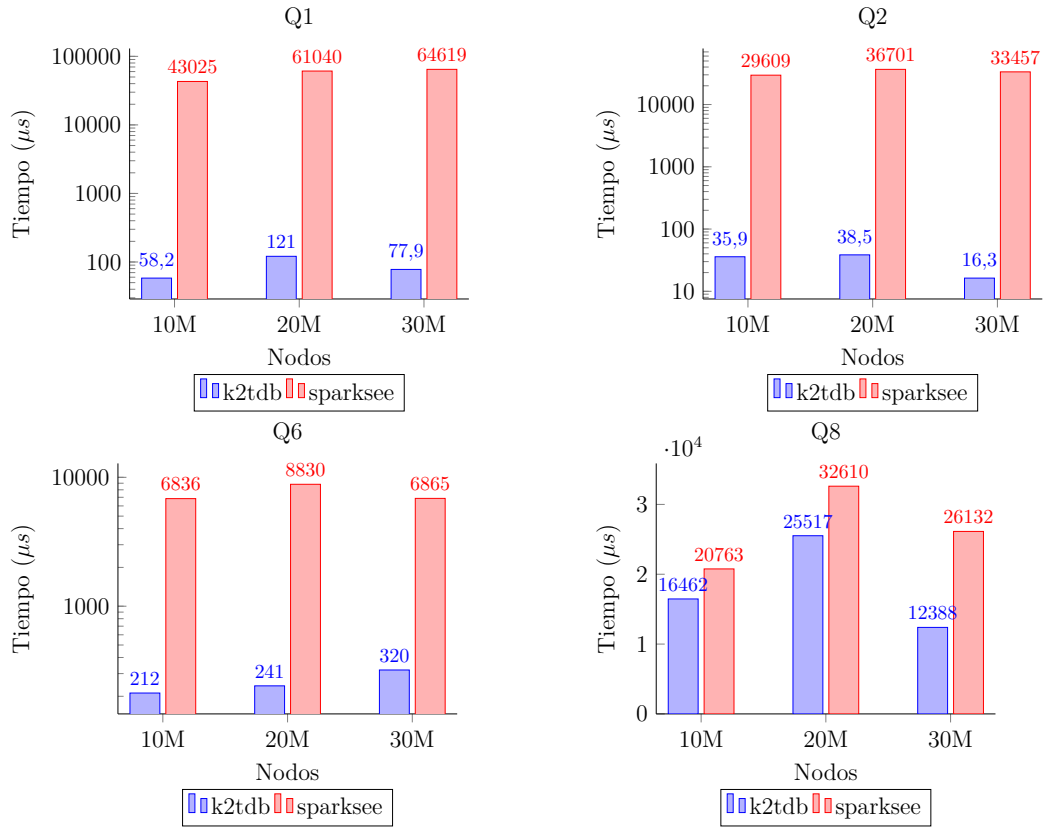


Figura 5.3: Tiempo de respuesta promedio para escenario en frío.

5.4. Bases de Datos de Grafos Reales

Es claro que la eficiencia de los k^2-trees, y por consecuencia de *k2tdb*, depende en gran medida de las características y forma de los grafos involucrados. En las secciones anteriores se presentó un análisis experimental sobre una serie de grafos sintéticos y por lo tanto el buen comportamiento de *k2tdb* pudo estar determinado por la existencia de *clusters* introducidos de manera artificial por el uso de grafos sintéticos. Para dilucidar el comportamiento en un escenario real, en la presente sección se presenta un análisis sobre bases de datos de grafos reales, haciendo consultas similares a las introducidas en las de las secciones anteriores.

LinkedMDB

LinkedMDB[4] es un proyecto que tiene como objetivo publicar la primera base de datos sobre películas para la Web Semántica. *LinkedMDB* está distribuido como una base de datos RDF y el dump utilizado posee al rededor de 2M objetos distribuidos en 6M de triples[5].

Las consultas evaluadas en esta experimentación consideran los objetos que representan actores y películas, además de las relaciones entre estos. La Tabla 5.3 muestra las consultas realizadas

[4] `http://http://linkedmdb.org/`
[5] `http://queens.db.toronto.edu/~oktie/linkedmdb/`

Q	Descripción	Consulta
1	Obtener películas en las que ha actuado un actor A.	`A[actor⁻]`
2	Obtener actores que han actuado en la misma película que un actor A ("coactores").	`A[actor⁻/actor]`
3	¿Existe un camino de "coactores" entre A1 y A2?	`A1[(actor⁻/actor)*)]A2`

Tabla 5.3: Consultas para *LinkedMDB*

simplificando la notación para la etiqueta que representa la relación de haber actuado. Con esta simplificación un triple `(M,actor,A)` significa que el actor `A` actuó en la película `M`.

DBLP

DBLP es una extensa base de datos con información bibliográfica de publicaciones en ciencias de la computación. En su versión RDF la base de datos cuenta con alrededor de 30M de objetos y 70M de triples[6].

De manera similar que en el caso de *LinkedMDB* las consultas realizadas en *DBLP* se basan en la relación de coautoría existente entre personas. En este caso un triple de la forma `(P, creator, A)` significa que el autor `A` participó en la creación de la publicación `P`. La tabla 5.4 muestra en detalle las consultas involucradas.

Q	Descripción	Consulta
1	Obtener publicaciones del autor A.	`A[creator⁻]`
2	Obtener coautores de un autor A.	`A[creator⁻/creator]`
3	¿Existe un camino de coautoría entre A1 y A2?	`A1[(creator⁻/creator)*)]A2`

Tabla 5.4: Consultas para *DBLP*

5.4.1. Construcción

Dado que lo *dumps* de las bases de datos venían en el formato ntriples, la construcción se hizo con la funcionalidad provista en la clase `NTLoader`. En ambos casos la construcción de las bases de datos tardó un par de horas al igual que en el caso de los grafos sintéticos. La Tabla 5.5 muestra un resumen con la información de la construcción de las bases de datos.

	Nodos	Triples	k2tdb	k2tdb+dic
LinkedMDB	6 147 978	8.4MB	453 MB	
DBLP	70 267 668	137MB	7.9 GB	

Tabla 5.5: Espacio ocupado por cada representación.

[6] `http://dblp.l3s.de/dblp++.php`

5.4.2. Resultados

Al revisar la Tabla 5.5, lo primero que puede notarse es que el nivel de compresión en este caso es bastante menor que para los grafos sintéticos. Para el caso de *DBLP* se obtiene una representación de 16bits/arista en comparación con los grafos sintéticos que alcanzan una representación de 4.5bits/arista. Este comportamiento sugiere la existencia de *clusters* muy marcados en los grafos sintéticos que no se observa en estos casos reales.

En la Figura 5.4 se muestran los resultados para cada consulta. Para *LinkedMDB* se puede observar que tanto la consulta Q1 como la Q2 tienen tiempos de respuesta similares a las consultas análogas en el caso de los grafos sintéticos. Para el caso de *DBLP* las consultas Q1 y Q2 son evaluadas en varios ordenes de magnitud más que en los pares para grafos sintéticos. Finalmente la consulta Q3 posee tiempos de respuestas considerablemente altos en ambos casos.

Estos resultados no solo sugieren un deterioro en el desempeño a causa de las características de los k^2-trees, sino que anuncia problemas intrínsecos en algunos grafos para para evaluar consultas de navegación. El método de evaluación utilizado en *k2tdb* requiere hacer un recorrido sobre el grafo que en el caso de tener pocas componentes conexas puede forzar el recorrido casi completo, lo que para grafos demasiado grandes puede ser prohibitivo.

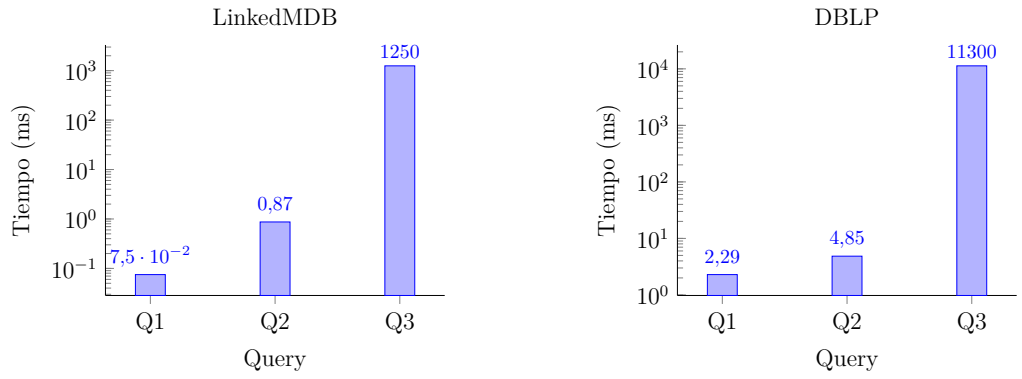


Figura 5.4: Tiempo de respuesta promedio (en milisegundos) para las consultas.

Capítulo 6

Conclusiones y Trabajo Futuro

En los términos más generales puede concluirse del trabajo realizado que la implementación de una base de datos de grafos supone una serie de importantes desafíos que deben ser considerados al momento de su diseño. Pasar por alto estos desafíos puede acarrear resultados desafortunados, tal es el caso de *Neo4j* que a pesar de poseer una rica interfaz, las decisiones en su diseño lo hacen una alternativa inviable para casos que requieren trabajar con grandes volúmenes de información. Por otro lado la utilización de un lenguaje de consultas es enormemente favorable ya que incluso representaciones eficientes de un grafo podrían ser pobremente utilizadas cuando no se tiene el conocimiento a fondo de las estructuras involucradas. Un ejemplo de esto es *Sparksee*, donde se observó que la implementación ingenua de algunas consultas tuvo un peor desempeño del esperado.

Al evaluar la propuesta de este trabajo, se comprobó que ésta presenta un buen rendimiento, comparable incluso con alternativas que poseen un desarrollo avanzado como *Sparksee*. Adicionalmente nuestra propuesta posee algunas ventajas debido a la representación comprimida de las estructuras. En primer lugar el espacio utilizado por la implementación es considerablemente menor que en el caso de *Sparksee*, situación que puede ser beneficiosa en muchos escenarios. Adicionalmente esta misma característica debería permitir expandir el uso de la propuesta para representar grafos aún más grandes.

A pesar de su buen desempeño, el diseño de la propuesta presenta amplias posibilidades de mejora donde un punto importante es la optimización de los conjuntos involucrados en la evaluación de las consultas. Por otro lado un avance interesante sería la incorporación de mecanismos de *caching* para mejorar el desempeño de la implementación.

El estudio realizado permite notar adicionalmente que la elección del modelo de la base de datos de grafos influye también en el rendimiento de las consultas. Un ejemplo es como la incorporación de atributos en los nodos permite obtener algunas consultas de manera más eficiente. La incorporación de esta capacidad permitiría adicionalmente realizar eficientemente consultas que involucren operadores de comparación ($<$, $>$, $!=$) utilizando estrategias ampliamente conocidas.

Por otra parte, es posible concluir que la evaluación de consultas depende de manera considerable de la forma de los grafos involucrados. Esta situación se evidencia en el deterioro del desempeño con la inclusión de grafos reales. Este deterioro no sólo está inducido por las características de las estructuras utilizadas en este trabajo, sino que sugiere un problema en el método de evaluación clásico presentado en la teoría. Esto propone como desafío encontrar mejores estrategias para evaluar

el tipo de consultas dependiendo de las características del grafo.

Lo anterior sugiere también la posibilidad de aprovechar las características de las estructuras de datos sucintas y utilizar las operaciones que permiten resolver de manera eficiente, para diseñar un algoritmo de evaluación que presente un mejor desempeño. Un mejor uso de las estructuras podría permitir la inclusión de nuevas operaciones al lenguaje o la generación de planes de consultas que ayuden a optimizar próximas consultas.

Finalmente otras contribuciones interesante para la continuación de este trabajo son la incorporación de un mejor diccionario y la extensión del lenguaje de consultas. Por ejemplo, es posible extender la propuesta para soportar *nested regular expressions* [16] o incluso avanzar en una implementación completa de SPARQL1.1 siguiendo los trabajos que exploran el uso de k^2-trees para evaluar consultas de *pattern matching* [1].

Bibliografía

[1] Sandra Álvarez-García, Nieves R. Brisaboa, Javier D. Fernández, Miguel A. Martínez-Prieto, and Gonzalo Navarro. Compressed vertical partitioning for full-in-memory rdf management. *CoRR*, abs/1310.4954, 2013.

[2] Renzo Angles and Claudio Gutiérrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1), 2008.

[3] Renzo Angles, Arnau Prat-Pérez, David Dominguez-Sal, and Josep-Lluis Larriba-Pey. Benchmarking database systems for social network applications. In *GRADES*, page 15, 2013.

[4] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. Counting beyond a yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012, Lyon, France, April 16-20, 2012*, pages 629–638, 2012.

[5] Pablo Barceló, Leonid Libkin, Anthony Widjaja Lin, and Peter T. Wood. Expressive languages for path queries over graph-structured data. *ACM Trans. Database Syst.*, 37(4):31, 2012.

[6] Pablo Barceló, Jorge Pérez, and Juan L. Reutter. Relative expressiveness of nested regular expressions. In *Proceedings of the 6th Alberto Mendelzon International Workshop on Foundations of Data Management, Ouro Preto, Brazil, June 27-30, 2012*, pages 180–195, 2012.

[7] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible markup language (xml). *World Wide Web Journal*, 2(4):27–66, 1997.

[8] N. Brisaboa, S. Ladra, and G. Navarro. DACs: Bringing direct access to variable-length codes. *Information Processing and Management (IPM)*, 49(1):392–404, 2013.

[9] Nieves R. Brisaboa, Rodrigo Cánovas, Francisco Claude, Miguel A. Martínez-Prieto, and Gonzalo Navarro. Compressed string dictionaries. In *Experimental Algorithms - 10th International Symposium, SEA 2011, Kolimpari, Chania, Crete, Greece, May 5-7, 2011. Proceedings*, pages 136–147, 2011.

[10] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. Compact representation of web graphs with extended functionality. *Inf. Syst.*, 39:152–174, 2014.

[11] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[12] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, Greece, 2005. CTI Press and Ellinika Grammata.

[13] M Graves, E.R. bergeman, and C.B. lawrence. Graph databse systems. *entineering in Medicine and Biology Magazine*, 14(6):737–45, 1995.

[14] John E. Hopcroft and Jeffrey D. Ullman. *Introducction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[15] Gabriel M. Kuper and Moshe Y. Vardi. The logical data model. *ACM Trans. Database Syst.*, 18(3):379–413, 1993.

[16] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. nsparql: A navigational language for rdf. *J. Web Sem.*, 8(4):255–270, 2010.

[17] Gonzalo Ríos. Diseño e implementación de un lenguaje de consulta para bases de datos de grafos. Master's thesis, Universidad de Chile, 2013.

[18] N Trinajstiac. *Chemical Graph Therory.* CRC press, 1983.

[19] Peter T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, 2012.