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

DISEÑO E IMPLEMENTACIÓN DE UN LENGUAJE DE CONSULTA PARA BASES DE DATOS DE GRAFOS

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS MENCIÓN COMPUTACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL MATEMÁTICO

GONZALO ANDRÉS RÍOS DÍAZ

PROFESOR GUÍA:
PABLO BARCELÓ BAEZA

MIEMBROS DE LA COMISIÓN:
GONZALO NAVARRO BADINO
JORGE PÉREZ ROJAZ
MARCOS KIWI KRAUSKOPF

SANTIAGO DE CHILE
JULIO 2013

Dedicado a mis padres, mis hermanos y en especial a mi Señora,
por apoyarme en los momentos más difíciles.

Resumen

Las bases de datos de grafos son un modelo que ha ganado terreno en los últimos años, dada la necesidad de modelar situaciones complejas en donde el modelo relacional no es suficiente. En este trabajo introducimos el tema haciendo una revisión del estado del arte de las bases de datos de grafos, explicando algunas de sus aplicaciones reales, definiendo algunos de los diferentes modelos teóricos y analizando las implementaciones más importantes que existen en la realidad.

De nuestro análisis concluimos que la gran falencia en el tema es la ausencia de un lenguaje de consulta formal, con una sintaxis y semántica clara, y que tenga un buen equilibrio entre expresividad y complejidad. Nuestra propuesta para solucionar este problema es utilizar *Converse-PDL* como lenguaje de consulta de bases de datos de grafos, definiéndolo formalmente, y demostrando que su complejidad teórica es óptima. Además, mostramos que la expresividad de este lenguaje es suficiente para una gran cantidad de aplicaciones.

Una vez definido nuestro lenguaje, procedemos a diseñar una implementación eficiente, definiendo los algoritmos y las estructuras de datos necesarias, cuidando de cumplir todas las restricciones que están presentes en nuestro modelo computacional. Luego, procedemos a realizar la implementación en sí, describiendo en detalle las representaciones internas de los distintos elementos, respaldando con resultados experimentales las decisiones tomadas. Además, explicamos las distintas mejoras y optimizaciones que realizamos, con el fin de obtener la mayor eficiencia posible. Una vez terminada la implementación, procedemos a explicar todos los archivos programados y la interfaz de usuario implementada, con el fin de facilitar los futuros desarrollos.

Con el fin de validar nuestra implementación, procedemos a diseñar un experimento para evaluar cuantitativamente el desempeño de nuestra implementación. El experimento diseñado se ejecuta en nuestra implementación, así como en otras tres implementaciones existentes, escogiendo los proyectos más competitivos, para así realizar una comparación objetiva. Finalizamos nuestro trabajo con las conclusiones obtenidas a partir de los experimentos, destacando los aspectos más importantes de nuestra implementación, y exponiendo algunas ideas a desarrollar en el futuro.

Agradecimientos

Quisiera agradecer a mi profesor guía, Pablo Barceló, por ayudarme y aconsejarme durante el largo proceso que fue desarrollar este trabajo. Además quisiera agradecer a Claudio Gutierrez, Jorge Pérez y Renzo Angles, por todo el tiempo invertido en reuniones y conversaciones, las cuales me permitieron encontrar el enfoque práctico que se le dió a este trabajo.

Tabla de Contenido

Capítulo 1

Introducción

1.1. Motivación

Desde la década de los *70's*, el modelo de base de datos más empleado en el mundo ha sido el modelo relacional [14], con el lenguaje de consulta estructurado [12] o *SQL* (por sus siglas en inglés *Structured Query Language*), por su gran versatilidad, potencia y por los formalismos matemáticos sobre los que se basa.

Las primeras nociones de lo que hoy conocemos como base de datos de grafos emergieron en la década de los *80's* [29], con el objetivo de modelar información cuya estructura es un grafo [9]. Después de un par de años, la cantidad de trabajos publicados referentes a las bases de datos de grafos fue disminuyendo considerablemente hasta que a mediados de la década de los *90's* cesó por completo. La razón del declive del área fue la aparición de los datos semiestructurados; la emergencia de *XML* [10] capturó toda la atención en las estructuras de datos de árbol, por el hecho que este modelo es suficiente para una gran cantidad de aplicaciones.

Pero dado que el mundo ha cambiado, y el surgimiento de procesar grandes flujos de información para millones de usuarios en tiempo real ha colapsado al modelo relacional, las grandes empresas como Google, Facebook y Amazon han dejado de lado este viejo modelo en busca de modelos que se adapten mejor a sus necesidades. Esto no significa que el modelo relacional esté obsoleto, pero ya no es útil para determinados proyectos, y es así como han surgido las tecnologías *NoSQL* [36] (comunmente denotadas "no sólo SQL").

A lo largo de los años se han desarrollado diversos modelos, tales como las bases de datos documentales [41], orientadas a objetos [7], tabulares [23], de clave-valor [17] y en especial las bases de datos de grafos [3], las cuales son el tema principal de este trabajo. Este tipo de base de datos está diseñado para almacenar grandes cantidades de datos, fuertemente relacionados entre sí, que es natural representarlos en un grafo, tales como redes sociales y redes viales.

1.2. Contribuciones

En el presente trabajo nos concentraremos en los modelos de bases de datos de grafos, revisando los aspectos tanto teóricos como prácticos. Las principales contribuciones de nuestro

trabajo serán las siguientes:

1. Dar a conocer el estado del arte de las bases de datos de grafos, tales como los modelos principales, sus aplicaciones en la vida real y las implementaciones más importantes.

2. Analizar las implementaciones existentes, revisando sus características y dar a conocer los principales problemas existentes.

3. Proponer un lenguaje de consultas para bases de datos de grafos, con una sintaxis y una semántica clara, manteniendo un buen equilibrio entre su poder expresivo y su complejidad teórica.

4. Diseñar e implementar un sistema de gestión de bases de datos de grafos, utilizando algoritmos y estructuras de datos ad hoc, que satisfagan las restricciones inherentes de nuestro modelo.

5. Comparar nuestra implementación con otros proyectos activos de bases de datos en grafos, evaluando principalmente la expresividad y la eficiencia de los lenguajes de consultas.

6. Finalmente, nuestra intención es publicar nuestro trabajo como un proyecto *Open-Source*, liberando el código fuente de la implementación, con el fin de proveer a científicos e investigadores de una herramienta útil para sus investigaciones y experimentos.

Como veremos a lo largo de este trabajo, los actuales sistemas de gestión de bases de datos de grafos presentan una gran cantidad de carencias, donde la más grave es la ausencia de un lenguaje de consulta bien definido y eficiente, elemento fundamental en el área.

En el presente trabajo proponemos una solución a este problema, definiendo formalmente un lenguaje de consulta y una metodología de implementación, la cual cumple con los requisitos de eficiencia teórica, hecho que es respaldado por pruebas experimentales.

1.3. Organización del Documento

El presente documento está organizado de la siguiente manera:

- Capítulo 1, Introducción. Tal como vimos, iniciamos con una motivación al tema de las bases de datos de grafos, para luego listar los objetivos y contribuciones del presente trabajo, finalizando con la organización del documento.

- Capítulo 2, Bases de Datos de Grafos. Realizamos una revisión general de las bases de datos de grafos, describiendo algunas de sus aplicaciones reales, definiendo formalmente el modelo de bases de datos y haciendo un análisis cualitativo de las implementaciones actuales, presentando los problemas más importantes.

- Capítulo 3, Lenguaje de Consulta Propuesto. En este capítulo nos concentramos en el lenguaje de consulta propuesto, empezando con una intuición del lenguaje, siguiendo con las definiciones formales, para terminar con un estudio de su expresividad y complejidad.

- Capítulo 4, Diseño de la Implementación. El objetivo de este capítulo es diseñar todos los elementos necesarios para llevar a cabo la implementación del lenguaje de consulta propuesto. Para esto, definimos los algoritmos utilizados en la evaluación de consultas, las estructuras de datos para el almacenamiento, y la especificación de los objetos principales de nuestra implementación.

- Capítulo 5, Implementación del Sistema. Aquí es donde explicamos los aspéctos técnicos más importantes de nuestra implementación, con el fin de lograr la mayor eficiencia posible. El primer caso es el de los conjuntos de nodos, explicando cuál es la forma más eficiente de representación, y cómo realizar las operaciones entre conjuntos. Luego explicamos en detalle las técnicas utilizadas para consultar el grafo lo más eficientemente posible, considerando varias optimizaciones implementadas. Finalmente se detalla la evaluación de cada tipo de consulta, utilizando todos los elementos explicados anteriormente.

- Capítulo 6, Descripción de la Implementación. En este capítulo hacemos una revisión de la implementación en Java, listando y explicando cada uno de los archivos, para finalizar con ejemplos de uso de la interfaz de usuario implementada.

- Capítulo 7, Evaluación de la Implementación. En este capítulo diseñamos un experimento con el fin de evaluar la eficiencia de nuestra implementación al momento de evaluar consultas de distintas naturalezas. Este experimento se ejecuta en otras implementaciones disponibles, con el fin de realizar una comparación experimental de nuestra implementación.

- Capítulo 8, Conclusiones y Trabajo Futuro. En este capítulo realizamos un análisis de nuestro trabajo, destacando nuestras principales contribuciones, y exponiendo posibles mejores a desarrollar en el futuro.

Capítulo 2

Bases de Datos de Grafos

Una base de datos de grafos puede ser caracterizada como una estructura de datos, que es modelada como un grafo, en donde los nodos representan los datos, y los arcos representan relaciones entre pares de nodos. Como es natural tener más de un tipo de relación, se procede a etiquetar los árcos con el nombre de la relación. La Figura 2.1 nos entrega un ejemplo de una base de datos de grafo que almacena la información de una pequeña red social.

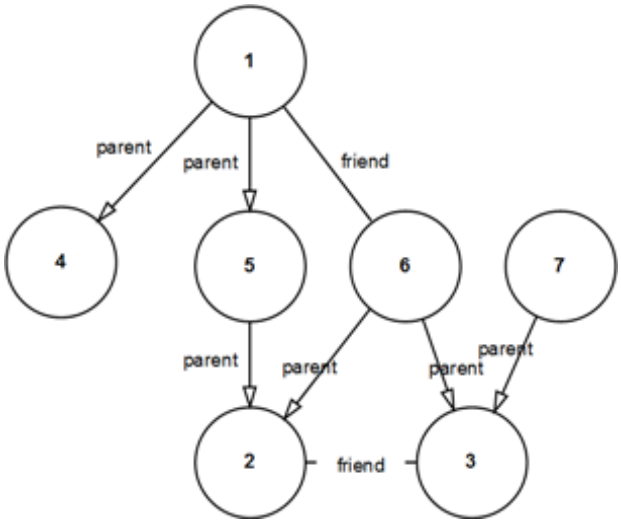

Figura 2.1: Ejemplo de Base de Datos de Grafo

En el ejemplo, los nodos están enumerados del 1 al 7, que representa un identificador único para los nodos, mientras que los árcos están etiquetados, dependiendo si representan la relación *parent* o *friend*. La semántica del grafo está dada por la etiqueta de los árcos, ya que el arco *(1,parent,5)* se interpreta como "1 es padre de 5". Este paradigma nos permite modelar muchas situaciones complejas que no son descritas de forma natural con bases de datos relacionales, principalmente por la naturaleza de los datos.

Un aspecto importante en su uso, es que la gran mayoría de las aplicaciones reales manejan un gigantesco volumen de datos, alcanzando los miles de millones de datos. Ejemplo de ello, algunas empresas deben proveer servicios las 24 horas días, los 365 días del año y a todo

el mundo, como son los casos de *Google, Facebook* y *Twitter,* que procesan más de 100 millones de consultas diarias. Otro caso importante es el de *World Data Centre for Climate* [47] (traducido como el Centro Mundial de Datos para el Clima), donde su base de datos almacena todas las mediciones históricas del clima en el mundo, y que en el año 2012 alcanzó el tamaño de 652 TeraBytes. A continuación listamos algunas de las aplicaciones reales más comunes en el uso de las bases de datos de grafos en la actualidad [11].

2.1. Aplicaciones Reales

Química y Biología

En química los datos son modelados como grafos [44] en donde los átomos se representan como nodos y los enlaces entre ellos se representan como arcos. En biología los datos también son representados por grafos [21] en donde los aminoácidos se representan como nodos, y los enlaces también se representan como arcos.

Estos grafos son utilizados para el análisis y el descubrimiento de fármacos. Los datos tienen muchas etiquetas, por lo que la operación principal es el reconocimiento de patrones, buscando subgrafos frecuentes en los datos. Modelados con bases de datos tradicionales, estas operaciones tomarían mucho más tiempo, debido a la naturaleza recursiva de recorrer un grafo.

Redes Sociales

Las redes sociales son estructuras compuestas por conjuntos de actores (tales como individuos u organizaciones) que están conectados entre sí por lazos interpersonales, que se pueden interpretar como relaciones de amistad, parentesco, gustos, entre otros. Las redes sociales actuales, tales como Facebook y Twitter, están compuestas por grandes cantidades de datos modelados como un gran grafo [46]. Estos grafos representan el conjunto de actores como el conjunto de nodos, mientras que los arcos representan las relaciones o flujos entre los actores. Las operaciones más comunes en grandes grafos sociales son la búsqueda de caminos más cortos y las de agrupamiento (*clustering* en inglés), donde los algoritmos para resolver estas operaciones proveen un análisis de la relación entre dos individuos, o del agrupamiento de individuos en grupos o comunidades dentro de la red social [45].

La Web

La web es esencialmente un grafo de datos e información vinculados entre sí, [28]. Este grafo de datos masivo es utlizado en aplicaciones tales como los buscadores web y los recolectadores de datos. *PageRank*, posiblemente el algoritmo mejor conocido de Google, analiza la web para determinar un *ranking* para las páginas, dependiendo de que tantas páginas están vinculadas a ésta. Otros algoritmos de grafos importantes son la búsqueda de palabras claves (*keyword* en inglés) y *clustering* de documentos.

Redes de Información

Las bases de datos de grafo permiten modelar flujos de información (*information networks* en inglés), tal como es el caso de las los artículos científicos [16], en los cuales los investigadores, las conferencias y los artículos en sí son representados como nodos, mientras que los arcos representan las relaciones tales como las citaciones entre artículos, la publicación de un artículo en una conferencia y la participación de un investigador en un determinado *paper*.

Bases de Conocimiento

Desde el año 2012, Google incluye dentro de su motor de búsqueda una base de conocimiento, llamada *Knowledge Graph* [26], con el fin de obtener mejores resultados y acceder a información más estructurada que archivos planos. Uno de los principales recursos que usa es *Freebase* [8], una base de conocimiento de grafo en línea, colaborativo y libre, permitiendo que cualquier persona con acceso a internet pueda consultar o agregar nuevo *conocimiento* a esta única base. Actualmente, esta base contiene más de 100 millones de afirmaciones sobre distintos tópicos de 4000 tipos diferentes. Un enfoque diferente es el del proyecto *YAGO-NAGA* [27], que busca construir y mantener de forma automática una gran base de conocimiento, extrayendo datos de la web.

Web Semántica

El objetivo de la *Web Semántica* [6], actividad desarrollada principalmente por la *W3C* [48], es que toda la información presente en la *Web* sea comprensible por máquinas, mediante la agregación de información semántica a los recursos web. Mientras que actualmente la *Web* se asemeja a un grafo, donde no existen diferentes tipos de nodos y de arcos, en la *Web Semántica* cada nodo (o recurso) pertenece a algún tipo, identificado con una etiqueta, y los arcos representan relaciones explícitamente diferenciadas. Hoy en día, *RDF* [40] (del inglés *Resource Description Framework*) es el lenguaje estandarizado para la definición de tipos y metadatos en la web, y *SPARQL* [42] (acrónimo recursivo del inglés *SPARQL Protocol And RDF Query Language*) es el lenguaje estandarizado para realizar consultas en los grafos *RDF*.

2.2. Modelos Teóricos

Existen diversas variantes de modelos de base de datos de grafo [3], pero todas siguen la definición matemática básica de un grafo: un conjunto de nodos, que representan las entidades atómicas, y un conjunto de arcos etiquetados, que representan las relaciones entre los datos.

El modelo de base de datos de grafos más simple que podemos considerar está compuesto por un conjunto de nodos y un conjunto de arcos no etiquetados, el cuál sirve para representar una red de influencia entre distintos objetos, pero que en la práctica no es muy utilizado por ser muy limitado. Por otro lado, un modelo de base de datos de *hipernodos* [39] consiste en un grafo dirigido donde sus nodos pueden ser grafos, permitiendo grafos anidados. Este modelo permite modelar objetos complejos de una manera simple, pero en la práctica genera una gran redundancia de datos. Por último, el modelo de base de datos de *hipergrafos* [30] es

una generalización de un grafo, en donde se extiende la definición de arco a *hiperarco*, el cuál relaciona un conjunto arbitrario de nodos. Este modelo permite definir objetos complejos, sin generar redundancia de datos.

El modelo de base de datos de grafo que vamos a considerar en nuestro trabajo es un modelo simple, pero lo suficientemente expresivo para modelar situaciones complejas. Este modelo está compuesto por los dos elementos principales, un conjunto de nodos y un conjunto de arcos etiquetados, destacando que los arcos son dirigidos (es decir, tienen una dirección) y vamos a permitir que los nodos almacenen valores en sus atributos, tales como números, palabras, etc. A continuación está la formalización de nuestra definición.

Definición 2.2.1 *Base de Datos de Grafos*

Sea $\mathcal{V}$ un conjunto infinito numerable de identificadores de nodos (ids), $\mathcal{S}$ un conjunto de palabras sobre algún alfabeto, Σ un alfabeto finito que representa los tipos de relaciones y $\mathcal{A}$ un alfabeto finito que representa los tipos de atributos. Una *Base de Datos de Grafos* (de ahora en adelante *GDB* por su abreviatura en inglés) G sobre Σ es una tupla *(V,E,ρ)* donde:

- V es un conjunto finito de ids (es decir, un subconjunto finito de $\mathcal{V}$), llamado los nodos de *G;*
- $E \subseteq V \times \Sigma \times V$ es el conjunto de arcos etiquetados en Σ*;*
- $\rho : V \times \mathcal{A} \longrightarrow \mathcal{S}$ es la función parcial de asignación que a cada par nodo-atributo le asigna un valor en $\mathcal{S}$.

La interpretación de un arco $(u, c, v) \in E$, donde $u, c \in V$ y $c \in \Sigma$, es que existe un arco con etiqueta c que va desde u hasta v en G. Esto significa que los nodos u, v están relacionados por c. Además, que $\rho(v, \sigma) = k$ sea cierto en G, donde $v \in V$, $\sigma \in \mathcal{A}$ y $k \in \mathcal{S}$, significa que el nodo v tiene asignado el valor k en el atributo σ.

Notación 2.2.1 Nos vamos a permitir el uso de la notación abreviada para la función de asignación, de modo que $\rho(v, \sigma) = k$ lo podemos denotar como $\sigma(v) = k$.

Existen variantes del modelo descrito, tales como permitir que las etiquetas sean nodos, o que los arcos conecten conjuntos de nodos. Sin embargo, el modelo recién definido es lo suficientemente expresivo para la gran mayoría de aplicaciones, y es este modelo el más implementado por los distintos proyectos existentes de bases de datos de grafos, que trataremos a continuación.

2.3. Implementaciones Actuales

En los últimos años ha habido un incremento en la cantidad de implementaciones de bases de datos de grafos, existiendo en la actualidad más de 20 proyectos distintos [20], cada uno con distintas características, pros y contras. Algunos de los componentes que debe proveer una buena implementación de base de datos de grafos [2] son los siguientes:

- Interfaces externas: Una interfaz puede estar diseñada para los usuarios finales, tales como una interfaz gráfica (*GUI* del inglés *Graphical User Interface*) o una interfaz de línea de comandos (*CLI* del inglés *Command Line Interface*), o diseñada para los desarrolladores de aplicaciones, tales como una *API* (del inglés *Application Programming Interface*).
- Lenguajes de bases de datos: Los tipos de lenguajes que pueden proveer los lenguajes de definición, lenguajes de manipulación y/o lenguajes de consulta.
- Optimización de consultas: El tiempo de ejecución de las consultas es un tema clave en todo sistema de base de datos, por lo que debe implementar técnicas avanzadas para la optimización del plan de ejecución.
- Motor de almacenamiento: La forma en que se almacenan los datos en disco, y la forma en que se leen y escriben debe ser de forma correcta y consistente.
- Manejo de transacciones: El sistema debe cumplir las características *ACID*, un acrónimo del inglés *Atomicity, Consistency, Isolation and Durability*, conjunto de características necesarias para que una serie de instrucciones no pueda finalizar en un estado intermedio.

Las implementaciones que satisfacen, en su mayoría, las condiciones anteriores son: *AllegroGraph, DEX, HypergraphDB, InfiniteGraph* y *Neo4J*. A continuación vamos a describir estas implementaciones destacadas, identificando sus principales características y restricciones.

AllegroGraph

AllegroGraph [1] es uno de los precursores de la actual generación de bases de datos de grafo. Aunque fue creada inicialmente como una base de datos de grafo, actualmente su desarrollo está orientado a los estándares de la Web Semántica, es decir RDF/S [40], SPARQL [42] y OWL [38]. Además, *AllegroGraph* provee características para análisis de redes sociales y para datos geotemporales. El producto tiene una licencia comercial soportada por la empresa *Franz Inc,* con una versión gratis para desarrollo con un límite de elementos.

DEX

DEX [32] es, según sus creadores, una biblioteca de alto rendimiento para gestionar grandes grafos o redes. En concreto, soporta multigrafos dirigidos y etiquetados con atributos. Desarrollada como una librería para Java, tiene una gran cantidad de métodos para gestionar y consultar grafos, aunque no proporciona un lenguaje de consulta, por lo que su uso es a bajo nivel. Su licencia es comercial, soportada por la empresa *Sparsity-Technologies* [43], aunque permite una descarga gratuita con fines académicos y de evaluación.

HyperGraphDB

HyperGraphDB [24] es un *framework* de almacenamiento para Java, basado en hipergrafos generalizados como modelo de datos, que internamente funciona con un almacén clave-valor:

una base de datos *BerkeleyDB* [37]. Está licenciado bajo los términos de la licencia GNU Lesser GPL.

InfiniteGraph

InfiniteGraph [25] es una base de datos de grafo distribuida, diseñada en Java para soportar aplicaciones que generan modelos de datos de grafo, complejos y de gran tamaño. Mientras que su arquitectura está diseñada para soportar requisitos de alto rendimiento y escalabilidad, su *API* es muy simple y en principio sólo requiere de un mínimo esfuerzo de desarrollo y diseño para construir aplicaciones. El producto tiene licencia comercial, y la compañía que se encarga de su soporte es *Objectivity Inc.*

Neo4j

La base de datos *Neo4j* [35] es una base de datos en grafo *NoSQL* de alto rendimiento, con todas las características de una base de datos robusta y madura. Ofrece una buena *API* orientada a objetos, un almacen nativo para manejar grafos en disco y un *framework* para recorrer grafos. Es un proyecto de código abierto disponible bajo licencia GPLv3 para su edición *Community*, y con ediciones comerciales con soporte por la empresa *Neo Technology*.

2.4. Identificación del Problema

El Cuadro 2.1 nos muestra una tabla comparativa entre los gestores de bases de datos de grafo recién mencionados.

Graph Database	**Etiquetas Arcos**	**Atributos Nodos**	**API**	**Lenguaje Consultas**
AllegroGraph			•	•
DEX	•	•	•	
HyperGraphDB			•	
InfiniteGraph	•	•	•	
Neo4j	•	•	•	○

Cuadro 2.1: Tabla comparativa.

AllegroGraph y *HyperGraphDB* no permiten etiquetar los arcos, por lo que su uso no es tan extendido como lo son los otros casos. Con respecto a *Neo4j*, cabe destacar que su lenguaje de consultas *Cypher* [15] (a pesar de permitir modelar grafos con etiquetas y atributos, además de contar con un extensa API) tiene una sintaxis y semántica poco clara, además de ser tan expresivo que su tiempo de ejecución puede llegar a ser muy alto, incluso para bases de datos pequeñas. Por otro lado, *DEX* e *InfiniteGraph* tienen un buen rendimiento, pero ninguno implementa un lenguaje de consultas.

Como podemos ver, el mecanismo más común en el uso de bases de datos de grafos es a través de *APIs*, las cuales son en forma de librerías con una descripción de todos los métodos disponibles en la implementación. Este mecanismo presenta varias ventajas:

1. Vocabulario estándar para funciones y métodos, ya que todas las bases de datos de grafos implementan los mismos métodos básicos de manipulación y de consulta de datos.

2. Fácil desarrollo de aplicaciones, ya que con sólo integrar la *API* en algún proyecto, se tendrá acceso a la base de datos de grafo y a todos sus métodos implementados.

3. Gran poder expresivo para consultar datos, ya que al contar con las operaciones básicas de grafos, otras consultas más complejas se pueden construir a partir de éstas.

Por otro lado, este enfoque presenta serios problemas:

1. Bajo nivel de abstracción para el usuario general, ya que el uso de la *API* requiere conocimientos de programación, y el usuario debe saber cómo computar las respuestas, mientras que con el uso de un lenguaje de consultas sólo es necesario declarar lo que se quiere.

2. Restricción según el lenguaje de programación, ya que por lo general la *API* está diseñada para ser usada en sólo un lenguaje de programación, teniendo que migrar la aplicación a dicho lenguaje para poder usar la base de datos de grafo.

3. La eficiencia depende de la implementación, ya que como el usuario es quien tiene el trabajo de diseñar los algoritmos para computar los resultados, es muy difícil lograr una gran eficiencia sin suficientes conocimientos.

4. Aparición de problemas de decisión, ya que como se dispone de todo el poder expresivo de un lenguaje de programación para computar consultas, es posible que alguna consulta que se desee implementar sea no decidible, o de una complejidad muy alta.

Si tuviéramos un lenguaje de consultas de bases de datos, podríamos extraer datos de forma fácil y rápida, ya que basta con declarar los datos que se quieren obtener, dejando todo el trabajo de cómo hacerlo al sistema. Además, con un lenguaje bien definido, se tiene más claro que se puede expresar y lo que no, evitando así la aparición de problemas indecidibles o de complejidad mayor.

Que no contemos con una buena implementación de algún lenguaje de consulta para bases de datos de grafos, que mantenga un buen equilibrio entre expresividad y eficiencia, es el principal problema identificado. Para solucionar dicho problema, vamos a continuar con el diseño de un lenguaje de consultas, que sea lo suficientemente expresivo para que tenga alguna utilidad interesante, pero cuidando que su evaluación se mantenga *tratable* computacionalmente, ya que de lo contrario no tiene ninguna utilidad práctica.

Capítulo 3

Lenguaje de Consulta Propuesto

El lenguaje propuesto es una variación de la lógica temporal *Propositional Dynamic Logic* (denotado *PDL*), de la cual vamos a considerar su extensión conocida como *Converse-PDL*, que extiende *PDL* con un operador de inverso (*converse* en inglés) para los arcos. Los operadores definidos en este lenguaje (a grandes rasgos) son los mismos que definen las expresiones regulares, además del inverso de arcos, combinaciones booleanas de fórmulas, un operador existencial con condiciones y fórmulas atómicas. La intuición de la expresividad de este lenguaje es que permite seleccionar un conjunto de nodos que cumpla ciertas condiciones, las cuales pueden ser:

- que un atributo cumpla una cierta propiedad,
- que exista un camino etiquetado con una condición de término,
- que los nodos intermedios del camino cumplan condiciones adicionales, o
- combinaciones booleanas de condiciones.

En la siguiente sección presentaremos algunos ejemplos que ayuden a clarificar el poder expresivo del lenguaje propuesto.

3.1. Intuición del Lenguaje

Para ejemplificar el poder expresivo del lenguaje propuesto, vamos a considerar un grafo en donde cada nodo representa una persona, y todos los arcos están etiquetados con *parent*, donde $(u, parent, v)$ se interpreta como "u es *padre* de v", por lo que el grafo representa un árbol genealógico. Además vamos a considerar que $name(p)$ denota el nombre de una persona p. A continuación vamos a mostrar algunos ejemplos de consultas que podemos expresar en *Converse-PDL*.

1. El tipo de consultas más básicas que se pueden definir en este lenguaje son las consultas atómicas, que definen conjuntos de nodos que cumplen alguna propiedad muy simple:

   a) Ser una persona (nodo) del grafo: $[\top]$

b) Ser la persona p en particular: $[\downarrow p]$

c) Ser una persona de nombre John: $[name = John]$

Como podemos observar, estos 3 tipos de consultas son los más básicos y todas las demás consultas se van construyendo de forma recursiva a partir de estas.

2. Mientras que para resolver las consultas anteriores no fueron necesarios los arcos del grafo, ahora podríamos pensar en el tipo de consultas más simples para los cuales sí sea necesario recorrer los arcos. Este tipo de consultas están compuesta por un solo predicado del alfabeto (en nuestro caso *parent*) y por una condición del tipo anterior. Por ejemplo, la expresión

$$\langle parent \rangle [\downarrow p]$$

se interpreta como el conjunto de nodos que tienen un arco con la etiqueta *parent* hasta el nodo p. Una característica de esta notación es que la consulta anterior puede ser leída textualmente como "los *padres* de p", la cual resulta ser muy intuitiva en nuestro idioma.

3. Si en nuestra base de datos es posible definir el conjunto de *padres* de un nodo en particular, es natural querer definir el conjunto de personas que tienen a un nodo en particular como *padre*. Este conjunto puede ser definido con la siguiente consulta:

$$\langle parent^- \rangle [\downarrow p].$$

El operador *converse* ($^-$) invierte la relación sobre la cual esté actuando, y como *parent* se interpreta como *padre*, entonces la expresión $parent^-$ se interpreta como *hijo*, por lo que la consulta anterior se lee como "los *hijos* de p".

4. El lenguaje propuesto permite realizar combinaciones booleanas de consultas, por lo que podemos realizar cualquiera de las operaciones básicas para manipular conjuntos. Por ejemplo, para definir el conjunto de "los *hijos* en común de las personas p_1 y p_2" basta con escribir

$$\langle parent^- \rangle [\downarrow p_1] \wedge \langle parent^- \rangle [\downarrow p_2].$$

Si en vez de querer conocer los *hijos* en común que tiene dos personas quisieramos conocer el conjunto de "los *hijos* no común de p_1 y p_2"podemos escribir

$$(\langle parent^- \rangle [\downarrow p_1] \wedge \neg \langle parent^- \rangle [\downarrow p_2]) \vee (\neg \langle parent^- \rangle [\downarrow p_1] \wedge \langle parent^- \rangle [\downarrow p_2]).$$

5. Todas las consultas que hemos definido hasta aquí han recorrido sólo los arcos incidentes al conjunto de nodos de partida. Si habláramos en términos de caminos (secuencias de nodos unidos por arcos), todas las consultas anteriores recorren caminos de largo 1. Una operación deseable sería recorrer los nodos del grafo siguiendo los arcos incidentes (es decir, navegar el grafo) e ir verificando condiciones en los caminos. La operación que permite esto es la *concatenación*, comúnmente representada con un punto ($\cdot$), logrando

así consultas más expresivas. Por ejemplo, si quisieramos definir el conjunto de *hermanos* (personas con un *padre* en común) de una persona p, es posible con la siguiente expresión:

$$\langle parent^{-} \cdot parent \rangle [\downarrow p].$$

Esta consulta se puede leer como "los *hijos* de los *padres* de p", coincidiendo con la definición de *hermanos* (considerando que *yo* soy *hermano* de *yo* mismo).

6. Gracias a la concatenación podemos definir muchas relaciones más complejas, por ejemplo

$$\langle parent \cdot parent \rangle [\downarrow p]$$

define el conjunto de *abuelos* de p. También podemos definir los conjuntos de *bisabuelos* y *tatarabuelos* de p, pero nos gustaría llegar a definir el conjunto de *ancestros* de p, el cual no es posible usando sólo los operadores anteriores. Nuestro lenguaje permite el uso de expresiones regulares, por lo que usando el operador unario conocido como *clausura de Kleene*, representada como un asterisco (*), podemos definir el conjunto de *ancestros* de p a través de la expresión

$$\langle parent^{*} \rangle [\downarrow p],$$

que se interpreta como el conjunto de personas que están conectadas por una secuencia (de largo 0 o más) de *padres* al nodo p, es decir, los *padres* o los *abuelos* o los *bisabuelos* o.... de p.

7. Finalmente, la combinación de las caracteristicas del lenguaje y el uso de atributos permite expresar consultas más sofisticadas, por ejemplo, podemos agregar condiciones adicionales a los nodos intermedios de un camino. Por ejemplo, podriamos definir el conjunto de *ancestros* de p en donde todo el linaje intermedio se llama John:

$$\langle parent \cdot ([name = John] \cdot parent)^{*} \rangle [\downarrow p].$$

Al agregar el operador de *test*, representado como ?, podríamos escribir consultas aún más sofisticadas, por ejemplo, la expresión

$$\langle parent \cdot ((parent^{-} \cdot [name = John])? \cdot parent)^{*} \rangle [\downarrow p]$$

define el conjunto de *ancestros* de p en donde todo el linaje intermedio tiene un *hijo* que se llama John.

La semántica de todas estas expresiones se hará clara en la siguiente sección, una vez que definamos formalmente el lenguaje de consultas.

3.2. Definición Formal

Nuestro lenguaje de consultas para bases de datos en grafos es una extensión del lenguaje *Proposicional Dynamic Logic* [19], denotado *PDL* por sus siglas en inglés, al cual se le agrega el operador *Converse*, obteniendo un mayor poder expresivo del lenguaje pero sin costo computacional adicional. A continuación definiremos la sintaxis y la semántica del lenguaje, para luego estudiar su expresividad, complejidad y computación.

3.2.1. Sintaxis

Definición 3.2.1 Sea $\mathcal{V}$ un conjunto infinito numerable de identificadores de nodos (ids), $\mathcal{S}$ un conjunto de palabras sobre algún alfabeto (palabras, enteros, etc) y Σ, $\mathcal{A}$ dos alfabetos finitos. Se definen las *fórmulas* ϕ y los *programas* α en *Converse-PDL* sobre Σ simultáneamente con la siguiente gramática:

$$\begin{aligned} \phi &: = \top \mid [\downarrow v] \mid [\sigma = k] \mid \phi \vee \phi \mid \phi \wedge \phi \mid \neg\phi \mid \langle\alpha\rangle\, \phi \\ \alpha &: = \varepsilon \mid c \mid c^{-} \mid \alpha + \alpha \mid \alpha \cdot \alpha \mid \alpha^{*} \mid \phi? \end{aligned}$$

donde $c \in \Sigma$, $v \in \mathcal{V}$, $\sigma \in \mathcal{A}$ y $k \in \mathcal{S}$.

En la definición, el símbolo | es usado como separador de las distintas reglas que componen la gramática, las cuales son usadas para generar el conjunto de *fórmulas* y *programas* válidos de nuestro lenguaje, y el símbolo ε denota la cadena vacía. La intuición de este lenguaje es que las *fórmulas* representan propiedades de nodos y los *programas* conectan pares de nodos a través de caminos etiquetados. Veamos a continuación la semántica de este lenguaje.

3.2.2. Semántica

Definición 3.2.2 Sea *G=(V,E,ρ)* una *GDB* sobre Σ, donde $V \subseteq \mathcal{V}$. Cada *programa* α en *Converse-PDL* define sobre G una relación binaria $\|\alpha\|_G$ en V. Análogamente, cada *fórmula* ϕ en *Converse-PDL* define sobre G un subconjunto $\|\phi\|_G$ de V. Las definiciones de $\|\alpha\|_G$ y de $\|\phi\|_G$ son simultáneas y recursivas. Vamos a partir con el caso de los *programas.* Asumamos que $c \in \Sigma$, que α, α_1 y α_2 son *programas* y que ϕ es una *fórmula*:

- $\|\varepsilon\|_G = \{(v, v) \mid v \in V\}$.
- $\|c\|_G = \{(u, v) \mid (u, c, v) \in E\}$.
- $\|c^{-}\|_G = \{(u, v) \mid (v, c, u) \in E\}$.
- $\|\alpha_1 + \alpha_2\|_G = \|\alpha_1\|_G \cup \|\alpha_2\|_G$.
- $\|\alpha_1 \cdot \alpha_2\|_G = \|\alpha_1\|_G \circ \|\alpha_2\|_G$.
- $\|\alpha^{*}\|_G = \bigcup_{n \geq 0} \|\alpha^n\|_G$.
- $\|\phi?\|_G = \{(u, u) \mid u \in \|\phi\|_G\}$.

En la definición anterior, $\circ$ denota la composición usual de relaciones binarias, es decir

$$\|\alpha_1\|_G \circ \|\alpha_2\|_G = \{(u, v) \mid \exists w \in V \text{ tal que } (u, w) \in \|\alpha_1\|_G \text{ y } (w, v) \in \|\alpha_1\|_G\}.$$

Además, el término $\|\alpha^n\|_G$ denota la composición n veces α, definida recursivamente como $\|\alpha^n\|_G = \|\alpha\|_G \circ \|\alpha^{n-1}\|_G$ con $\|\alpha^0\|_G = \|\varepsilon\|_G$.

Dicho de otra forma, la semántica de los *programas* es la siguiente: ε define la identidad en $V \times V$, c define los pares de nodos que están unidos por un arco con etiqueta c, mientras

que c^- define el inverso de c. Luego $\alpha_1 + \alpha_2$ define la unión de ambas relaciones, $\alpha_1 \cdot \alpha_2$ define la composición ya descrita anteriormente, y α^* es la cláusula transitiva-reflexiva que define los pares de nodos que están unidos por 0 o más veces por el *programa* α. Finalmente $\phi?$ define el conjunto de pares (u, u) tal que u satisface la *fórmula* ϕ.

Definición 3.2.3 Supongamos que $v \in \mathcal{V}$, $k \in \mathcal{S}$, α es un *programa,* y que ϕ, ϕ_1 y ϕ_2 son *fórmulas*. La semántica de las *fórmulas* se define como:

- $\|\top\|_G = V$.
- $\|[\downarrow v]\|_G = \{v\}$ si $v \in V$, y $\|[\downarrow v]\|_G = \emptyset$ en caso contrario.
- $\|[\sigma = k]\ \|_G = \{v \in V \mid \sigma(v) = k\}$.
- $\|\phi_1 \vee \phi_2\|_G = \|\phi_1\|_G \cup \|\phi_2\|_G$.
- $\|\phi_1 \wedge \phi_2\|_G = \|\phi_1\|_G \cap \|\phi_2\|_G$.
- $\|\neg\phi\|_G = V \setminus \|\phi\|_G$.
- $\|\langle\alpha\rangle\, \phi\|_G = \{u \in V \mid \exists v \in V$ tal que $(u, v) \in \|\alpha\|_G$ y $v \in \|\phi\|_G\}$.

Dicho de otra forma, la semántica de las *fórmulas* es la siguiente: $\top$ define el conjunto de todos los nodos, $[\downarrow v]$ es verdad sólo en el nodo v, y $[\sigma = k]$ define el conjunto de nodos que tienen el valor k en el atributo σ. Luego $\phi_1 \vee \phi_2$ define la unión de los conjuntos, $\phi_1 \wedge \phi_2$ define la intersección de los conjuntos, y $\neg\phi$ define el complemento del conjunto. Adicionalmente, $\langle\alpha\rangle\, \phi$ define el conjunto de nodos u desde el cual es posible alcanzar a través del programa α algún nodo v que satisfaga ϕ.

3.3. Estudio de Expresividad

En la sección 3.1 presentamos varios ejemplos de consultas relevantes que pueden ser expresadas en *Converse-PDL*. Este lenguaje contiene otros importantes lenguajes *navegacionales* para bases de datos en grafos que han sido estudiados en la literatura, tales como *nested regular expression [5]*, pero es incomparable con otros lenguales tales como *conjuntive regular path queries [4]*. A continuación vamos a dar una intuición de la expresividad de nuestro lenguaje, para luego compararlo con otros lenguajes para bases de datos en grafos importantes en el área.

La gran expresividad que se logra con *Converse-PDL* se le puede atribuir a que se cuenta con todo el poder expresivo de las expresiones regulares, acompañado de la lógica booleana y de un cuantificador existencial. La combinación de estos tres elementos juntos permite expresar un gran número de consultas bastante complejas. Por ejemplo, podemos definir una nueva expresión utilizando sólo elementos en *Converse-PDL,* obteniendo así un nuevo operador que no es trivial.

Ejemplo 3.3.1 Definamos el operador $(\forall\alpha)\phi \equiv \neg\langle\alpha\rangle\neg\phi$. A continuación vamos a desarrollar su semántica, con el fin de encontrar alguna expresión más clara y equivalente:
$\|(\forall\alpha)\phi\|_G = \|\neg\langle\alpha\rangle\neg\phi\|_G = V \setminus \|\langle\alpha\rangle\neg\phi\|_G$
$= \{u \mid \neg\exists v \text{ tal que } (u,v) \in \|\alpha\|_G \text{ y } v \in \|\neg\phi\|_G\}$
$= \{u \mid \forall v \text{ se cumple que } (u,v) \notin \|\alpha\|_G \text{ o } v \notin \|\neg\phi\|_G\}$
$= \{u \mid \forall v \text{ se cumple que } (u,v) \notin \|\alpha\|_G \text{ o } v \in \|\phi\|_G\}$,
concluyendo así que su semántica corresponde a

$$\|(\forall\alpha)\phi\|_G = \{u \in V \mid \text{Si } \forall v \in V\ (u,v) \in \|\alpha\|_G, \text{ entonces } v \in \|\phi\|_G\}.$$

Ejemplo 3.3.2 Podemos observar que el operador $(\forall\alpha)\phi$ corresponde a un cuantificador universal, el cual nos permite expresar consultas más complejas, tales como consultas de la forma $(\forall\alpha)\langle\alpha\rangle[\phi]$, que se corresponde a un tipo de alternación de cuantificadores. Por ejemplo, si α lo identificamos como *padre* y ϕ como $\top$, entonces la consulta anterior se interpreta como "las personas que todos sus *hijos* tienen algún *hijo*".

3.3.1. Comparación con *Nested Regular Expression*

El lenguaje de consultas de base de datos de grafos *Nested regular expression* (que originalmente fue propuesto para consultar datos en la *Web Semántica*, abreviado como *NRE*) es considerado bastante expresivo, ya que contiene un importante lenguaje de consultas conocido como *two-way regular path queries,* y a su vez tiene una complejidad tratable *[5]*. A continuación se enuncia un resultado bastante bueno, que nos permitirá entender de mejor manera la expresividad de *Converse-PDL*.

Proposición 3.3.1 Para toda *NRE* n existe un *programa* en *Converse-PDL* α tal que para todo *GDB* G se cumple que $\|n\|_G = \|\alpha\|_G$.

Este resultado nos indica que *Converse-PDL* contiene a *NRE*, por lo que puedo expresar cualquier *nested regular expression.* Para demostrar la proposición, vamos a definir formalmente el lenguaje *NRE*.

Definición 3.3.1 *NREs*
(Sintaxis) Sea Σ un alfabeto finito. Una *NRE* es una consulta en grafos con la siguiente gramática

$$E := \varepsilon \mid c \mid c^- \mid E_1 \cdot E_2 \mid E_1 + E_2 \mid E^* \mid [E]$$

donde $c \in \Sigma$ y E, E_1, E_2 son *NREs*
(Semántica) Dada una *RPQ* E y una *GDB* $G = (V, A)$. La evaluación de E sobre G, denotada $\|E\|_G$, se define como:

- $\|\varepsilon\|_G = \{(u,u) \mid u \in V\}$
- $\|c\|_G = \{(u,v) \mid (u,c,v) \in A\}$
- $\|c^-\|_G = \{(u,v) \mid (v,c,u) \in A\}$
- $\|E_1 \cdot E_2\|_G = \|E_1\|_G \circ \|E_2\|_G$

- $\|E_1 + E_2\|_G = \|E_1\|_G \cup \|E_2\|_G$
- $\|E^*\|_G = \bigcup_{n\geq 0} \|E^n\|_G$, donde $E^0 = \varepsilon$, y $E^n = EE...E$ concatenado n veces
- $\|[E]\|_G = \{(u,u) \mid \exists z\ (u,z) \in \|E\|_G\}$

Podemos notar que la definición de *NRE* es muy similar a la definición de los *programas* de *Converse-PDL*, por lo que la demostración es bastante intuitiva.

Demostración 3.3.1 Para demostrar que toda fórmula en *NRE* puede expresarse en *Converse-PDL* (también denotado como PDL^-), basta con encontrar una traducción a cada una de las reglas que definen el lenguaje. Definamos la transformación $\Phi : NRE \to PDL^-$ que asigna a cada fórmula en *NRE* un programa en PDL^- con ua semántica equivalente, es decir $\|E\|_G = \|\Phi(E)\|_G$. Los casos bases son:

- $\Phi(\varepsilon) = \varepsilon$,
- $\Phi(c) = c$, donde $c \in \Sigma$,
- $\Phi(c^-) = c^-$, donde $c \in \Sigma$.

Luego procedemos a definir los casos de la concatenación, unión y recursión:

- $\Phi(E_1 \cdot E_2) = \Phi(E_1) \cdot \Phi(E_2)$,
- $\Phi(E_1 + E_2) = \Phi(E_1) + \Phi(E_2)$,
- $\Phi(E^*) = \Phi(E)^*$.

Hasta ahora las gramáticas coinciden completamente, pero podemos ver que el *nesting* es un caso especial:

- $\Phi([E]) = (\langle \Phi(E) \rangle \top)?$.

Es trivial que los seis primeros casos coinciden en su semántica, por lo que basta con revisar el último caso.

1. De la definición de la semántica del operador *nesting*
   $\|[E]\|_G = \{(u,u) \mid \exists z\ (u,z) \in \|E\|_G\}$.
2. De la definición de Φ y del operador *test*
   $\|\Phi([E])\|_G = \|(\langle \Phi(E) \rangle \top)?\|_G = \{(u,u) \mid u \in \|\langle \Phi(E) \rangle \top\|_G\}$.
3. De la definición de las fórmulas $\langle \alpha \rangle \phi$ y $\top$
   $\|\langle \Phi(E) \rangle \top\|_G = \{u \in V \mid \exists z\ (u,z) \in \|\Phi(E)\|_G\}$.
4. Finalmente, de *1, 2* y *3* se deduce la equivalencia
   $\|\Phi([E])\|_G = \{(u,u) \mid \exists z\ (u,z) \in \|\Phi(E)\|_G\} = \|[E]\|_G$,

finalizando con nuestra demostración.

El resultado anterior nos muestra que *Converse-PDL* es al menos tan expresivo como *NRE*, pero no es claro que sea más expresivo. Por fortuna, tenemos un resultado que nos muestra que si lo es. Para mostrar nuestro resultado, primero necesitamos definir una propiedad de los lenguajes de consulta: la *monotonía*.

Definición 3.3.2 Sea Σ un alfabeto finito. Dos grafos $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ sobre Σ se dicen $G_1 \subseteq G_2$ ssi $V_1 \subseteq V_2$ y $E_1 \subseteq E_2$.Un lenguaje de consultas de grafos L es monótono ssi para toda fórmula $\phi \in L$ y para todo grafo G_1, G_2 tal que $G_1 \subseteq G_2$ se cumple que

$$\|\phi\|_{G_1} \subseteq \|\phi\|_{G_2}.$$

Lema 3.3.1 NRE es monótono y PDL^- no es monótono.

Demostración 3.3.2 Demostremos primero que NRE es monótono. Sean $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ dos grafos cualquieras que cumplen $G_1 \subseteq G_2$. Por inducción sobre la gramática que define NRE, es fácil ver que se cumple la propiedad.

- $\|\varepsilon\|_{G_1} = \{(u,u) \mid u \in V_1\} \subseteq \{(u,u) \mid u \in V_2\} = \|\varepsilon\|_{G_2}$,
- $\|c\|_{G_1} = \{(u,v) \mid (u,c,v) \in E_1\} \subseteq \{(u,v) \mid (u,c,v) \in E_2\} = \|c\|_{G_2}$,
- $\|c^-\|_{G_1} = \{(u,v) \mid (v,c,u) \in E_1\} \subseteq \{(u,v) \mid (v,c,u) \in E_2\} = \|c^-\|_{G_2}$,
- $\|E_1 \cdot E_2\|_{G_1} = \|E_1\|_{G_1} \circ \|E_2\|_{G_1} \subseteq \|E_1\|_{G_2} \circ \|E_2\|_{G2} = \|E_1 \cdot E_2\|_{G_2}$,
- $\|E_1 + E_2\|_{G_1} = \|E_1\|_{G_1} \cup \|E_2\|_{G_1} \subseteq \|E_1\|_{G_2} \cup \|E_2\|_{G_2} = \|E_1 + E_2\|_{G_2}$,
- $\|E^*\|_{G_1} = \bigcup_{n\geq 0} \|E^n\|_{G_1} \subseteq \bigcup_{n\geq 0} \|E^n\|_{G_2} = \|E^*\|_{G_2}$,
- $\|[E]\|_{G_1} = \{(u,u) \mid \exists z\ (u,z) \in \|E\|_{G_1}\} \subseteq \{(u,u) \mid \exists z\ (u,z) \in \|E\|_{G_2}\} = \|[E]\|_{G_2}$.

Esto basta para demostramos que NRE es monótono. Para revisar el caso de PDL^- basta con considerar el siguiente caso:

- $\phi = \neg \langle c \rangle \top$,
- $V_1 = V_2 = \{1, 2\}$,
- $E_1 = \emptyset$ y $E_2 = \{(1, c, 2)\}$.

La evaluación de ϕ en un grafo G está dada por $\|\phi\|_G = \{v \in V \mid \neg\exists z\ (v,c,z) \in E\}$. Es fácil ver que $G_1 \subseteq G_2$, pero la evaluación de ϕ en G_1 y G_2 no cumple la inclusión:

$$\|\phi\|_{G_1} = \{1, 2\} \not\subseteq \{2\} = \|\phi\|_{G_2},$$

demostrando así la *no-monotonía* de PDL^-.

Como vimos, NRE está contenido en PDL^-, pero como NRE es monótono y PDL^- no lo es, los lenguajes no pueden coincidir, permitiendonos deducir el siguiente corolario.

Corolario 3.3.1 PDL^- es estrictamente más expresivo que NRE.

3.3.2. Comparación con *Conjuntive Two-way Regular Path Queries*

Un conocido lenguaje de consultas de base de datos de grafo que tiene una gran expresividad es el *conjuntive two-way regular path queries* (denotado *C2RPQ*), ya que permite expresar propiedades de ciclos. Veamos a continuación la definición formal de este lenguaje.

Definición 3.3.3 Sea Σ un alfabeto finito. Sea $\bar{x} = (x_1, ..., x_n)$ e $\bar{y} = (y_1, ..., y_m)$ tuplas de variables distintas. Denotemos $\bar{x} \cup \bar{y} = \{x_1, ..., x_n, y_1, ..., y_m\}$. Una $C2RPQ$ sobre Σ con variables libres $\bar{x}$ es una fórmula φ de la forma

$$\varphi(\bar{x}) = \exists \bar{y} \bigwedge_{i=1}^{k} (z_i, r_i, z_i'),$$

donde r_i son expresiones regulares con inverso sobre Σ, para todo $1 \leq i \leq k$, y $\bar{x} \cup \bar{y} = \bigcup_{i=1}^{k} \{z_i, z_i'\}$.

Proposición 3.3.2 Existe un *programa* en PDL^- sobre el alfabeto $\Sigma = \{a, b\}$ que no puede ser expresado como uniones de $C2RPQ$ (denotado $UC2RPQ$).

Demostración 3.3.3 El programa $\alpha = (a \cdot (\langle b \rangle \top)?)^+$ no puede ser expresado en $UC2RPQ$, por el hecho que es equivalente a la expresión $E = (a \cdot [b])^+$ contenido en NRE, la cual se demuestra en *[5]* que no es expresable en $UC2RPQ$.

Lema 3.3.2 Existe una fórmula unaria en $C2RPQ$ que no puede ser expresada en PDL^-.

Demostración 3.3.4 Consideremos el alfabeto $\Sigma = \{a\}$. Definamos la fórmula $\varphi(x) = (x, a, x)$, y consideremos para $n \geq 1$ el grafo $G_n = (V_n, E_n)$ donde $V_n = \{n, ..., 2n\}$ y $E_n = \{(k, a, k+1) \mid n \leq k \leq 2n-1\} \cup \{(2n, a, n), (2n, a, 2n)\}$. Podemos ver que la evaluación de φ en G_n es

$$\|\varphi\|_{G_n} = \{2n\}.$$

Demostremos por inducción que para toda *fórmula* ϕ en PDL^- existe un n tal que su evaluación en G_n es $\emptyset$ o V_n, y que todo *programa* α en PDL^- existe un n tal que su evaluación en G_n es $\emptyset$ o $I_n = \{(u, u) \mid u \in V_n\}$. Verifiquemos los casos bases de la gramática que define los *programas* de PDL^-:

- $\|\varepsilon\|_{G_n} = \{(u, u) \mid u \in V_n\} = I_n$ para todo n,
- $\|a\|_{G_n} = \{(u, v) \mid (u, c, v) \in E_n\} = I_n$ para todo n,
- $\|a^-\|_{G_n} = \{(u, v) \mid (v, c, u) \in E_n\} = I_n$ para todo n,

Verifiquemos los casos bases de la gramática que define las *fórmulas* de PDL^-:

- $\|\top\|_{G_n} = V_n$ para todo n,
- $\|[\downarrow v]\|_{G_n} = \emptyset$ si $n > v$ ya que en ese caso $v \notin V_n$,

- $\|[\sigma = k]\|_{G_n} = \emptyset$ para todo n ya que el grafo no tiene atributos.

Por inducción, vamos a suponer que para todos los *programas* y *fórmulas* de hasta un cierto largo cumplen nuestra hipótesis inductiva. Los casos de inducción para los *programas* son:

- $\|\alpha_1 + \alpha_2\|_{G_{\text{máx}(n_1,n_2)}} = \begin{cases} \emptyset & \text{si } \|\alpha_1\|_{G_{n_1}} = \emptyset \text{ y } \|\alpha_2\|_{G_{n_2}} = \emptyset \\ I_n & \text{en caso contrario} \end{cases}$,
- $\|\alpha_1 \cdot \alpha_2\|_{G_{\text{máx}(n_1,n_2)}} = \begin{cases} \emptyset & \text{si } \|\alpha_1\|_{G_{n_1}} = \emptyset \text{ ó } \|\alpha_2\|_{G_{n_2}} = \emptyset \\ I_n & \text{en caso contrario} \end{cases}$,
- $\|\alpha^*\|_{G_n} = I_n$ para todo n,
- $\|\phi?\|_{G_n} = \begin{cases} \emptyset & \text{si } \|\phi\|_{G_n} = \emptyset \\ I_n & \text{si } \|\phi\|_{G_n} = V_n \end{cases}$ para todo n.

Los casos de inducción para las *fórmulas* son:

- $\|\phi_1 \vee \phi_2\|_{G_{\text{máx}(n_1,n_2)}} = \begin{cases} \emptyset & \text{si } \|\phi_1\|_{G_{n_1}} = \emptyset \text{ y } \|\phi_2\|_{G_{n_2}} = \emptyset \\ V_n & \text{en caso contrario} \end{cases}$,
- $\|\phi_1 \wedge \phi_2\|_{G_{\text{máx}(n_1,n_2)}} = \begin{cases} \emptyset & \text{si } \|\phi_1\|_{G_{n_1}} = \emptyset \text{ ó } \|\phi_2\|_{G_{n_2}} = \emptyset \\ V_n & \text{en caso contrario} \end{cases}$,
- $\|\neg\phi\|_{G_n} = \begin{cases} \emptyset & \text{si } \|\phi\|_{G_n} = V_n \\ V_n & \text{si } \|\phi\|_{G_n} = \emptyset \end{cases}$,
- $\|\langle\alpha\rangle\,\phi\|_{G_{\text{máx}(n_1,n_2)}} = \begin{cases} \emptyset & \text{si } \|\phi\|_{G_{n_1}} = \emptyset \text{ ó } \|\alpha\|_{G_{n_2}} = \emptyset \\ V_n & \text{si } \|\phi\|_{G_n} = V_n \text{ y } \|\phi\|_{G_n} = V_n \end{cases}$.

Luego, para toda *fórmula* ϕ en PDL^- existe un grafo G_n donde su evaluación es $\emptyset$ ó V_n, mientras que la evaluación de $\varphi(x) = (x, a, x)$ siempre es $\{2n\}$, por lo que todas las fórmulas en PDL^- difieren de $\varphi(x)$ en algún G_n, por lo cual $\varphi(x)$ no es expresable en PDL^-.

Gracias a estos dos resultados, podemos concluir que los lenguajes PDL^- y $(U)C2RPQ$ son incomparables, ya que ninguna está contenido en el otro. Las expresiones en *Converse-PDL* cumplen la propiedad de ser *acíclicas*, es decir, no pueden expresar propiedades de ciclos en la *GDB*. Por ejemplo, si consideramos el mismo alfabeto $\Sigma = \{parent\}$ que utilizamos anteriormente, no existe ninguna fórmula ϕ en *Converse-PDL* tal que para toda *GDB* G sobre Σ, el conjunto $\|\phi\|_G$ coincida con el conjunto de personas que tengan dos hijos diferentes *[5]*. Esta carencia de expresividad del lenguaje es lo que permite que el lenguaje sea eficiente, ya que incluir ciclos en las consultas convierte el problema de evaluación en intratable *[4]*.

3.4. Análisis de Complejidad

Para poder realizar un análisis de complejidad, debemos definir los términos que necesitaremos de aquí en adelante.

Definición 3.4.1 Sean Σ, $\mathcal{A}$ dos alfabetos finitos, $G = (V, E, \rho)$ una *GDB*, ϕ una *fórmula* en PDL^- y α un *programa* en PDL^-. Definimos sus tamaños de la siguiente forma:

1. Conjuntos

   $|C| = \#\{c \mid c \in C\}$ para $C \in \{\Sigma,\ \mathcal{A},\ V, E\}$.

2. Grafos

   $|G| = |V| + |E| + |\rho| = |V| + |E| + |V| \times \mathcal{A} = |E| + |V| \times (1 + \mathcal{A})$.

3. Fórmulas

   $|\phi| = 1$ en los casos $\top \mid [\downarrow v] \mid [\sigma = k]$,

   $|\phi| = 1 + |\phi'|$ en el caso $\neg\phi'$,

   $|\phi| = 1 + |\phi'| + |\phi''|$ en los casos $\phi' \vee \phi'' \mid \phi' \wedge \phi''$,

   $|\phi| = 1 + |\alpha| + |\phi'|$ en el caso $\langle\alpha\rangle\, \phi'$.

4. Programas

   $|\alpha| = 1$ en los casos $\varepsilon \mid c \mid c^-$,

   $|\alpha| = 1 + |\bar{\alpha}|$ en el caso $\bar{\alpha}^*$,

   $|\alpha| = 1 + |\bar{\alpha}| + |\tilde{\alpha}|$ en los casos $\bar{\alpha} + \tilde{\alpha} \mid \bar{\alpha} \cdot \tilde{\alpha}$,

   $|\alpha| = 1 + |\phi|$ en el caso $\phi?$.

Ahora que ya hemos definido el concepto de tamaño de los distintos elementos, el siguiente paso es definir el costo computacional a considerar. Vamos a considerar que las operaciones de lectura/escritura/comparación tienen costo 1 en nuestro modelo de computación, y que ejecutar n operaciones consecutivas de lectura/escritura/comparación tiene costo n.

Definición 3.4.2 Diremos que un algoritmo A computa la función de evaluación $\lambda(\phi, G) = \|\phi\|_G$ en tiempo $T(n, m)$ si, para toda *fórmula* ϕ y para todo grafo G, el algoritmo retorna el conjunto $\|\phi\|_G$ en a lo sumo $T(|\phi|, |G|)$ operaciones. La definición para caso de los *programas* α es análogo.

Existen casos en donde el tamaño de los conjuntos $\|\phi\|_G$ y $\|\alpha\|_G$ puede llegar a ser $|G|$, por lo que resolver algunas fórmulas de largo 1 requieren realizar por lo menos $|G|$ operaciones, tal como es el caso del *programa* ε y de la *fórmula* $\top$. Por otro lado, existen casos de *programas* donde el tamaño de la salida es de tamaño $O(|G|^2)$, tal como el caso de $G = (V, E)$ con $V = \{1, ..., n\}$ y $E = \{(k, a, k+1) \mid 1 \leq k \leq n-1\} \cup \{(n, a, 1)\}$ en donde $|\|a^*\|_G| = |V \times V| = |V|^2$, mientras que $|G| = |V| + |E| = |V| + |V| = 2|V|$. Además, para leer una fórmula ϕ se deben realizar como mínimo $|\phi|$ operaciones de lectura del grafo, que en el peor de los casos puede llegar a ser equivalente a resolver fórmulas de largo 1, como es el caso de $\phi = \top \vee \top \vee ... \vee \top$,

que al evaluar cada $\top$ obtengo un conjunto de tamaño $|V|$, y al realizar la unión de dos conjuntos debo leer sus elementos, se alcanza un costo total de $O(|G| \cdot |\phi|)$.

El costo de computar los conjuntos $\|\phi\|_G$ y $\|\alpha\|_G$ en el peor caso se obtiene del siguiente resultado conocido, que puede ser probado usando técnicas estándares de *model checking [13]*.

Teorema 3.4.1 Complejidad de Evaluación de *Converse-PDL*

1. El costo de computar el conjunto $\|\phi\|_G$, para una *GDB* G y una *fórmula* ϕ en *Converse-PDL* sobre un alfabeto Σ de tamaño fijo es $O(|G| \cdot |\phi|)$.

2. El costo de computar el conjunto $\|\alpha\|_G$, para una *GDB* G y un *programa* α en *Converse-PDL* sobre un alfabeto Σ de tamaño fijo es $O(|G|^2 \cdot |\alpha|)$.

El resultado anterior nos indica que las *fórmulas* en *Converse-PDL* pueden ser evaluadas en tiempo lineal en el tamaño de los datos, y que los *programas* en *Converse-PDL* pueden ser evaluados en tiempo cuadrático en el tamaño de los datos, coincidiendo con el tamaño máximo del conjunto de salida. Como las bases de datos son dinámicas, sus tamaños pueden seguir creciendo en el tiempo, alcanzado un orden de miles de millones de datos, por lo que permitir un tiempo de ejecución proporcional al cuadrado o más del tamaño de la base de datos es prohibitivo. Supongamos que podamos procesar 1 millón de datos por segundo, entonces computar un *programa* en una base de datos de 1 millón de datos, puede llegar a demorar más de 11 días.

Por esta razón, el lenguaje propuesto como consultas de bases de datos de grafos debe tener una complejidad de evaluación de $O(|G| \cdot |\phi|)$, coincidiendo con el caso de las *fórmulas* en *Converse-PDL*, mientras que los *programas* en *Converse-PDL* son constructores implícitos utilizados en la definición de dichas fórmulas. En la práctica, el tamaño de las consultas es mucho menor que el tamaño de la base de datos (del orden de miles de millones de veces), por lo que el tamaño de la base es lo más importante al momento de evaluar su complejidad. Por esta razón, vamos a considerar que el tamaño de la consulta es constante, por lo que el tiempo de evaluación se expresa en términos del tamaño de G, razón por la cual definimos una noción de optimalidad.

Definición 3.4.3 Un algoritmo A que computar la función de evaluación se dice que es *óptimo débil,* en términos de *complejidad de evaluación,* si el peor caso sobre todas las *fórmulas* de tamaño $|\phi| = n$ de computar el conjunto $\|\phi\|_G$ toma tiempo $O(|G| \cdot n)$.

Note que existen nociones más fuerte de optimalidad, como es la de computar cada fórmula en el menor tiempo posible. Determinar algoritmos que sean óptimos en un sentido más fuerte es parte del trabajo futuro, mientras que en el siguiente capítulo diseñaremos algoritmos óptimos en el sentido débil.

Capítulo 4

Diseño de la Implementación

4.1. Evaluación de Consultas

Que la teoría nos diga que la evaluación de *fórmulas* en *Converse-PDL* tome tiempo lineal en el tamaño de los datos, nos sugiere que es posible implementar nuestro lenguaje sobre bases de datos de grafos de forma eficiente. Sin embargo, computar el conjunto de respuestas de una *fórmula* en *Converse-PDL* sobre una *GDB* requiere una gran cantidad de trabajo adicional, ya que existen diversas restricciones a considerar al momento de implementar el lenguaje. A lo largo de este capítulo desarrollaremos el diseño de la implementación de nuestro lenguaje de consulta, comenzando con los algoritmos de mayor nivel de abstracción, hasta llegar al nivel más bajo de abstracción.

4.1.1. Función de Evaluación

Para desarrollar un algoritmo de evaluación de consultas, vamos a partir por un fragmento de *Converse-PDL* conocido como *Recursion-Free-Converse-PDL*, el cuál se obtiene a partir de *Converse-PDL* removiendo el operador $()^*$. La gramática que describe este lenguaje, donde $c \in \Sigma$, es la siguiente:

$$\begin{aligned} \phi &: = \top \mid [\downarrow v] \mid [\sigma = k] \mid \phi \vee \phi \mid \phi \wedge \phi \mid \neg\phi \mid \langle\alpha\rangle\, \phi \\ \alpha &: = \varepsilon \mid c \mid c^{-} \mid \alpha + \alpha \mid \alpha \cdot \alpha \mid \phi? \end{aligned}$$

Computar las fórmulas atómicas y las combinaciones booleanas las revisaremos más adelante, por lo que ahora nos vamos a concentrar en las fórmulas de la forma $\langle\alpha\rangle\, \phi$. Si consideramos todos los casos posibles de *programas* que puede tomar la expresión α, es fácil ver que se cumple la siguiente proposición.

Proposición 4.1.1 Las siguientes equivalencias son siempre verdaderas:

- $\langle\varepsilon\rangle\, \phi \equiv \phi$.
- $\langle c\rangle\, \phi \equiv \{u \in V \mid \exists v \text{ tal que } (u, a, v) \in E \text{ y } v \in \|\phi\|_G\}$.

- $\langle c^- \rangle \, \phi \equiv \{u \in V \mid \exists v \text{ tal que } (v, a, u) \in E \text{ y } v \in \|\phi\|_G\}$.
- $\langle \alpha_1 + \alpha_2 \rangle \, \phi \equiv \langle \alpha_1 \rangle \, \phi \vee \langle \alpha_2 \rangle \, \phi$.
- $\langle \alpha_1 \cdot \alpha_2 \rangle \, \phi \equiv \langle \alpha_1 \rangle \, (\langle \alpha_2 \rangle \, \phi)$.
- $\langle \phi_0 ? \rangle \, \phi \equiv \phi_0 \wedge \phi$.

Gracias a estas equivalencias, y por el hecho que el lenguaje es cerrado bajo disyunción y conjunción, es posible construir en tiempo lineal para cada *fórmula* en el fragmento *Recursion-Free-Converse-PDL*, una *fórmula* en *Converse-PDL* equivalente en el lenguaje restringido definido por la siguiente gramática:

$$\phi := \top \mid [\downarrow v] \mid [\sigma = k] \mid \phi \vee \phi \mid \phi \wedge \phi \mid \neg\phi \mid \langle c \rangle \, \phi \mid \left\langle c^- \right\rangle \phi.$$

Utilizando la gramática anterior, vamos a definir una función de evaluación λ, que a cada *fórmula* ϕ le asigne un conjunto de nodos V, el cual corresponde a todos los nodos que satisfacen la *fórmula* ϕ. Esta función λ representa la semántica del lenguaje de consulta, por lo cual es el objetivo final de la implementación.

Definición 4.1.1 Sea L el conjunto de *fórmulas en Recursion-Free-Converse-PDL.* Dado un *GDB* $G = (E, V, \rho)$, se define su función de evaluación de *Recursion-Free-Converse-PDL* $\lambda : L \to 2^V$ como:

$$\lambda(\phi) = \bar{\lambda}(\phi, V),$$

donde $\bar{\lambda} : L \times 2^V \to 2^V$ es la función de evaluación auxiliar definida inductivamente como:

- $\bar{\lambda}(\top, V_0) = V_0$.
- $\bar{\lambda}([\downarrow v] \, , V_0) = \{u \in V_0 |\ u = v\}$.
- $\bar{\lambda}([\sigma = k], V_0) = \{u \in V_0 |\ \sigma(v) = k \ \}$.
- $\bar{\lambda}(\langle c \rangle \, \phi, V_0) = \{u \in V |\ \exists v \in \bar{\lambda}(\phi, V_0) \text{ tal que } (u, c, v) \in E\}$.
- $\bar{\lambda}(\langle c^- \rangle \, \phi, V_0) = \{u \in V |\ \exists v \in \bar{\lambda}(\phi, V_0) \text{ tal que } (v, c, u) \in E\}$.
- $\bar{\lambda}(\phi_1 \vee \phi_2, V_0) = \bar{\lambda}(\phi_1, V_0) \cup \bar{\lambda}(\phi_2, V_0)$.
- $\bar{\lambda}(\phi_1 \wedge \phi_2, V_0) = \bar{\lambda}(\phi_1, V_0) \cap \bar{\lambda}(\phi_2, V_0)$.
- $\bar{\lambda}(\neg\phi, V_0) = V \setminus \bar{\lambda}(\phi, V_0)$.

De ahora en adeltante, siempre y cuando no haya confusión, vamos a denotar $\bar{\lambda}$ como λ. La intuición de esta función es que, dado un *fórmula* ϕ y un conjunto de nodos V_0, computa el conjunto de todos los nodos desde los cuales es posible alcanzar a un nodo en V_0, a través de un camino etiquetado que satisfaga la *fórmula* ϕ. Suponiendo que logramos implementar la función $\bar{\lambda}$ de forma correcta, entonces el conjunto devuelto al evaluar la *fórmula* ϕ en la *GDB* $G = (E, V, \rho)$ coincide con su semántica $\|\phi\|_G$.

A la gramática anterior podemos agregar tres operadores adicionales, que no agregan mayor expresividad al lenguaje, pero simplifican la escritura de algunas consultas, y nos permitirán expresar la función de evaluación para el caso de la recursión.

Proposición 4.1.2 El lenguaje *Recursion-Free-Converse-PDL* es equivalente al lenguaje definido por la gramática

$$\phi := \top \mid [\downarrow v] \mid [\sigma = k] \mid \phi \vee \phi \mid \phi \wedge \phi \mid \neg\phi \mid \langle c \rangle \phi \mid \left\langle c^{-} \right\rangle \phi \mid \phi - \phi \mid \phi \cdot \phi \mid \phi? \ ,$$

donde la función de evaluación se extiende de la siguiente forma:

- $\lambda(\phi_1 - \phi_2, V_0) = \lambda(\phi_1, V_0) \setminus \lambda(\phi_2, V_0)$.
- $\lambda(\phi_1 \cdot \phi_2, V_0) = \lambda(\phi_1, \lambda(\phi_2, V_0))$.
- $\lambda(\phi?, V_0) = \lambda(\phi, V) \cap V_0$.

Proposición 4.1.3 Supongamos que las operaciones de consultar, insertar y eliminar un elemento de un conjunto toma tiempo $O(1)$, y que evaluar los casos $\top, [\downarrow v], [\sigma = k], \langle c \rangle \phi$ y $\langle c^{-} \rangle \phi$ toma tiempo $O(|G|)$. Entonces, computar la función de evaluación $\lambda(\phi, V)$ toma tiempo $O(|\phi||G|)$.

Demostración 4.1.1 Bajo las hipótesis, es fácil ver que la evaluación toma tiempo $O(|\phi||G|)$, ya que todos los conjuntos son de tamaño $O(|G|)$, y las operaciones de unión, intersección, diferencia y concatenación se computan en tiempo lineal en la suma de los tamaños de los conjuntos. Basta con iterar todos los elementos de los conjuntos e ir chequeando si pertenece o no al conjunto de salida, donde sólo es necesario hacer $O(1)$ operaciones en cada paso. □

Nuestras hipótesis serán revisadas en el Capítulo 5, cuando expliquemos en detalle los algoritmos y las estructuras de datos en nuestra implementación.

4.1.2. Agregando la Recursión

Uno de los operadores que agregan una gran expresividad al lenguaje es la *recursión* (o cláusula reflexiva-transitiva). El operador unario $()^*$ se define como

$$\phi^* = \bigvee_{k \geq 0} \phi^k,$$

donde $\phi^0 = \varepsilon$ y $\phi^k = \phi \cdot \phi ... \cdot \phi$ concatenado k veces. Esta definición no es factible en la práctica, ya que no es posible realizar una cantidad infinita de operaciones. Más aún, el objetivo final es que el tiempo de evaluación de $\lambda(\phi^*, V_0)$ sea lineal en el tamaño del grafo.

Para resolver este problema, vamos a realizar un análisis del comportamiento de la recursión dentro de la función de evaluación λ. Las siguientes propiedades de λ nos van a permitir desarrollar nuestro algoritmo:

Observación 4.1.1 La función de evaluación λ cumple las siguientes equivalencias:

- $\lambda(\bigcup_{k \geq 0} \phi^k, V_0) = \bigcup_{k \geq 0} \lambda(\phi^k, V_0)$.
- $\lambda(\phi^k, V_0) = \lambda(\phi \cdot \phi^{k-1}, V_0) = \lambda(\phi, \lambda(\phi^{k-1}, V_0))$.

Observación 4.1.2 Si definimos $V_k = \lambda(\phi, V_{k-1})$, entonces se cumple la siguiente igualdad

$$\lambda(\phi^*, V_0) = \bigcup_{k\geq 0} V_k = V_0 \cup \lambda(\phi, V_0) \cup \lambda(\phi, V_1) \cup \lambda(\phi, V_2) \cup .... .$$

Como podemos ver, computar la recursión de ϕ se puede reducir a ejecutar $\lambda(\phi, *)$ y realizar uniones. Para resolver este problema, vamos a definir conjuntos auxiliares que nos llevaran a la solución.

Definición 4.1.2 Se definen de forma inductiva los siguientes conjuntos:

- $\bar{V}_0 = \Sigma_0 = V_0$.
- $\bar{V}_k = \lambda(\phi, \bar{V}_{k-1}) \setminus \Sigma_{k-1}$.
- $\Sigma_k = \Sigma_{k-1} \cup \bar{V}_k$.

Necesariamente se cumple que $\bar{V}_k = \emptyset$ para todo $k \geq n$ para algún $n \leq |V|$, y de este modo se obtiene que

$$\lambda(\phi^*, V_0) = \bigcup_{k\geq 0} V_k = \bigcup_{k\geq 0} \bar{V}_k = \Sigma_{n-1}.$$

Por definición, los $\bar{V}_k$ son disjuntos entre sí, por lo que cada vez se ejecuta $\lambda(\phi, *)$ sobre un conjunto de nodos que en los pasos anteriores no estaban presentes, recorriendo así sólo una vez el conjunto de nodos V y computando el conjunto $\lambda(\phi^*, V_0)$ de forma eficiente.

4.1.3. Operaciones a Implementar

Por la definición del lenguaje, las consultas de largo 1 son la base del lenguaje, ya que el resto de las consultas se construyen a partir de éstas, por lo que deben ser computadas de la forma más eficiente posible. Las consultas a considerar son:

1. $\lambda(\top, V_0) = V_0$.
2. $\lambda([\downarrow v]\ , V_0) = \{u \in V_0 |\ u = v\}$.
3. $\lambda([\sigma = k], V_0) = \{u \in V_0 |\ \sigma(v) = k\ \}$.
4. $\lambda(c, V_0) = \{u \in V |\ \exists v \in V_0$ tal que $(u, c, v) \in E\}$.
5. $\lambda(c^-, V_0) = \{u \in V |\ \exists v \in V_0$ tal que $(v, c, u) \in E\}$.

Resolver el caso 1 es trivial, al igual que el caso 2, donde sólo es necesario verificar si un nodo pertenece a un conjunto de nodos. El caso 3 se obtiene al filtrar el conjunto V_0 según el valor que toma su atributo. Los casos 4 y 5 son los más complejos, que trataremos a continuación.

La idea de computar $\lambda(c, V_0)$, es que dado un conjunto de nodos V_0, se busca computar el conjunto de nodos V_1 que corresponden a los "vecinos por c"de V_0. Es decir, $V_1 = \lambda(c, V_0)$ ssi para todo nodo $u \in V_1$ existe un nodo $v \in V_0$ tal que el arco $(u, c, v) \in E$. Para computar V_1 tenemos dos opciones:

1. *Recorrer todos los nodos* $u \in V$

   Si existe un arco de la forma (u, c, v) *tal que* $v \in V_0$

   Insertar u *al conjunto* V_1

2. *Recorrer todos los nodos* $v \in V_0$

   Para todos los arcos de la forma (u, c, v)

   Insertar u *al conjunto* V_1

Vemos que el primer método se basa en chequear a cada nodo si cumple la propiedad de que salga un arco etiquetado con c hacia algún nodo del conjunto V_0, mientras que el segunda método se basa en unir todos los vecinos, a través de arcos etiquetado con c, de todos los nodos en V_0. El primer método es más eficiente cuando V_0 es *grande*, ya que en el caso extremo que $V_0 = V$ sólo debo revisar 1 arco por nodo tomando tiempo $O(|V|)$, mientras que el segundo método es más eficiente cuando V_0 es *pequeño,* ya que en el caso extremo que $V_0 = \{v\}$ sólo debo recorrer los vecinos de 1 nodo tomando tiempo $O(1)$ si la cantidad de vecinos es constante o $O(|V|)$ en el peor caso.

El caso de computar $\lambda(c^-, V_0)$ es análogo al caso anterior, sólo que se invierte la dirección de los arcos, es decir $V_1 = \lambda(c^-, V_0)$ ssi para todo nodo $u \in V_1$ existe un nodo $v \in V_0$ tal que el arco $(v, c, u) \in E$. Las operaciones necesarias para implementar los algoritmos anteriores son los siguientes:

1. Dado un nodo u obtener el valor de su atributo $\sigma(u)$.
2. Iterar un conjunto de nodos.
3. Dada una etiqueta c, recorrer los arcos de la forma (u, c, v).
4. Dada una etiqueta c y un nodo v, recorrer los arcos de la forma (u, c, v).
5. Verificar si un nodo v pertenece a un conjunto de nodos.
6. Agregar un nodo u a un conjunto de nodos.

De aquí se deducen las operaciones que deben implementar las distintas estructuras de datos, que representan cada uno de las componentes del grafo.

4.2. Almacenamiento de Datos

Durante todo el capítulo, hemos hecho la suposición que tenemos a nuestra disposición las estructuras de datos necesarias para evaluar nuestro lenguaje de consulta. A continuación vamos a revisar una de las mayores restricciones computacionales, que nos dará todas las directrices a seguir por el resto de la implementación, incluyendo el diseño de las estructuras de datos necesarias.

4.2.1. Restricción de Memoria

Nuestro modelo computacional tiene la restricción que existen dos tipos distintos de memoria, las cuales tienen distintas propiedades y características. Describamos a continuación estos recursos computacionales.

Memoria Principal La memoria principal, comúnmente conocida como Memoria *RAM* (del inglés *Random Access Memory*), es la memoria de trabajo, ya que es no persistente, de tamaño limitado y tiene un gran rendimiento. Esto quiere decir que los datos no permanecen luego que el sistema es apagado, el espacio que puede almacenar es del orden del número de nodos del grafo (incapaz de almacenar todo el grafo), y la velocidad de lectura y escritura es muy alta.

Memoria Externa La memoria externa, comúnmente conocida como memoria secundaria o simplemente disco, es la memoria de almacenamiento, ya que es persistente, de tamaño ilimitado y de bajo rendimiento. Esto quiere decir que los datos permanecen almacenados incluso si el sistema es apagado, el espacio que puede almacenar es del orden del grafo, pero la velocidad de lectura y escritura es muy baja.

La gran mayoría de los algoritmos para grafos en la literatura consideran que es posible almacenar todo el grafo en memoria principal, y es ahí donde se realizan todas operaciones. Esto no es posible en la práctica, ya que si consideramos grafos de millones de nodos con miles de millones de arcos, el espacio ocupado puede llegar a ser varios TeraBytes, mientras que las memorias RAM sólo llegan a decenas de GigaBytes, es decir, cientos de veces de diferencia. Además, como el grafo es dinámico, puede ir aumentando su tamaño considerablemente a lo largo del tiempo.

Por este hecho, durante toda la implementación se va a considerar que el grafo es almacenado en disco y que en memoria principal sólo pueden ser almacenados conjuntos de nodos. Dicho de otro modo, el tamaño de la memoria principal es de $O(|V|)$ y el tamaño de la memoria externa es de $O(|G|)$, donde $|G| = O(|V| + |E|)$ es el tamaño del grafo, y $|V|$ es la cantidad de nodos del grafo. Esta restricción aumenta en gran medida la complejidad del diseño de algoritmos y estructuras de datos eficientes, ya que para recorrer el grafo se debe acceder a la memoria externa, y si este acceso no se hace de forma óptima, el rendimiento del sistema se ve fuertemente perjudicado. Estudiar algoritmos que permitan resolver nuestro problema, considerando que no es posible almacenar conjuntos de tamaña $O(|V|)$ memoria principal, es parte del trabajo futuro, mientras que nos concentraremos a diseñar algoritmos bajo nuestras suposiciones.

4.2.2. Optimizando el Acceso a Disco

Una de las operaciones más sensibles para obtener un buen rendimiento global es la forma de acceder al disco duro. El método de lectura de datos más simple a implementar es que, con cada acceso al disco duro se lee un sólo *byte* de datos, obteniendo el peor rendimiento de todos, alcanzando un velocidad de lectura de 200 *KiloBytes/seg.* Si consideramos una base de datos de 1 *GigaBytes*, este método puede llegar a demorar 85 minutos en leer la base de forma secuencial, lo cual es impensable en cualquier aplicación real.

La razón porque el método anterior es tan ineficiente es que cada acceso a disco es muy caro computacionalmente, y el disco duro lee varios *bytes* consecutivos de forma simultánea en forma de bloques, por lo que la forma más óptima de leer datos es guardar en memoria principal la mayor cantidad de *bytes* posible. El método más eficiente de lectura es leer un bloque de datos y guardarlos en un *buffer* en memoria principal, para luego leer los datos desde el *buffer* a medida que es requerido. Es importante conocer cuáles son los datos que están en el *buffer*, para que se vaya actualizando con nuevos datos cuando se quiera acceder a un nuevo bloque de datos. Cabe mencionar que la lectura debe realizarse de forma secuencial, lo cual es una restricción importante al momento de implementar los algoritmos y las estructuras de datos finales.

Para lograr la mejor velocidad de lectura posible, se realizaron múltiples experimentos, implementando diferentes métodos de lectura del disco, siguiendo la metodología descrita en [34], basada en implementar más de 10 métodos distintos, e ir ejecutando un conjunto de prueba variando los parámetros, para así obtener un óptimo experimental. Los experimentos muestran que es posible alcanzar velocidades de lectura de 250 *MBytes/seg*, mejorando el rendimiento más de 1000 veces que el método inicial, logrando así leer una base de datos de 1 *GigaBytes* en 4 segundos. El óptimo experimental se obtuvo con un *buffer* de 128 *KiloBytes* como podemos ver en la Figura 5.1.

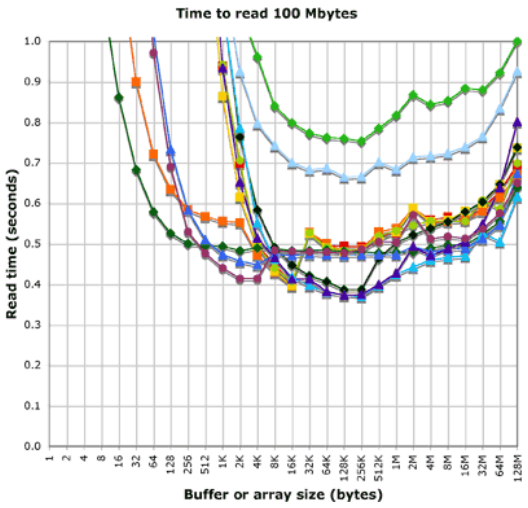

Figura 4.1: Tiempo de lectura vs tamaño de *Buffer*.

Los experimentos fueron ejecutados en un notebook con sistema operativo Windows 7 Profesional de 32-bits, con 4 GB de memoria RAM (3.45 GB utilizable) y un procesador Intel Core 2 Duo P9400 de 2.40 GHz. Los resultados pueden variar al utilizar otra configuración de sistema operativo y de hardware, pero las conclusiones generales se deben mantener.

Con la metodología recién descrita se implementó en *Java* la clase *Disk*, que se preocupa del acceso a disco y del uso del *buffer* para un archivo, implementando los métodos necesarios para la escritura y lectura de datos de la forma secuencial. Es posible obtener un mayor rendimiento si es que se implementa este método de lectura en algún lenguaje de programación más eficiente, por ejemplo *C/C++*.

4.2.3. Estructura de Datos para los Arcos

Recordemos que como los grafos son dinámicos, la cantidad de arcos por nodo puede estar continuamente creciendo, por lo que las vecindades no pueden estar acotadas por ningún número. La estructura de datos ideal para almacenar los arcos del grafo es la siguiente: por cada etiqueta c, se tiene un arreglo de tamaño $|V|$, donde cada elemento del arreglo es una lista enlazada que contiene a los vecinos del nodo correspondiente. Esta estructura no puede ser implementada de forma explícita en memoria externa, pero puede ser emulada de la siguiente forma:

1. Los nodos son representados por enteros del tipo *long* (de 64 bits).
2. Por cada etiqueta *label* se crean dos archivos: *label.gdb* y *label.gdb.aux*. Estos archivos se componen de lineas de tamaño fijo (inicializadas con valores 0), no necesariamente del mismo tamaño, pero recomendando que el tamaño de *label.gdb* sea menor que el de *label.gdb.aux*.
3. La k-ésima línea del archivo *label.gdb* almacenará los datos relacionados al nodo k-ésimo. Esto permite saber en $O(1)$ que línea del archivo se debe leer para extraer la vecindad del nodo k.
4. Cada línea del archivo *label.gdb* tendrá n *longs* consecutivos, donde el primer y último *long* se usarán como identificador del nodo y un puntero al archivo auxiliar, respectivamente. Los *n-2 longs* restantes se usarán para almacenar los vecinos del respectivo nodo. En nuestra implementación consideramos 8 *longs*.
5. Una vez que estos *n-2 longs* estén ocupados, se creará una nueva línea vacía de tamaño m al final del archivo *label.gdb.aux,* asignando el puntero del archivo *label.gdb* a la posición de esta nueva linea. En nuestra implementación consideramos 128 *longs*.
6. Los *m-1* primeros *longs* se usarán para almacenar los vecinos del respectivo nodo, y el último *long* se usará como puntero de la misma forma que en el archivo *label.gdb.*

El Cuadro 4.1 representa un ejemplo de los archivos *label.gdb* y *label.gdb.aux* .

Key	1	2	...	6	P
1	10	0	...	0	0
2	6	3	...	12	1
⋮					

1	2	3	...	127	P
10	9	6	...	5	2
4	13	0	...	0	0
⋮					

Cuadro 4.1: Ejemplo de *label.gdb* y de *label.gdb.aux*

Como podemos observar, esta estructura implementa en memoria externa un *HashMap Multivaluado*, ya que por cada *long* puede almacenar una cantidad arbitraria de *longs*, permitiendo extraer para un nodo fijo $\bar{u}$ el conjunto de nodos v para los que existe un arco de la forma $(\bar{u}, c, v)$, minimizando la cantidad de accesos a disco.

Para el caso de las consultas de la forma $\phi = c^-$, se procede de forma análoga, sólo que por cada etiqueta $label^-$ se crean los archivos *label_ .gdb* y *label_ .gdb.aux,* siguiendo la descripción anterior.

4.2.4. Estructura de Datos para los Atributos

Las consultas de la forma $\phi = \top$ y $\phi = [\downarrow v]$ son muy simples de resolver, ya que sólo es necesario guardar en memoria la cantidad de nodos del grafo. Las consultas de la forma $\phi = [\sigma = k]$ requieren una estructura de datos distinta que detallaremos a continuación.

Atributos del tipo Long

Nuestro lenguaje de consulta debe implementar la selección de conjuntos de nodos donde el valor de su atributo es igual a un valor fijo. Vamos a considerar inicialmente que los valores que puede tomar un atributo son enteros positivos (*longs*). Luego, nuestra estructura de datos debe almacenar para cada nodo el valor que toma su atributo, y análogamente, por cada valor de atributo debe almacenar la lista de nodos en que su atributo coincide con dicho valor. La estructura que permite almacenar estos datos en memoria externa es muy similar a la estructura para almacenar arcos, sólo que está optimizada de la siguiente forma:

1. Los nodos y los valores son representados por enteros del tipo *long* (de 64 bits).
2. Por cada atributo *label* se crean tres archivos: *label.attr, label_.attr* y *label_.attr.aux*. Estos archivos se componen de lineas de tamaño fijo, inicializadas con valores 0.
3. La k-ésima línea del archivo *label.attr* almacenará el valor del atributo del nodo k-ésimo.
4. Cada línea del archivo *label.attr* tendrá 2 *longs* consecutivos, donde el primer *long* es el identificador del nodo y el segundo *long* será el valor del atributo de dicho nodo.
5. La k-ésima línea del archivo *label_.attr* almacenará los nodos que tengan en su atributo el valor k.
6. Cada línea del archivo *label_.attr* tendrá n *longs* consecutivos, donde el primer y último *long* se usarán como el valor k y un puntero al archivo auxiliar, respectivamente. Los $n - 2$ *longs* restantes se usarán para almacenar la lista de nodos que tengan en su atributo el valor k. En nuestra implementación consideramos 8 *longs*.
7. Una vez que estos *n-2 longs* estén ocupados, se creará una nueva linea vacía de tamaño m al final del archivo *label_.attr.aux,* asignando el puntero del archivo *label_.attr* a la posición de esta nueva linea. En nuestra implementación consideramos 32 *longs*.
8. Los *m-1* primeros *longs* se usarán para almacenar la lista de los nodos que tengan en su atributo el valor fijado desde el archivo *label_.attr*, y el último *long* se usará como puntero de la misma forma que en el archivo *label_.attr*

Como podemos observar, esta estructura implementa en memoria externa, por un lado un *HashMap* que nos entrega en $O(1)$ el valor en el atributo de cada nodo, y por otro lado un *HashMap Multivaluado*, ya que por cada *long* se puede almacenar una cantidad arbitraria de *longs*, permitiendo extraer para un valor fijo k el conjunto de nodos v que tienen en su atributo el valor k.

Atributos del tipo String

Vamos a extender nuestra estructura de datos para permitir que los atributos almacenen valor es del tipo *String*. Es necesario diferenciar los atributos de tipo *Long* de los atributos de tipo *String*, ya que el manejo de ambos casos es distinto. Para lograr esto vamos a almacenar en disco el tipo de cada atributo en un archivo llamado *_esq.sch,* y como es de tamaño a lo más $O(|V|)$, podemos almacenar esta información en memoria principal y discriminar en tiempo $O(1)$.

Para el caso de los atributos de tipo *String*, la estructura de datos es exactamente igual al caso de tipo *Long*, agregando un paso intermedio: a cada *String* vamos a asignar un *long* correlativo, almacenando este diccionario en el archivo *_text.sch*. Según se agreguen nuevas palabras este diccionario va creciendo, en una fila por cada nueva palabra, y si consultamos por una palabra ya agregada anteriormente, nos debe devolver el *long* asignado.

1. Cada carácter de un *String* ocupará 2 *bytes.*
2. Cada fila de *_text.sch* tendrá una extensión de m *bytes*. En nuestra implementación consideramos 256 *bytes.*
3. Los primeros 8 *bytes* almacenarán el *long* identificador del *String.*
4. Los $m - 8$ *bytes* restantes se usarán para almacenar el *String.* Nuestra implementación permite *Strings* de hasta 124 caracteres

Podemos notar que este diccionario sólo se ocupa en la etapa de preprocesamiento de la consulta, donde se busca el *long* identificador del *String* en la consulta, para luego proceder como un atributo del tipo numérico. Esto implica que el rendimiento de ejecución de consultas no se ve afectado al considerar atributos del tipo *String*. Además, se puede almacenar este diccionario en memoria principal, permitiendo encontrar el *long* identificador de un *String* de forma más eficiente.

4.3. Especificación de los Objetos

4.3.1. Representación de Conjuntos

Mientras que en memoria externa se almacena todo el grafo, en memoria principal se debe almacenar los conjuntos de nodos que son los resultados de las consultas, los cuales son representados como conjuntos de *longs*. La clase *Nodes* debe implementar las siguientes operaciones:

- Insertar, consultar y eliminar elementos al conjunto.
- Iterar todos los elementos del conjunto.
- Unir, intersectar y restar conjuntos.

Todas estas operaciones deben hacerse en tiempo lineal, ya que de lo contrario la implementación no será óptima. La estructura de datos escogida como representación interna es un *HashSet*, estructura que representa un conjunto de elementos utilizando una *tabla hash* como estructura interna, y el costo promedio de insertar, consultar y eliminar un elemento es de $O(1)$, mientras que las operaciones de conjuntos se pueden implementar en tiempo lineal gracias a que el iterador es lineal. La clase *Nodes* debe proveer como mínimo los siguientes métodos :

- *add(long n)* : Inserta al conjunto el nodo *n*.
- *remove(long n)* : Elimina del conjunto el nodo *n*.
- *contains(long n)* : Devuelve *true* si el nodo *n* pertenece al conjunto; devuelve *false* en caso contrario.
- *init()* : Inicializa el iterador del conjunto.
- *notNull()* : Verifica que el iterador apunte a un nodo.
- *getNode()* : Devuelve el nodo apuntado por el iterador.
- *next()* : El iterador apunta al siguiente nodo del conjunto.

4.3.2. Requerimientos del Grafo

El grafo de la base de datos será implementado por la clase *Graph*, la cuál tendrá acceso a toda la información almacenada, además de tener la capacidad de definir y manipular datos. Los métodos que debe implementar la clase *Graph*, para proveer de todo lo necesario para resolver las consultas de nuestro lenguje, son los siguientes:

- *newNode()* : Agrega un nuevo nodo al grafo.
- *countNodes()* : Devuelve la cantidad de nodos en el grafo.

Los nodos podrán almacenar valores en sus atributos, que podrán ser del tipo *Long* o del tipo *String*. Para incluir el manejo de atributos, los métodos a implementar son los siguientes:

- *setAttribute(long n,String attr,Obj o)* : Asigna para el nodo *n* el valor del atributo *attr* con el valor *o*.
- *getAttribute(long n,String attr)* : Devuelve para el nodo *n* el valor del atributo *attr*.
- *getNodosWhere(String attr,String o)* : Devuelve el conjunto de nodos tal que el valor del atributo *attr* sea el valor *o*.

Los arcos del grafo serán representados como tuplas de la forma *(String label,long n1,long n2)*, que se interpreta como un arco con etiqueta *label* desde el nodo *n1* hasta el nodo *n2*. Los métodos a implementar relacionados con los arcos son:

- *addEdge(String label,long n1,long n2)* : Inserta un arco con etiqueta *label* desde el nodo *n1* hasta el nodo *n2.*
- *getTailsWithHeadsIn(String label,Nodes Heads)* : Devuelve el conjunto de todos los nodos que tengan algún arco de salida, con etiqueta *label,* hasta algún nodo en el conjunto *Heads.*
- *getHeadsWithTailsIn(String label,Nodes Tails)* : Devuelve el conjunto de todos los nodos que tengan algún arco de entrada, con etiqueta *label,* desde algún nodo en el conjunto *Tails.*

4.3.3. Representación de las Consultas

Para poder ejecutar alguna consulta en nuestra base de datos, primero debemos definir una representación interna de las consultas, que debe estar bien definida y sin ambigüedad. Las clases que representarán las consultas son 3: *Predicate*, *Operation* y *Query.* Revisemos la estructura y los métodos implementados de cada una de estas clases.

Operation

La clase *Operation* es bastante simple, ya que su función es representar las operaciones que soporta el lenguaje, y discriminar según su tipo. Para esto, la clase tiene una sola variable interna, que guardará el simbolo de la operación, y un método por cada operación: *isConverse(), isKleene(), isNesting(), isNot(), isConcat(), isOr(), isAnd()* y *is Minus().* Cada uno de estos métodos devolverá *true* si el objeto representa la operación en cuestión, de lo contrario devolverá *false.* Adicionalmente, se implementan los métodos *isUnary()* y *isBinary()*, que discrimina si la operación representada es unaria o binaria.

Predicate

La clase *Predicate* representa el tipo de consultas atómicas, las cuales no pueden descomponerse en otras más simples. Tiene implementado varios constructores para cada uno de los tipos de consultas atómicas:

- *Predicate(boolean bool)* : Representa la consulta $\top$ (*true*) o $\bot$ (*false*).
- *Predicate(long node)* : Representa la consulta $[\downarrow v]$, donde *node* es el *id* del nodo.
- *Predicate(String attr,long v)* : Representa la consulta $[attr = v]$, donde *attr* es el atributo y v el valor.
- *Predicate(String label)* : Representa la consulta c, donde *label* es la etiqueta.

Esta clase también implementa los métodos que discriminan el tipo de consulta que representa: *isTrue(), isFalse(), isSingleton(), isCondition()* y *isLabel().*

Query

La clase *Query* es la representación interna de todas las consultas de nuestro lenguaje. Esta clase modela las consultas como árboles, donde las hojas del árbol son objetos del tipo *Predicate*, y cada nodo interno está compuesto por un objeto *Operation* y 1 o 2 objetos *Query*, dependiendo si la operación es unaria o binaria. Los constructores son:

- *Query(Predicate pre)* : Representa las consultas atómicas.
- *Query(Operation op, Query izq)* : Representa las consultas con una operación unaria.
- *Query(Operation op, Query izq, Query der)* : Representa las consultas con una operación binaria.

El método que computa el conjunto de nodos que satisfacen la consulta en una *GDB* es

public Nodes ejecutar(Graph g) ,

cuya implementación vamos a explicar en el Capítulo 5.

Capítulo 5

Implementación del Sistema

En el presente capítulo vamos a detallar la implementación realizada, explicando de la mejor manera los algoritmos y métodos programados. Cabe mencionar que la implementación se llevó a cabo utilizado el lenguaje de programación Java, y que todos los experimentos fueron ejecutados en un computador con sistema operativo Windows 7 Profesional de 32-bits, con 4 GB de memoria RAM y un procesador Intel Core 2 Duo P9400 de 2.40 GHz.

5.1. Conjuntos de Nodos

5.1.1. Representación Interna

Tal como especificamos anteriormente, en nuestra implementación los nodos son identificados por números de tipo *long*, por lo que un conjunto de nodos se representa como un conjunto de *longs*. Existen diferentes implementaciones en Java de conjuntos de elementos, por lo que se procedió a realizar experimentos con el fin de utilizar la librería más eficiente. En los experimentos se evaluaron 5 librerías distintas, midiendo la velocidad de insertar, consultar y eliminar 1.000.000 *longs*. El Cuadro 5.1 muestra una tabla con los tiempos de ejecución de las 5 librerías en las 3 operaciones.

seg / $1x10^6$ ops	Insertar	Consultar	Eliminar
TreeSet	3.2	0.31	0.54
HashSet	1.72	0.16	0.22
TLongHashSet	1.72	0.11	0.19
fastutil	0.21	0.01	0.37
hppc	1.4	0.13	0.32

Cuadro 5.1: Tiempos de Ejecución de las distintos Librerías de Conjuntos de *Longs*

La implementación que obtuvo el mejor rendimiento fue la clase *LongOpenHashSet* de la librería Open Source *fastutil* [18], ya que su diseño está optimizada para representar conjuntos de *longs.* A pesar que esta clase no es la más rápida para eliminar elementos, esa operación tiene mucho menos importancia en nuestros algoritmos que insertar o consultar elementos. Esta clase nos permite insertar/consultar *longs* a una velocidad promedio del orden de 8×10^6 *longs/seg,* tiempo más que suficiente para nuestros fines.

5.1.2. Operaciones de Conjuntos

Las operaciones de conjuntos que se deben implementar en la clase *Nodes* son la *unión, intersección* y *diferencia* de conjuntos. El complemento se obtiene a partir de la *diferencia* y del conjunto de todos los nodos. Para obtener el mayor rendimiento posible, se consideraron las siguientes hipótesis:

1. Los objetos iniciales pueden ser destruidos o modificados, pero el resultado debe ser correcto. Esta hipótesis se consideró para utilizar menos memoria principal.

2. Basta iterar uno de los conjuntos, por lo que siempre se elige el que tiene menos elementos. La variable *this* hace referencia al conjunto actual.

3. Se implementó el método *isUniverse()*, que nos permite discriminar si alguno de los conjuntos es el conjunto de todos los nodos del grafo, ya que el resultado se puede calcular fácilmente.

El método *union* procede de la siguiente forma:

1. Si X es el conjunto de todos los nodos, se retorna X, ya que la unión de X con cualquier otro conjunto sigue siendo igual.

2. Si *this* es el conjunto de todos los nodos, se retorna *this*, ya que la unión de *this* con cualquier otro conjunto sigue siendo igual.

3. Si X tiene menos elementos que *this*, se itera el conjunto X y se agrega al conjunto *this*. Luego, se retorna *this* que es la unión.

4. Si *this* tiene menos elementos que X, se itera el conjunto *this* y se agrega al conjunto X. Luego, se retorna X que es la unión.

Código Unión

```
public Nodes union(Nodes X){
    if(X.isUniverse())
        return X;
    else if(this.isUniverse())
        return this;
    else if(X.count()<this.count()){
        for(X.init();X.notNull();X.next())
            this.add(X.getNodo());
        return this;
    }else{
        for(this.init();this.notNull();this.next())
            X.add(this.getNodo());
        return X;
    }
}
```

El método *interseccion* procede de la siguiente forma:

1. Si X es el conjunto de todos los nodos, se se retorna *this* ya que intersectarlo con X no altera el conjunto.

2. Si *this* es el conjunto de todos los nodos, se se retorna X ya que intersectarlo con *this* no altera el conjunto.

3. Si X tiene menos elementos que *this*, se itera el conjunto X y se eliminan todos los elementos que no pertenezcan a *this*. Luego se retorna X que es la intersección.

4. Si *this* tiene menos elementos que X, se itera el conjunto *this* y se eliminan todos los elementos que no pertenezcan a X. Luego se retorna *this* que es la intersección.

Código Intersección

```
public Nodes interseccion(Nodes X){
    if(X.isUniverse())
        return this;
    else if(this.isUniverse())
        return X;
    else if(X.count()<this.count()){
        for(X.init();X.notNull();X.next()){
            if(!this.contains(X.getNodo()))
                X.remove(X.getNodo());
        }
        return X;
    }else{
        for(this.init();this.notNull();this.next()){
            if(!X.contains(this.getNodo()))
                this.remove(this.getNodo());
        }
        return this;
    }
}
```

El método *diferencia* procede de la siguiente forma:

1. Si X es el conjunto de todos los nodos, se retorna un conjunto vacío.

2. Si X tiene menos elementos que *this*, se itera el conjunto X y se eliminan todos los nodos del conjunto *this*. Luego se retorna *this* que es la diferencia.

3. Si *this* tiene menos elementos que X, se itera el conjunto *this* y se eliminan todos los nodos del conjunto que pertenezcan a X. Luego se retorna *this* que es la diferencia.

Código Diferencia

```
public Nodes diferencia(Nodes X){
    if(X.isUniverse())
        return new Nodes();
    else if(X.count()<this.count()){
        for(X.init();X.notNull();X.next())
            this.remove(X.getNodo());
        return this;
    }else{
        for(this.init();this.notNull();this.next()){
            if(X.contains(this.getNodo()))
                this.remove(this.getNodo());
        }
        return this;
    }
}
```

Para el complemento, se implementó la subclase *NodesU*, que nos permite iterar todos los nodos del grafo, además que implementa las operaciones entre conjuntos de forma mucho más eficiente. El método *complemento* procede de la siguiente forma:

1. Se crea un conjunto vacío *Out* que será la salida, y un conjunto *NodesU* para iterar todos los nodos del grafo.

2. Si *this* no contiene el nodo actual de *NodesU*, entonces se agrega a *Out.*

3. Retornar *Out.*

Código Complemento

```
public Nodes complemento(){
    Nodes out=new Nodes();
    NodesU X=new NodesU();
    for(X.init();X.notNull();X.next(){
        if(!this.contains(X.getNodo()))
            out.add(X.getNodo());
    }
    return out;
}
```

5.2. Consultando el Grafo

5.2.1. Selección de Nodos por Atributo

El método *getNodosWhere(label,val)* es la implementación de las consultas atómicas de la forma $[label = k]$. En resumen, este método lee el archivo *label_.attr* en la línea que corresponde al *id* del valor *val*. Se van agregando los nodos al conjunto de salida, accediendo al archivo *label_.attr.aux* siguiendo los punteros de forma ordenada. El pseudo-algoritmo es el siguiente:

1. Obtener el *id* de *val.*
   a) Si el atributo *label* es de tipo *Long,* entonces *id=val.*
   b) Si el atributo *label* es de tipo *String,* buscar *val* en *text.sch* y obtener el *id.*
2. Leer la línea *id* del archivo *label_.attr.*
   a) Agregar los *n-2* valores de la línea *id* al conjunto de salida.
   b) Si el puntero *P>0*, entonces ir a *(3).*
3. Leer la *P-ésima* línea del archivo *label_.attr.aux.*
   a) Agregar los *m-1* de la línea *P* al conjunto de salida.
   b) Si el puntero *P>0*, entonces ir a *(3).*

5.2.2. Vecindad de un Conjunto de Nodos

Como habíamos visto anteriormente, el método *getTailsWithHeadsIn(String label,Nodes Heads)* devuelve el conjunto de todos los nodos que tengan algún arco de salida, con etiqueta *label,* hasta algún nodo en el conjunto *Heads.* Este método juega un papel muy importante en nuestra implementación, ya que es la base del lenguaje de consultas para expresar caminos.

Una primera idea para implementar este método es seguir los pasos descritos al principio del Capítulo 4, que podriamos reescribir de la siguiente forma:

1. *Para cada línea K del archivo label.gdb.*
   a) *Para cada long N en la línea actual.*
      1) *Si N pertenece al conjunto Heads.*
         a′ Agregar el nodo K al conjunto Output.
         b′ Seguir con la siguiente línea de (1).
   b) *Leer el puntero P de la línea actual.*
      1) *Si el puntero P>0, ir a (2).*
      2) *Si el puntero P=0, ir a (1).*

2. *Leer la línea P del archivo label.gdb.aux.*

   a) Para cada long M en la línea actual.

      1) *Si M pertenece al conjunto Heads.*

         a′ Agregar el nodo K al conjunto Output.

         b′ Seguir con la siguiente línea (1).

   b) Leer el puntero P de la línea actual.

      1) *Si el puntero P>0, ir a (2).*
      2) *Si el puntero P=0, ir a (1).*

Este pseudo-algoritmo implementa de forma correcta el método *getTailsWithHeadsIn*, pero en la práctica no es eficiente. Si los punteros P en los archivos *label.gdb* y *label.gdb.aux* estuviesen ordenados de tal forma que el acceso a disco fuese secuencial, entonces el pseudo-algoritmo anterior entregaría la eficiencia requerida.

Desgraciadamente, el orden de los datos depende del orden de inserción, el cual no se puede saber a priori y se supone desordenada, produciendo casos como el del Cuadro 5.2.

Key	1	2	3	4	5	6	P
1	10	2	7	8	6	9	1
2	6	3	2	13	14	12	2

1	2	3	...	127	P
4	9	6	...	5	3
10	13	5	...	2	4
13	...	...	...	...	...
7	...	...	...	...	...

Cuadro 5.2: Ejemplo de label.gdb y label.gdb.aux en una insercción desordenada.

Podemos observar en esta estructura, que la primera línea de *label.gdb* hace referencia a la primera línea de *label.gdb.aux*, mientras que ésta a su vez hace referencia a la tercera línea. De forma similar, la segunda línea de *label.gdb* hace referencia a la segunda línea de *label.gdb.aux*, mientras que ésta a su vez hace referencia a la cuarta línea. Esto produce que la secuencia de lectura de las líneas del archivo *label.gdb.aux* sea *1-3-2-4*, resultando una lectura no secuencial, por lo cual no es óptima.

5.2.3. Implementar una Lectura Secuencial

Para poder resolver el problema de lectura no secuencial, cabe destacar los siguientes puntos:

- Los punteros P del archivo *label.gdb* pueden estar completamente desordenados.
- Los punteros P del archivo *label.gdb.aux* tienen un orden parcial, ya que para toda línea K que tiene un puntero P se cumple que $P>K$.
- El orden de inserción de los nodos en el conjunto de salida no altera el resultado final.

Gracias a estos puntos, es posible hacer una modificación al algoritmo que permite leer de forma secuencial ambos archivos, obteniendo así una *complejidad de evaluación óptima débil*. La idea del algoritmo es tener en memoria principal una estructura de datos, que almacenará los accesos *pendientes* a las lineas del archivo *label.gdb.aux,* de forma de recorrerlo al final de forma ordenada. La estructura *Map* será la encargada de esta tarea, y debe permitir las siguientes operaciones:

- Insertar pares de la forma *(key,value)* en todo momento.
- Extraer el par *(key,value)* donde *key* es el menor valor en la estructura.

La implementación que utilizamos es la clase *Long2LongOpenHashMap* de la librería Open Source *fastutil* [18], ya que está optimizada para representar mapeos de *longs* a *longs.* Para implementar el iterador ordenado en la primera coordenada, se inicializa *key=1* y en cada paso se consulta a la estructura si contiene la llave *key*, para luego incrementar *key* en 1 hasta llegar al final. Esto permite extraer o agregar elementos a la estructura en todo momento, sin que se vea afectado el iterador. Denotando a esta estructura como *LazyP*, el pseudo-algoritmo para lograr una lectura secuencial es el siguiente:

1. Leer de forma secuencial todas las líneas de *label.gdb*, donde la *K-ésima* línea se corresponde con el *K-ésimo* nodo.

   a) Si alguno de los *n-2* valores pertenece al conjunto *Heads*, entonces agregar el nodo *K* al conjunto de salida; *continue.*

   b) Si ninguno de los *n-2* valores pertenece al conjunto *Heads* y el puntero *P=0*; *continue.*

   c) Si ninguno de los *n-2* valores pertenece al conjunto *Heads* y el puntero *P>0*, entonces agregar a *LazyP* el par *(P,K)*; *continue.*

2. Mientras la estructura *LazyP* tenga elementos, extraer el par *(P,K)* en orden creciente en *P,* y leer la *P-ésima* línea en el archivo *label.gdb.aux.*

   a) Si alguno de los *m-1* valores pertenece al conjunto *Heads*, entonces agrego el nodo *K* al conjunto de salida; *continue.*

   b) Si ninguno de los *m-1* valores pertenece al conjunto *Heads* y el nuevo puntero $\bar{P}=0$; *continue.*

   c) Si ninguno de los *m-1* valores pertenece al conjunto *Heads* y el nuevo puntero $\bar{P}>0$, entonces agregar a *LazyP* el par $(\bar{P},K)$; *continue.*

El algoritmo se divide en dos partes: en la primera parte se recorre el archivo *label.gdb* de forma secuencial, agregando a la salida los nodos que cumplan la condición y agregando a la estructura *LazyP* los accesos pendientes al archivo *label.gdb.aux*, mientras que en la segunda parte se recorre el archivo *label.gdb.aux* de forma secuencial, agregando a la salida los nodos que cumplan la condición e ir actualizando la estructura *LazyP*. De esta forma, para computar la salida del método *getTailsWithHeadsIn* se procede a

1. leer de forma secuencial los archivos *label.gdb* y *label.gdb.aux* de tamaño a lo más $|G|$,
2. realizar a lo más $|G|$ consultas de pertenencia e inserciones,

obteniendo así un algoritmo que toma tiempo $2|G|$, alcanzando el *óptimo débil* en términos *de complejidad de evaluación.* Análogamente, el método *getHeadsWithTailsIn(String label,Nodes Tails)* devuelve el conjunto de todos los nodos que tengan algún arco de entrada, con etiqueta *label,* desde algún nodo en el conjunto *Tails.* La implementación de este método es la misma que la anterior, sólo que se deben leer los archivos *label_.gdb* y *label_.gdb.aux* en vez de los archivos originales.

5.2.4. Algoritmo Alternativo

Analicemos el caso extremo donde el conjunto de nodos es vacío, es decir $V_0 = \{\}$. Para computar $\lambda(c, V_0)$ con el algoritmo anterior, cómo ningún nodo pertenece a V_0, se debe recorrer completamente los archivos *label.gdb* y *label.gdb.aux,* para finalmente entregar el conjunto vacío.

Una alternativa para computar este tipo de casos es usar los archivos asociados a la etiqueta inversa c^-, los cuales denotamos *label_.gdb* y *label_.gdb.aux.* Los datos que almacenan estos archivos es exactamente la misma que los otros, sólo que está transpuesta, permitiendo acceder rápidamente a todos los nodos u tal que exista un arco de la forma $(u, c, \bar{v})$, donde $\bar{v}$ es un nodo fijo. Luego, dado un conjunto V_0, se iteran todos los nodos del conjunto para luego insertar al conjunto de salida todos los nodos vecinos.

Cabe destacar que se debe tener cuidado de iterar los nodos de forma ordenada y creciente, para mantener un acceso a disco de forma secuencial y eficiente. El pseudo-algoritmo es el siguiente:

1. Iterar el conjunto de nodos *Heads* de forma creciente en el *id* del nodo *N.*
   a) Leer la *N-ésima* línea del archivo *label_.gdb.*
   b) Agregar los *n-2* valores de la línea *N* al conjunto de salida.
   c) Si el puntero *P>0*, entonces agregar a *LazyP* el par *(P,N)*; *continue.*
2. Mientras la estructura *LazyP* tenga elementos, extraer el par *(P,K)* en orden creciente en *P.*
   a) Leer la *P-ésima* línea del archivo *label.gdb.aux.*
   b) Agregar los *m-1* de la línea *P* al conjunto de salida.
   c) Si el puntero *P>0*, entonces agregar a *LazyP* el par *(P,N)*; *continue.*

Nuevamente el algoritmo se divide en dos partes: en la primera parte se recorre el archivo *label.gdb* de la forma creciente dada por el conjunto *Heads*, agregando a la salida todos los nodos vecinos y agregando a la estructura *LazyP* los accesos pendientes al archivo *label.gdb.aux*, mientras que en la segunda parte se recorre el archivo *label.gdb.aux* de forma creciente dada

por la estructura *LazyP*, agregando a la salida todos los nodos vecinos junto con ir actualizando la estructura *LazyP*. En este método también se leen los archivos de forma secuencial y realiza a lo más $|G|$ consultas de pertenencia e inserciones, obteniendo así otro algoritmo que toma tiempo $2|G|$, alcanzando el *óptimo débil* en términos *de complejidad de evaluación.*

5.2.5. Optimizando Casos de Borde

Dado que tenemos dos algoritmos distintos para solucionar el mismo problema, es natural intentar de *adivinar* cuál de los dos métodos será más eficiente, para obtener en todos los casos el mínimo.

En el caso de borde cuando $V_0 = \{\}$, el *Algoritmo1* recorre completamente los archivos, mientras que el *Algoritmo2* no necesita recorrer nada. El otro caso de borde es si $V_\phi = V$, en que el *Algoritmo1* recorre sólo el archivo *label.gdb*, mientras que el *Algoritmo2* recorre completamente *label_.gdb* y *label_.gdb.aux.*

Está claro que si tuvieramos una forma de calcular la función *T1_menor_T2(label,V),* que retorna *true* siempre y cuando el tiempo de ejecución del *Algoritmo1* es menos al tiempo de ejecución del *Algoritmo2*, entonces tendríamos un algoritmo más eficiente. La solución más rigurosa es calcular el tiempo de ejecución esperado de ambos algoritmos, en función del tamaño del grafo y de sus archivos.

Realizar este análisis riguroso no es estrictamente necesario, ya que el *Algoritmo1* es eficiente para conjuntos V grandes, mientras que el *Algoritmo2* es eficiente para conjuntos V pequeños, tal como se puede observar en la Figura 5.1. En nuestra implementación se consideró un umbral de decisión en función del tamaño del conjunto V, que fue definido experimentalmente como el 10 % del número de nodos, ya que lo más importante es mejorar el rendimiento en los casos en donde *Algoritmo1* presenta un mal desempeño, que corresponden a tamaños pequeños de V. Realizar un análisis más riguroso de los algoritmos y de su comportamiento es parte del trabajo futuro.

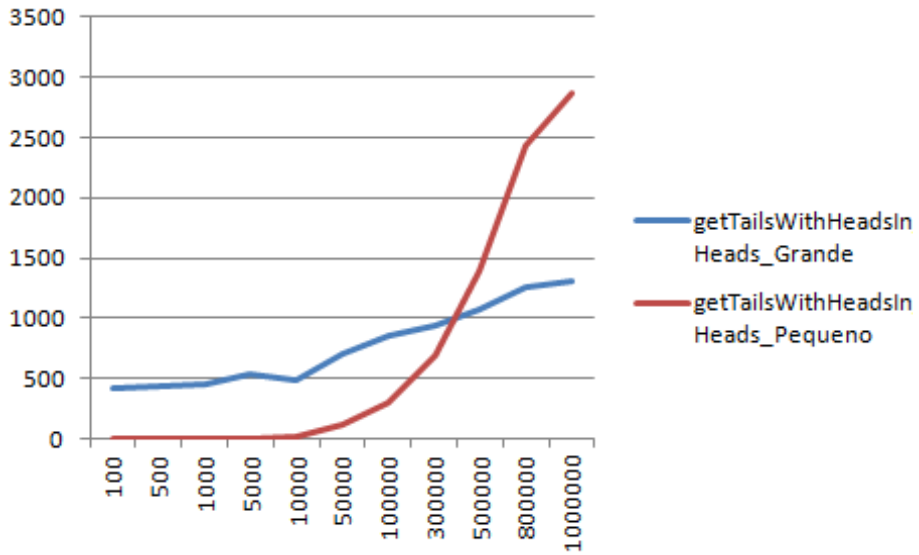

Figura 5.1: Tiempo de ejecución vs tamaño del conjunto.

5.3. Ejecución de Consultas

La sintaxis del lenguaje de consultas que vamos a considerar en nuestra implementación es la definida por la siguiente gramática

$$\phi := [T] \mid [v] \mid [attr = k] \mid c \mid c\hat{} \mid \phi.\phi \mid \phi + \phi \mid \phi\&\phi \mid \phi - \phi \mid \phi\tilde{} \mid \phi? \mid \phi^* \ .$$

Esta gramática es equivalente a la gramática original, sólo que fue adaptada para utilizar los caracteres del estándar *ASCII* [33]. Para evitar confusiones, vamos a volver a definir la función de evaluación utilizando esta gramática.

Definición 5.3.1 Dada una *GDB* G, donde V es el conjunto de nodos, la semántica de nuestro lenguaje L está definida a través de la función de evaluación $\lambda : L \times 2^V \rightarrow 2^V$, en que por defecto una consulta ϕ se evalúa como $\lambda(\phi, V)$. La definición de λ es la siguiente:

1. $\lambda([T], V_0) = V_0$.
2. $\lambda([v]\ , V_0) = \{u \in V_0 |\ u = v\}$.
3. $\lambda([attr = k], V_0) = \{u \in V_0 |\ attr(v) = k\ \}$.
4. $\lambda(c, V_0) = \{u \in V |\ \exists v \in V_0$ tal que $(u, c, v) \in E\}$.
5. $\lambda(c\hat{}, V_0) = \{u \in V |\ \exists v \in V_0$ tal que $(v, c, u) \in E\}$.
6. $\lambda(\phi_1.\phi_2, V_0) = \lambda(\phi_1, \lambda(\phi_2, V_0))$.
7. $\lambda(\phi_1 + \phi_2, V_0) = \lambda(\phi_1, V_0) \cup \lambda(\phi_2, V_0)$.
8. $\lambda(\phi_1\&\phi_2, V_0) = \lambda(\phi_1, V_0) \cap \lambda(\phi_2, V_0)$.
9. $\lambda(\phi_1 - \phi_2, V_0) = \lambda(\phi_1, V_0) \setminus \lambda(\phi_2, V_0)$.
10. $\lambda(\phi\tilde{}, V_0) = V \setminus \lambda(\phi, V_0)$.
11. $\lambda(\phi?, V_0) = \lambda(\phi, V) \cap V_0$.
12. $\lambda(\phi^*, V_0) = \bigcup_{k \geq 0} \bar{V}_k$, donde $\bar{V}_0 = \Sigma_0 = V_0$, $\bar{V}_k = \lambda(\phi, \bar{V}_{k-1}) \setminus \Sigma_{k-1}$ y $\Sigma_k = \Sigma_{k-1} \cup \bar{V}_k$ con $k > 0$.

A continuación se describirá la implementación de cada una de estas consultas. Para esto, la clase *Query* es la representación interna de las consultas, y cada consulta debe implementar la función *public Nodes ejecutar(Graph G,Nodes N)*, donde la clase *Graph* representa el grafo y la clase *Nodes* representa un conjunto de nodos.

5.3.1. Consultas Atómicas

Las consultas atómicas corresponden a los casos 1,2 y 3 de la definición anterior. La clase Query, que es la representación interna de las consultas, implementa distintas funciones para discriminar el tipo de consulta que representa. Lo que hace la función *ejecutar(Graph G,Nodes N)* dependiendo del caso es:

- $[T]$: Esta consulta es constante y siempre devuelve el conjunto de nodos N.
- $[v]$: Esta consulta devuelve un conjunto de un único elemento $\{v\}$, siempre y cuando $v \in N$, de lo contrario devuelve un conjunto vacío.
- $[attr = k]$: Esta consulta extrae del grafo G los nodos que en el atributo *attr* tienen un valor igual a k, para luego intersectar este conjunto con los nodos N.

Estos casos se traduce en el siguiente código:

```
public Nodes ejecutar(Graph g,Nodes N){
    if(isTrue())
        return N;
    else if(isSingleton()){
        if(N.contains(node))
            return new Nodes(node);
        else
            return new Nodes();
    } else if(isCondition())
        return N.interseccion(g.getNodosWhere(label, val));
}
```

La implementación del método *getNodosWhere* fue explicado en la sección anterior, mientras que la implementación del método *interseccion* será explicado más adelante.

5.3.2. Navegando el Grafo

Las consultas que nos permiten recorrer los arcos del grafo, y así formar caminos, son los casos 4,5 y 6 de la definición de λ. Lo que implementa la función *ejecutar(Graph G,Nodes N)* en estos caso es:

- c : Esta consulta entrega todos los nodos que tengan algún arco de salida, con etiqueta c, que llegue a algún nodo en el conjunto N.
- c^- : Esta consulta entrega todos los nodos que tengan algún arco de entrada, con etiqueta c, que salga desde algún nodo en el conjunto N.
- $\phi_1.\phi_2$: Esta consulta entrega todos los nodos que, siguiendo primero la consulta ϕ_1 y luego la consulta ϕ_2, alcanzan algún nodo del conjunto N.

Código

```
public Nodes ejecutar(Graph g,Nodes N){
    if(isLabel())
        return g.getTailsWithHeadsIn(label,N);
    else if(isConverse()){
        return g.getHeadsWithTailsIn(label,N);
    } else if(isConcat())
        return izq.ejecutar(g,der.ejecutar(g,N));
}
```

Los métodos *getTailsWithHeadsIn* y *getHeadsWithTailsIn* fueron explicados en la sección anterior. El proceder de la concatenación se puede resumir en:

1. Ejecuta la consulta ϕ_2, que corresponde al subárbol de la derecha, obteniendo el conjunto de nodos N_2 desde los cuales es posible alcanzar algún nodo del conjunto N siguiendo ϕ_2.

2. Ejecuta la consulta ϕ_1, que corresponde al subárbol de la izquierda, obteniendo el conjunto de nodos N_1 desde los cuales es posible alcanzar algún nodo del conjunto N_2 siguiendo ϕ_1.

3. Retorna el conjunto N_1, que coincide con los nodos que alcanzan al conjunto N, siguiendo primero la consulta ϕ_1 y luego la consulta ϕ_2.

5.3.3. Combinaciones Booleanas

Las combinaciones booleanas de consultas modelan todas las operaciones básicas de conjuntos. Lo que hace la función *ejecutar(Graph G,Nodes N)* en este tipo de consultas es:

- $\phi_1 + \phi_2$: Esta consulta entrega la *unión* de los resultados de las consultas ϕ_1 y ϕ_2.
- $\phi_1 \wedge \phi_2$: Esta consulta entrega la *intersección* de los resultados de las consultas ϕ_1 y ϕ_2.
- $\phi_1 - \phi_2$: Esta consulta entrega la *diferencia* de los resultados de las consultas ϕ_1 y ϕ_2.
- $\neg\phi$: Esta consulta entrega el *complemento* de los resultados de las consultas ϕ_1 y ϕ_2.
- $\phi?$: La operación *nesting* no es una operación booleana, pero su implementación es muy similar. Esta consulta entrega todos los nodos del conjunto N que satisfacen ϕ.

Código

```
public Nodes ejecutar(Graph g,Nodes N){
    if(isOr()){
        return izq.ejecutar(g, N).union(der.ejecutar(g, N));
    }else if(isAnd()){
        return izq.ejecutar(g, N).interseccion(der.ejecutar(g, N));
    }else if(isMinus()){
        return izq.ejecutar(g, N).diferencia(der.ejecutar(g, N));
    }else if(isNot()){
        return izq.ejecutar(g, N).complemento(g);
    }else if(isNesting()){
        return izq.ejecutar(g).interseccion(N);
}
```

5.3.4. Computando la Recursión

Tal como se explicó en el Capítulo 4, para computar $\lambda(\phi^*, V_0)$ es necesario definir de forma inductiva los conjuntos:

- $\bar{V_0} = \Sigma_0 = V_0$.
- $\bar{V_k} = \lambda(\phi, \bar{V}_{k-1}) \setminus \Sigma_{k-1}$.
- $\Sigma_k = \Sigma_{k-1} \cup \bar{V_k}$.

Ahora bien, como el grafo es finito, estas operaciones se realizan a lo más $|V|$ veces, deteniendo cuando k es la primera vez que $\bar{V_k} = \{\}$ y retornando Σ_k. La definición inductiva saca provecho del hecho que "si alcanzo un nodo que ya había alcanzado anteriormente, no se obtendrán resultados nuevos", por lo que en cada paso sólo prueba con los nodos nuevos. Con esto logramos recorrer el grafo sólo 1 vez, obteniendo el resultado óptimo. El algoritmo debe realizar en cada paso:

1. Computar el conjunto $\lambda(\phi, \bar{V}_{k-1})$.
2. Restar los conjuntos $\lambda(\phi, \bar{V}_{k-1})$ y Σ_{k-1}.
3. Unir los conjuntos Σ_{k-1} y $\bar{V_k}$.

Para optimizar el algoritmo, se extendió la clase *Nodes* con el método *diferenciaUnion(Nodes N)*, el cual realiza las operaciones de resta y unión de los conjuntos de forma simultánea, iterando en cada paso sólo los conjuntos $\bar{V_k}$, asegurando recorrer a lo más $|V|$ nodos.

Código

```
public void diferenciaUnion(Nodes X){
    for(this.init();this.notNull();this.next()){
        if(X.contains(this.getNodo()))
            this.remove(this.getNodo());
        else
            X.add(this.getNodo());
    }
}

public Nodes ejecutar(Graph g,Nodes N){
    if(isKleene()){
        Nodes out = N;
        Nodes new_out = izq.ejecutar(g,N);
        new_out.diferenciaUnion(out);
        while(new_out.count()>0){
            new_out=izq.ejecutar(g,new_out);
            new_out.diferenciaUnion(out);
        }
    }
}
```

Capítulo 6

Descripción de la Implementación

En este capítulo vamos a describir brevemente las librerías usadas y los archivos programados, con el fin de dejar el proyecto lo más claro posible. Luego revisaremos la *CLI* (del inglés *Command Line Interface*), explicando la sintaxis válida a base de ejemplos de uso.

6.1. Diagrama de Clases en Java

La implementación está desarrollada en el lenguaje *Java,* y está separada en distintos paquetes lógicos (*packages* en inglés), los cuales contienen clases afines. La Figura 6.1 nos muestra el diagrama de clases, distribuidas en los distintos *packages* y las dependencias entre sí. A continuación vamos a explicar cada uno de los *packages* y las clases que la componen, partiendo por el nivel más bajo de abstracción, llegando hasta el nivel más alto.

6.1.1. libraries

Este *package* contiene todas las librerías externas utilizadas en la implementación, principalmente en la representación interna de las distintas estructuras de datos de memoria interna.

fastutil

Esta librería es usada principalmente en la representación interna de los conjuntos de nodos, representados como *longs*, ya que la clase *LongOpenHashSet* implementa las operaciones básicas de insercción, eliminación y pertenencia de elementos en el conjunto. El otro uso es en la representación interna del mapeo *nodo-puntero*, ambos representados como *longs*, que se realiza gracias a la clase *Long2LongOpenHashMap.*

guava

Esta librería, desarrollada por Google, es utilizada como complemento a la librería anterior, ya que implementa estructuras que son necesarias en nuestra implementación y no están disponibles en *fastutil*. Una de las clases usadas es *HashBiMap*, que nos permite crear un mapeo *String-long,* utilizado como representación interna del diccionario para los atributos

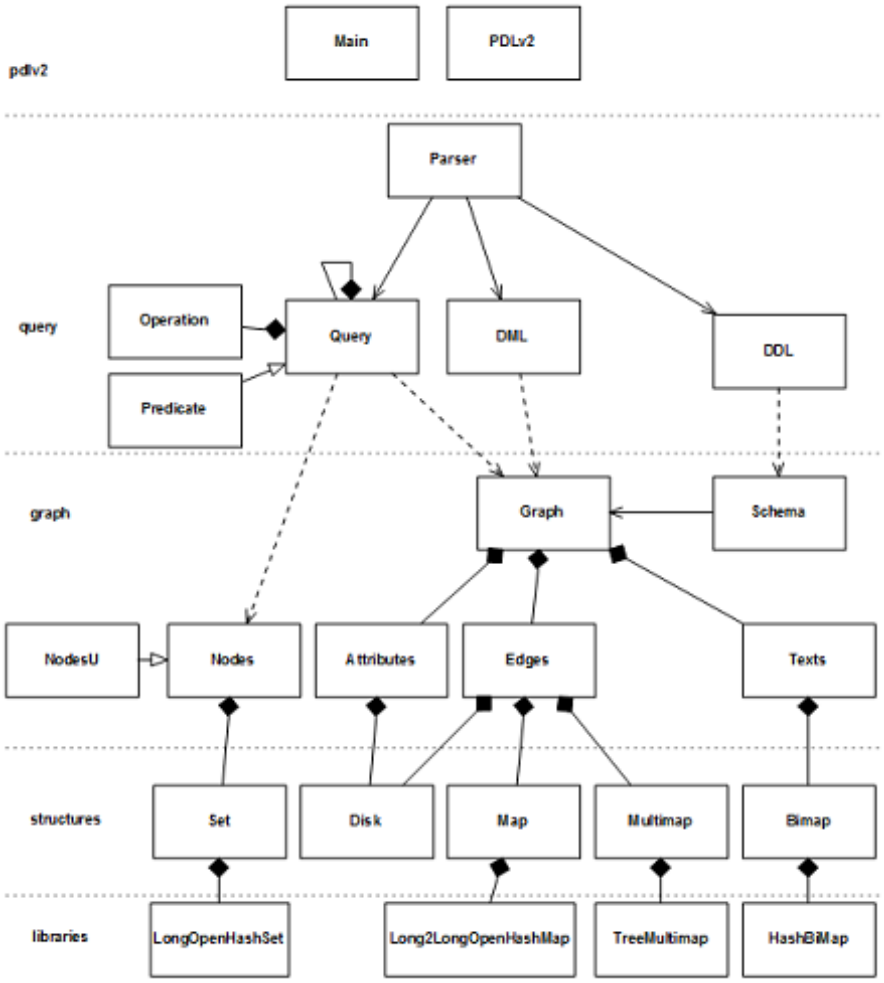

Figura 6.1: Diagrama de Clases en Java

del tipo *String*. La otra clase es *TreeMultimap*, que nos permite crear un multi-mapeo de *longs*, es decir almacenar conjuntos de *longs* con llave un *long*.

6.1.2. structures

Este *package* agrupa todas las clases que implementan las diferentes estructuras de datos, tanto de memoria externa como de memoria interna, que son utilizadas por las clases de más alto nivel.

Disk

La clase *Disk* implementa el acceso a disco de todas las estructuras de datos en memoria externa, que almacenan *longs*. Cada instancia hace referencia a un único *file*, con un número predeterminado de *longs* por cada fila, los cuales puede leer y escribir, utilizando *buffers* con el fin de optimizar el rendimiento.

Texts

La clase *Texts* es una modificación de la clase *Disk*, con la diferencia que por cada fila almacena un *long* y un *String* de 124 caracteres. Una optimización que tiene esta clasa es que mantiene una copia de la estructura en memoria interna, con el fin de hacer más eficiente la búsqueda que realizan los atributos del tipo *String*.

Set

La clase *Set* es la representación intermedia de un conjunto nodos, ya que utiliza la clase *LongOpenHashSet* de la librería *fastutil* como representación interna. Esta clase hereda los métodos de insertar, eliminar y consultar *longs* de la clase *LongOpenHashSet*, pero implementa el iterador del conjunto.

Map

La clase *Map* es muy similar a la clase *Set*, con la diferencia que es la representación intermedia del mapeo *key-value*, utilizando la clase *Long2LongOpenHashMap* de la librería *fastutil* como representación interna. Esta clase hereda de *Long2LongOpenHashMap* la inserción de *key-value*, pero implementa un iterador especial, ya que es creciente en la variable *key*, y a su vez elimina los registros a medida que los recorre.

Bimap

La clase *Bimap* es una estructura auxiliar, utilizada por la clase *Texts*, donde se funcionalidad es mantener en memoria principal una copia de la información almacenada en la memoria externa. Como representación interna utiliza la clase *HashBiMap* de la librería *guava.*

Multimap

La clase *Multimap* es una estructura auxiliar, donde su representación interna está implementada con la clase *TreeMultimap* de la librería *guava*. Esta clase es usada por la clase *Edges*, que la utiliza a modo de *buffer* para la inserción de nuevos arcos, con el fin de minimizar la cantidad de accesos al disco.

6.1.3. graph

Este *package* agrupa todas las clases que representan los componentes de una base de datos de grafos, tales como los nodos, arcos, atributos y el grafo en sí. La gran mayoría de las clases utilizan las clases de *structures* como representaciones internas.

Nodes

La clase *Nodes* es la representación de conjuntos de nodos, los cuales son usados principalmente para representar las respuestas a las consultas ejecutadas. Utilizan la clase *Set*, del *package structures,* como representación interna, de los cuales hereda los métodos de insertar,

eliminar, consultar e iterar los *longs*, pero a su vez implementa los métodos de unión, intersección, complemento, diferencia y diferenciaUnion, explicados anteriormente en el Capítulo 5.

NodesU

La clase *NodesU* es una subclase de *Nodes*, con la diferencia que representa el conjunto de todos los nodos del grafo. Por esta razón se reimplementan todos los métodos, ya que todas las operaciones tienen un resultado constante, y de esta forma optimizamos la implementación.

Attributes

La clase *Attributes* es la representación de una función de asignación, la cuál asigna a cada nodo un valor del tipo *long*. Esta clase se compone de 3 instancias diferentes de *Disk*, donde la primera corresponde a los valores de cada nodo, mientras que las otras dos se utilizan para almacenar la lista de nodos que tienen un cierto valor. Los métodos principales son los de setear el valor de un nodo, obtener el valor del nodo, e iterar los nodos que tienen un cierto valor.

Edges

La clase *Edges* es la representación del conjunto de arcos del grafo, dada una etiqueta *label* fija. Este objeto es uno de los más importantes en la implementación, ya que este objeto tiene la mayor cantidad de accesos al disco. Los métodos principales de esta clase es la insercción de arcos etiquetados, y la iteración de los arcos de forma eficiente. Su representación interna está compuesta por 2 instancias diferentes de *Disk*, una instancia de *Multimap* y una instancia de *Map*.

La primera instancia de *Disk* hace refencia al archivo que almacena, para cada nodo del grafo, la lista de los nodos alcanzados por un arco etiquetado por *label*. La segunda instancia de *Disk* almacena el desbordamiento de la clase anterior, emulando así un Multi Hash. Para más detalles sobre esta estructura de los archivos, revisar la sección 2 del Caítulo 4.

La clase *Multimap* es utilizada como un *buffer* que almacena los arcos pendientes por insertar, con el fin de optimizar el tiempo de creación de la base. La clase *Map* es utilizada como una estructura auxiliar, la cuál almacena todos los accesos pendientes al disco, con el fin de lograr una lectura secuencial de ambos archivos en la iteración. Para más detalles del algoritmo de iteración, revisar el Capítulo 5.

Schema

La clase *Schema* se utiliza como el constructor de la clase *Graph*, ya que esta clase es quien abre los archivos que corresponden, inicializa las clases *Edges*, *Attributes* y *Texts,* para así crear una instancia de la clase *Graph.* Además, esta clase se preocupa de la creación de nuevos tipos de arcos y de atributos.

Graph

La clase *Graph* es la representación en sí del grafo, por lo que tiene acceso a todas las clases *Edges*, *Attributes* y *Texts*, además de almacenar la cantidad de nodos en el grafo. Esta clase implementa los métodos de inserción de nodos, inserción de arcos, asignación de valores a los atributos y consultar el valor de un atributo, además de todos los métodos necesarios para computar las consultas atómicas, tales como *getNodosWhere*, *getTailsWithHeadsIn* y *getHeadsWithTailsIn*, explicados con mayor detalle en el Capítulo 5.

6.1.4. query

Este *package* contiene todas las clases necesarios para la representación y ejecución de las consultas del lenguaje *Converse-PDL*, además de clases que implementan los lenguajes de definición y de manipulación de datos. Adicionalmente, se incluye una clase que parsea un *string*, de una sintaxis clara, y construye la clase correspondiente.

Operation

La clase *Operation* representa una operación entre consultas, que pueden ser binarias o unarias, con métodos auxiliares que discriminan el tipo de operación que representa el objeto.

Predicate

La clase *Predicate* representa las consultas atómicas, las cuales no pueden descomponerse en otras más simples. Tiene variables internan que almacenan los parámetros, según el tipo de consulta que corresponde. Esta clase implementa el método *ejecutar*, que computa el resultado de la consulta que representa. Esta clase es la que accede a todos los métodos de consulta de datos de la clase *Graph*. Más detalles de la implementación se encuentran en la sección 2 del Capítulo 5.

Query

La clase *Query* es la representación interna de todas las consultas de nuestro lenguaje, la cual puede estar compuesta por un *Predicate*, una *Operation* y una *Query* (caso unario), o por una *Operation* y dos *Query* (caso binario). La implementación del método *ejecutar* se basa en los métodos de la clase *Predicate,* y en los métodos de la clase *Nodes*. Más detalles de la implementación se encuentran en la sección 3 del Capítulo 5.

DDL

La clase *DDL* representa el lenguaje de definición de datos (*Data Definition Language* en inglés), el cual permite crear nuevos tipos de arcos etiquetados, o nuevos tipos de atributos. Esta clase sirve de interfaz para ejecutar los métodos implementados en la clase *Schema*.

DML

La clase *DML* representa el lenguaje de manipulación de datos (*Data Manipulation Language* en inglés), el cual permite insertar nodos, insertar arcos y asignar valores en los atributos. Esta clase sirve de interfaz para ejecutar los métodos implementados en la clase *Graph*.

Debug

La clase *Debug* cumple el objetivo de hacer *debugging* del sistema, permitiendo imprimir los archivos según se indique. La funcionalidad de esta clase es netamente auxiliar, y tiene un uso sólo para los desarrolladores.

Parser

La clase *Parser* nos permite transformar un *string* en un objeto *Query, DDL, DML* o *Debug*, para luego ejecutarlo. Esta clase es la interfaz más útil, ya que nos permite controlar la base de datos por línea de comandos, a través del lenguaje de bases de datos. El detalle de la sintaxis reconocida por el parser se va a revisar en la próxima sección.

6.1.5. pdlv2

El *package pdlv2* contiene dos interfaces para interactuar con la base de datos, una a través de línea de comando, y la otra a través de objetos.

PDLv2

La clase *PDLv2* es la librería que engloba todos los métodos públicos de la base de datos, tales como abrir y cerrar la base de datos, ejecutar consultas, definir tipos de arcos y atributos, insertar nodos, insertar arcos, asignar valores en los atributos, consultar los atributos de los nodos e imprimir los resultados de una consulta. Esta clase hace referencia a las clases *Parser*, *Schema* y *Graph*.

Main

La clase *Main* implementa una interfaz a la base de datos a través de línea de comando, haciendo referencia a una instancia de *PDLv2*. Esta clase lee la entrada estándar, ejecuta el *Parser* para obtener la consulta, para luego ejecutarla en *Graph* e imprimir sus resultados por la salida estándar. En la próxima sección se detallará la sintaxis reconocida por el parser, mostrando ejemplos de uso.

6.2. Interfaz de Línea de Comandos

En esta sección vamos a revisar la interfaz de usuario implementada, listando todas las instrucciones que reconoce el parser, mostrando ejemplos en cada caso. Esta interfaz corresponde a una interfaz de línea de comandos, que se activa al ejecutar la clase *Main*, mostrando la siguiente interfaz:

```
PDL>
```

6.2.1. Lenguaje de Definición

El lenguaje de definición de datos implementado es bastante simple, ya que las únicas funciones disponibles son la definición de tipos de arcos y de tipos de atributos. La gramática se define como:

def [edge|long|string] label

donde *label* es la etiqueta del tipo de dato definido. Por ejemplo, si quisieramos construir una base de datos de personas, con atributos *edad* y *nombre*, del tipo *long* y *string* respectivamente, basta con escribir:

```
PDL> def long edad
PDL> def string nombre
```

Si a nuestra base de datos quisieramos agregar arcos con la etiqueta *amigo*, entonces debemos definirlos con la siguiente instrucción:

```
PDL> def edge amigo
```

6.2.2. Lenguaje de Manipulación

Una vez definidos los tipos de arcos y los tipos de atributos, debemos insertar datos en la base. Para insertar nodos basta con escribir:

```
PDL> new node
node(1)[ ]
PDL> new node
node(2)[ ]
PDL> new node
node(3)[ ]
```

Podemos observar que esta instrucción imprime el nodo insertado. Entre los corchetes de cada nodo se imprimen los valores de sus atributos, que hasta el momento no han sido asignados. Para asignar valores a los atributos de los nodos, la sintaxis es de la forma:

set attr val node

donde *attr* es la etiqueta del atributo, *val* es el valor a asignar y *node* es el id del nodo. Veamos un ejemplo de asignación de valores a los atributos:

```
PDL> set edad 18 1
node(1)[ edad=18 ]
```

```
PDL> set nombre Pedro 2
node(2)[ nombre=Pedro ]
PDL> set edad 24 3
node(3)[ edad=24 ]
PDL> set nombre Juan 3
node(3)[ nombre=Juan edad=24 ]
```

Dado que ahora disponemos de datos con atributos, a continuación vamos a relacionar datos entre sí a través de arcos. La sintaxis de esta instrucción es:

new label node1 node2

donde *label* es la etiqueta del arco, *node1* es el id del nodo de salida y *node2* es el id del nodo de entrada. Siguiendo con nuestro ejemplo, para relacionar que el nodo *1* es amigo del nodo *3*, y que el nodo *3* es amigo del nodo *2,* podemos escribir los siguientes comandos:

```
PDL> new amigo 1 3
node(1)[ edad=18 ]
node(3)[ nombre=Juan edad=24 ]
PDL> new amigo 3 2
node(2)[ nombre=Pedro ]
node(3)[ nombre=Juan edad=24 ]
```

Ahora que ya vimos como insertar datos en la base de datos, a continuación se va a revisar la sintaxis del lenguaje de consultas, que es el aspecto más interesante de esta implementación.

6.2.3. Lenguaje de Consulta

En esta sección revisaremos la sintaxis del lenguaje de consultas, principalmente a través de ejemplos. Para que las consultas sean más interesantes, vamos a suponer que tenemos definida la etiqueta *padre*, además de las ya definidas anteriormente.

La consulta más simple es aquella que devuelve todos los nodos del grafo:

```
PDL> [T]
node(1)[ edad=18 ]
node(2)[ nombre=Pedro ]
node(3)[ nombre=Juan edad=24 ]
node(4)[ nombre=Diego ]
node(5)[ edad=24 ]
```

Otra consulta importante saber si un nodo pertenece al grafo:

```
PDL> [3]
node(3)[ nombre=Juan edad=24 ]
PDL> [6]
```

Dado que los nodos almacenan valores en sus atributos, podemos consultar por el conjunto de nodos que tengan un cierto valor en uno de sus atributos:

```
PDL> [edad=24]
node(3)[ nombre=Juan edad=24 ]
node(5)[ edad=24 ]
```

Otro aspecto importante en nuestro lenguaje de consultas es la posibilidad de navegar el grafo recorriendo sus arcos. La consulta más simple de este tipo es saber cuales nodos tienen algún arco etiquetado de salida:

```
PDL> amigo
node(1)[ edad=18 ]
```

Análogamente, podemos consultar cuales nodos tienen algún arco etiquetada de entrada:

```
PDL> amigo^
node(2)[ nombre=Pedro ]
node(3)[ nombre=Juan edad=24 ]
```

Con estas consultas como base, podemos formular consultas más complejas, tales como formar caminos con los arcos, a través de la concatenación. Podemos consultar por los nodos que sean *amigos* de una persona que tenga un *padre*:

```
PDL> amigo.padre^
node(3)[ nombre=Juan edad=24 ]
```

Las consultas pueden tener largo arbitrario, formulando así consultas cada vez más complejas. El lenguaje permite expresar condiciones intermedias en el camino, logrando de esta forma construir árboles en vez de sólo caminos. Por ejemplo, podemos consultar por los nodos que sean *amigos* de una persona, que tenga un *padre* y que sea *amigo* de alguien:

```
PDL> amigo.(padre^)?.amigo^
node(3)[ nombre=Juan edad=24 ]
```

Otra de las características de este lenguaje es que implementa todas las operaciones básicas de conjuntos, permitiendo expresar la unión, intersección, diferencia y complemento:

```
PDL> padre+amigo
node(1)[ edad=18 ]
node(5)[ edad=24 ]
PDL> (padre+amigo)&[edad=18]
node(1)[ edad=18 ]
PDL> (padre+amigo)-((padre+amigo)&[edad=18])
```

```
node(5)[ edad=24 ]
PDL> (padre+amigo)~
node(2)[ nombre=Pedro ]
node(3)[ nombre=Juan edad=24 ]
node(4)[ nombre=Diego ]
```

Como podemos ver, el lenguaje es lo suficientemente expresivo para combinar todas las operaciones entre sí. Por última, una gran diferencia que tiene este lenguaje frente a otros, es que permite expresar recursión, lo que nos da un enorme poder expresivo. Por ejemplo, podemos expresar la consulta que representa el conjunto de los nodos que están conectados, por 0 o más arcos del tipo *padre* o *amigo*, con personas de *edad=24:*

```
PDL> (padre+amigo)*.[edad=24]
node(1)[ edad=18 ]
node(3)[ nombre=Juan edad=24 ]
node(5)[ edad=24 ]
```

Como podemos ver, la interfaz de línea de comandos es bastante simple de entender, y considera todos los aspectos teóricos estudiados en los capítulos anteriores. En el próximo capítulo vamos a realizar una comparación de nuestra implementación con otras bases de datos: una base de datos relacional de código libre, una base de datos de grafos de código libre con un lenguaje de consultas ambiguo y con una base de datos de grafos cerrada sin lenguaje de consultas.

Capítulo 7

Evaluación de la Implementación

En el presente capítulo vamos a realizar una evaluación de nuestra implementación, computando distintas consultas y midiendo el tiempo de ejecución. Finalmente, vamos a hacer una comparación con los resultados obtenidos de tres implementaciones distintas: DEX, Neo4J y MySQL. Cabe mencionar que todos los experimentos fueron ejecutados en un computador con un sistema operativo Windows 7 Profesional de 32-bits, con 4 GB de memoria RAM (3.45 GB utilizable) y un procesador Intel Core 2 Duo P9400 de 2.40 GHz.

El fin de esta evaluación es verificar que los algoritmos y las estructuras de datos funcionen, y que tengan una utilidad práctica. Estamos conscientes que los experimentos diseñados son simples y que no son concluyentes como un análisis empírico, pero sirven como una prueba de concepto, y los resultados obtenidos no dejan de ser interesantes. El código fuente de nuestra implementación quedará publicado en la web, con el fin que en el futuro otros investigadores puedan realizar experimentos de evaluación más avanzados.

7.1. Creación de la Base de Datos

Para nuestros experimentos, vamos a crear una base de datos lo suficientemente grande como para que nuestras hipótesis sean correctas. Para esto, vamos a generar una base con datos sintéticos, del orden de 100.000 nodos y 15.000.000 de arcos. El procedimiento seguido en las cuatro implementaciones fue el siguiente:

1. Definir el atributo *id*, de tipo *long*, donde será utilizado como identificador único del nodo.
2. Definir los atributos *edad* y *nombre*, de tipo *long* y *String* respectivamente.
3. Definir las relaciones *padre* y *conocido.*
4. Insertar 100.000 nodos, asignando valores a todos los atributos:
   a) *id*: números consecutivos, partiendo del 1 y terminando con el 100.000,
   b) *edad*: un número entre 18 y 88, distribuidos de forma uniforme.
   c) *nombre*: la palabra *persona* concatenada con un número entre 0 y 96, distribuidos de forma uniforme.

5. Por cada nodo, seleccionar de forma aleatoria 2 nodos, para luego insertar 1 arco con etiqueta *padre* desde cada uno de los nodos seleccionados hasta el nodo actual.

6. Por cada nodo, seleccionar de forma uniforme un número entre 0 y 300, para luego seleecionar ese número de nodos de forma aleatoria, para luego insertar 1 arco con etiqueta *conocido* desde el nodo actual a cada uno de los nodos seleccionados.

Para llevar a cabo esta tarea, se procedió a descargar las implementaciones de DEX, Neo4J y MySQL, leer y estudiar su documentación, para luego instalarlas y hacerlas funcionar. Una vez hechas las pruebas iniciales de uso, se realizó la definición de la base de datos recién descrita. Nuestra experiencia para cada una de las implementaciones es la descrita a continuación.

PDL

Nuestra implemetación es bastante simple a nivel de su instalación y uso, permitiendo el uso de su *CLI* o de su *API* para la definición de la base de datos. La insercción de datos fue lenta, demorándose varias horas en completar la tarea. La base de datos alcanzó un tamaño de 0.39 GBytes, distribuidos en 20 archivos.

MySQL

MySQL es una de las base de datos relacionales más usadas en el mundo, que implementa el lenguaje de consulta *SQL*, y es activamente desarrollada a lo largo del mundo. Por esta misma razón, esta implementación tiene una documentación muy completa, por lo que realizar una instalación es muy simple. Definir la base de datos fue muy fácil y requirió muy poca programación, mostrándose como una implementación robusta y sin errores. La insercción de datos fue lenta, demorándose un par de horas en completar la tarea. La base de datos alcanzó un tamaño de 0.57 GBytes, distribuidos en 10 archivos.

DEX

Para llevar a cabo la definición de la base de datos utilizando DEX, requirió un gran estudio de la documentación disponible, además de realizar muchas pruebas de uso. DEX provee de una *API* bastante completa, pero de gran complejidad, por lo que requiere de harta programación para completar una tarea. Además presenta muchos errores en el manejo de transacciones y del uso de memoria RAM, por lo que podemos considerar que la implementación es poco robusta. La insercción de datos fue muy rápida, demorándose menos de una hora en completar la tarea. La base de datos alcanzó un tamaño de 1.05 GBytes, distribuidos en sólo 2 archivos.

Neo4J

Neo4J es un proyecto de código abierto, actualmente en desarrollo, que provee de una *API* bastante completa, pero de una documentación deficiente. El problema principal de esta implementación es que para definir una base de datos, es necesario programar unas clases especiales que son bastante complejas y de difícil manejo. La insercción de datos fue

lenta, demorándose un par de horas para completar la tarea. La base de datos alcanzó un tamaño de 3.12 GBytes, distribuidos en cerca de 300 archivos.

Una vez definidas las bases de datos, procedemos a definir cada una de las consultas que se usarán para evaluar la eficiencia de las implementaciones.

7.2. Consultas de Pruebas

Para que nuestra evaluación sea lo más completa posible, es necesario definir un conjunto de consultas a ejecutar, lo más completo posible, que considere todos los tipos de consultas posible. Para esto se programaron, en cada una de las implementaciones, 15 tipos de consultas diferentes, las cuales usan distintos elementos del lenguaje.

Cabe destacar que ciertas consultas no son soportadas en MySQL y en Neo4J (las que incluyen recursión), por lo que se procedió a programar una variación de la consulta, aproximando de mejor manera la consulta objetivo. Otro punto a destacar es que DEX, al no contar con un lenguaje de consulta, se procedió a implementar nuestro lenguaje sobre la *API* que provee DEX, con el fin de evaluar el lenguaje en sí y de las respectivas implementaciones. A continuación vamos a explicar cada unas de las consultas implementadas.

1. *selectId(long id):* Implementa la consulta *[id]*, interpretada como la persona identificada como *id*.

2. *selectNodesWithEdad(long e)*: Implementa la consulta *[edad=e]*, interpretada como las personas de una *edad* igual a *e*.

3. *selectPadresOf(long id)*: Implementa la consulta *padre.[id]*, interpretada como los *padres* de *id*.

4. *selectHijosOf(long id)*: Implementa la consulta *padre^.[id]*, interpretada como los *hijos* de *id*.

5. *selectConocidoFrom(long id)*: Implementa la consulta *conocido.[id]*, interpretada como los *conocidos* de *id*.

6. *selectConocidosOf(long id)*: Implementa la consulta *conocido^.[id]*, interpretada como los que *conocen* a *id*.

7. *selectNotPadres()*: Implementa la consulta *padre~*, interpretada como las personas que no son *padres* de alguien.

8. *selectAbuelosOf(long id)*: Implementa la consulta *padre.padre.[id]*, interpretada como los *abuelos* de *id*.

9. *selectAbuelosOfEdad(long e)*: Implementa la consulta *padre.padre.[edad=e]*, interpretada como los *abuelos* de las personas de una *edad* igual a *e*.

10. *selectAbuelosOfEdad_ConocidoOfEdad(long e1,long e2)*: Implementa la consulta *(padre.padre.[edad=e1])&(conocido.[edad=e2])*, interpretada como los *abuelos* de las personas de una *edad* igual a *e1*, que además son *conocidos* por una persona de *edad* igual a *e2*.

11. *selectHermanoOfEdad(edad e)*: Implementa la consulta *padre^.padre.[edad=e]*, interpretada como los *hermanos* de las personas de una *edad* igual a *e*.

12. *selectConocidoConocido(long id)*: Implementa la consulta *conocido.conocido.[id]*, interpretada como los *conocidos* de los *conocidos* de *id*.

13. *selectConocidoRecursivo(long id)*: Implementa la consulta *conocido*.[id]*, interpretada como las personas que están conectadas por un número arbitrario de *conocidos* de *id*.

14. *selectPadreRecursivo(long id)*: Implementa la consulta *padre*.[id]*, interpretada como los *ancestros o si mismo* de *id*.

15. *selectPadresEdadConocidoEdadPadre(long e1,long e2)*: Implementa la consulta *padre.[edad=e1].(conocido*).[edad=e2].padre*, interpretada como las personas que son *padres* de alguien con *edad* igual a *e1*, y que son *conocidos** de alguien con *edad* igual a *e2* que es *padre*.

Como podemos observar, el conjunto de consultas diseñado es lo suficientemente representativo de nuestro lenguaje, ya que abarcan una gran cantidad de casos interesantes que sirven para nuestro análisis. En la próxima sección se mostrarán los resultados obtenidos, así como un análisis de estos mismos.

7.3. Resultados Obtenidos

Una vez definida la base de datos, y el conjunto de consultas a evaluar, el siguiente paso es la ejecución de dicho conjunto de pruebas. Con el fin de obtener resultados confiables, cada consulta se evaluó una cantidad de veces acorde a su complejidad, ejecutando miles de veces para las consultas más simples, cientos de veces las consultas de complejidad mediana, y sólo una decena de veces las consultas de mayor complejidad. Este experimento se ejecutó en 3 momentos diferentes, con el fin de minimizar los elementos externos que pudieran afectar los resultados. Los resultados de las pruebas pueden verse en la Figura 7.1.

Los resultados obtenidos son bastante satifactorios en primera instancia, ya que todas las consultas de menor complejidad se demoran del orden de 10 milisegundos por consulta, y las de complejidad mediana no sobrepasan los 100 milisegundos por consulta, y computar consultas con recursión toma sólo 2 segundos. Para poder evaluar de mejor manera estos resultados, se procedió a implementar el mismo experimento, pero ahora usando las implementaciones de *MySQL*, *Neo4J* y *DEX*, para luego comparar los tiempos de ejecución de cada consulta. Los resultados de la comparación entre las 4 implementaciones pueden verse en la Figura 7.2.

En la figura, la columna *PDL #ms/Q* muestra el tiempo promedio de ejecución para cada consulta. Las columnas restantes muestran el porcentaje del tiempo de PDL que utiliza

CONSULTA	#Q	Test1: ms	Test2: ms	Test3: ms	#ms/Q
selectId	5000	5194	7020	7536	1,32
selectNodesWithEdad	2000	9532	11279	8862	4,95
selectPadresOf	2000	12012	6833	11201	5,01
selectHijosOf	2000	10889	9516	9562	4,99
selectConocidosOf	500	6583	6364	4899	11,90
selectConocidoFrom	500	6053	8768	5959	13,85
selectAbuelosOf	1000	11825	9516	6895	9,41
selectConocidoConocido	100	8736	8372	11124	94,11
selectAbuelosOfEdad	200	6022	3932	5944	26,50
selectHermanoOfEdad	200	7769	4227	8518	34,19
selectNotPadres	200	8205	6708	5304	33,70
selectAbuelosOfEdad_ConocidoOfEdad	10	1685	2762	2184	221,03
selectPadreRecursivo	10	5195	2870	5117	439,40
selectConocidoRecursivo	10	20404	19142	23027	2085,77
selectPadresEdadConocidoEdadPadre	10	21076	22214	23089	2212,63

Figura 7.1: Resultados de las Pruebas de PDL

CONSULTA	PDL #ms/Q	vs DexPDL	vs MySQL	vs Neo4J
selectId	1,3	10%	26%	963%
selectNodesWithEdad	4,9	1%	173%	1320%
selectPadresOf	5,0	7%	43%	2009%
selectHijosOf	5,0	6%	69%	784%
selectConocidosOf	11,9	11%	35%	273%
selectConocidoFrom	13,9	8%	7%	232%
selectAbuelosOf	9,4	3%	3%	928%
selectConocidoConocido	94,1	81%	262%	66%
selectAbuelosOfEdad	26,5	205%	241%	537%
selectHermanoOfEdad	34,2	183%	170%	268%
selectNotPadres	33,7	3318%	977%	4463%
selectAbuelosOfEdad_ConocidoOfEdad	221,0	270%	46371%	3572%
selectPadreRecursivo	439,4	228%	5565%	50076%
selectConocidoRecursivo	2085,8	725%	2690%	23033%
selectPadresEdadConocidoEdadPadre	2212,6	588%	90390%	26494%

Figura 7.2: Comparación de los Resultados

dicha implementación. Por ejemplo, en la primera fila correspondiente a la consulta *selectId,* podemos ver que *DEX* ocupa sólo un 10 % del tiempo que *PDL*, *MySQL* ocupa un 26 % del tiempo de *PDL,* mientras que *Neo4J* ocupa un 963 % del tiempo de *PDL*. En otras palabras, los valores menores al 100 % indican que dicha implementación es más eficiente que *PDL*, mientras que los valores mayores al 100 % indican que dicha implementación es más ineficiente que *PDL*.

Con esto en mente, es fácil concluir que *Neo4J* obtuvo el peor desempeño en casi todas las consultas (salvo *selectConocidoConocido*), y que *PDL* es más eficiente. Podemos ver que *MySQL* se comporta bastante bien para casi todas las consultas de menor complejidad (excepto *selectNodesWithEdad*), pero su desempeño cae bruscamente al aumentar la complejidad de las consultas, hasta llegar a un desempeño similar a *Neo4J* en las consultas de mayor complejidad. Finalmente, se puede observar que *DEX* obtuvo el mejor desempeño

para las consultas de menor complejidad, llegando a tiempos de ejecución bastante bajos. El problema es que, similarmente a *MySQL*, su desempeño cae al aumentar la complejidad de las consultas, claro que en menor medida. Esto se debe ya que la implementación de *DEX* optimiza de muy buena forma todos los accesos a disco, pero la implementación de los objetos en memoria principal no es tan buena como la de *PDL*.

Capítulo 8

Conclusiones y Trabajo Futuro

Tal como hemos visto durante este trabajo, existe una creciente demanda en el uso de bases de datos de grafos, por lo que es necesario contar con mejores herramientas para extender su uso. De nuestro análisis de las implementaciones existentes, podemos concluir que aún hay espacio para el desarrollo de nuevas soluciones, principalmente en la definición de un buen lenguaje de consultas.

Demostramos que el lenguaje propuesto, *Converse-PDL*, cumple con el requisito de ser eficiente computacionalmente, además de ser lo suficientemente expresivo como para ser utilizado en diversas aplicaciones. Realizamos un estudio de expresividad, comparando nuestro lenguaje con dos lenguajes de consultas importantes en el área de bases de datos de grafos: *nested regular expression* y *conjuntive two-way regular path queries*, obteniendo unos resultados interesantes de expresividad.

Luego, mostramos que el lenguaje propuesto es implementable eficientemente, respetando todas las restricciones que existen en nuestro modelo computacional, principalmente el uso de la memoria principal y la lectura secuencial en la memoria secundaria, diseñando e implementando algoritmos que alcanzan el *óptimo débil* en su *complejidad de evaluación*, además de optimizar casos de borde. Todos los algoritmos importantes utilizados en nuestra implementación están detallados en el presente trabajo.

En nuestro diseño incluimos un motor de almacenamiento de bases de datos de grafos, que si bien su eficiencia para las consultas más simples es inferior a otras implementaciones existentes, tales como *DEX*, es la implementación más eficiente al evaluar consultas de mayor complejidad. Sin duda, con un soporte activo como el que recibe actualmente *DEX*, nuestra implementación podría mejorar aún más su eficiencia, llegando a alcanzar un puesto importante en el área. Se detalla la organización del código fuente en la implementación, incluyendo descripciones de los archivos y un diagrama de clases, así como ejemplos de uso de la intefaz de línea de comandos implementada.

Nuestro trabajo es un buen punto de partida para seguir desarrollando el tema de las bases de datos de grafos, considerando que el diseño y la implementación resultaron ser exitosas, además que propusimos un lenguaje de consultas formal, con una sintaxis y una semántica clara, además de cumplir buenas propiedades referentes a la expresividad y la complejidad del lenguaje.

Sin embargo, aún existen muchas mejoras posibles a implementar, además de posibles extensiones del lenguaje con el fin de hacerlo más expresivo, pero sin sacrificar su eficiencia.

Algunas de las ideas a desarrollar a futuro son:

- Implementar en un lenguaje más eficiente los métodos de acceso a disco. Un candidato es considerar el lenguaje *C/C++*.
- Investigar una estructura de datos alternativa para almacenar el grafo en memoria externa, con el fin de poder computar las consultas básicas lo más eficientemente posible.
- Estudiar algoritmos que permitan resolver nuestro problema considerando que no es posible almacenar conjuntos de tamaña $O(|V|)$ memoria principal.
- Determinar algoritmos que sean óptimos en un sentido más fuerte.
- Realizar experimentos más avanzados con el fin de evaluar de forma más exacta la complejidad empírica de la implementación.
- Extender nuestro modelo de base de datos, agregando atributos a los arcos y/o considerar arcos entre conjuntos arbitrarios de nodos.
- Extender nuestro lenguaje de consulta con operadores de agregación y operadores de comparación en los valores de los atributos, por ejemplo $>$, $<$ y $!=$ para los casos numéricos y expresiones regulares para los casos de cadenas.

Bibliografía

[1] AllegroGraph 4.10. http://www.franz.com/agraph/allegrograph/, 2013.

[2] Angles, Renzo. A comparison of current graph database models. En Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on. IEEE, 2012. p. 171-177.

[3] Angles, Renzo; Gutierrez, Claudio. Survey of graph database models. ACM Computing Surveys (CSUR), 2008, vol. 40, no 1, p. 1.

[4] Barcelo, P., Libkin, L., Lin, A. W., & Wood, P. T. (2012). Expressive languages for path queries over graph-structured data. ACM Transactions on Database Systems (TODS), 37(4), 31.

[5] Barceló, P., Pérez, J., & Reutter, J. L. Relative expressiveness of nested regular expressions. In Proceedings of the 6th Alberto Mendelzon Workshop on the Foundations of Data Management and the Web, 2012.

[6] Berners-Lee, T., Hendler, J., & Lassila, O. (2001). The semantic web. Scientific american, 284(5), 28-37.

[7] Bartels, Dirk, et al. The object database standard: ODMG 2.0. Eds. Roderic Geoffrey Galton Cattell, and Douglas K. Barry. Vol. 131. San Mateo, California, USA: Morgan Kaufmann Publishers, 1997.

[8] Bollacker, Kurt, et al. Freebase: a collaboratively created graph database for structuring human knowledge. Proceedings of the 2008 ACM SIGMOD international conference on Management of data. ACM, 2008.

[9] Bondy, John Adrian, and Uppaluri Siva Ramachandra Murty. Graph theory with applications. Vol. 290. London: Macmillan, 1976.

[10] Bray, Tim, et al. Extensible markup language (XML). World Wide Web Journal 2.4 (1997): 27-66.

[11] Buerli, M., & Obispo, C. P. S. L. (2012). The Current State of Graph Databases.

[12] Chamberlin, Donald D., and Raymond F. Boyce. SEQUEL: A structured English query language. Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control. ACM, 1974. p. 249-264.

[13] Clarke, Edmund M., Orna Grumberg, and Doron Peled. Model checking. The MIT press, 1999.

[14] Codd, E.F. (1970). A Relational Model of Data for Large Shared Data Banks. Communications of the ACM 13 (6): pp. 377–387.

[15] Cypher Query Language. Home. http://docs.neo4j.org/chunked/stable/cypher-query-lang.html, 2012.

[16] de Solla Price, Derek J. Networks of scientific papers. Science 149.3683 (1965): 510-515.

[17] DeCandia, Giuseppe, et al. Dynamo: amazon's highly available key-value store. ACM Symposium on Operating Systems Principles: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. Vol. 14. No. 17. 2007.

[18] Fastutil. Fast & compact type-specific collections for Java. http://fastutil.di.unimi.it/, 2013.

[19] Fischer, M. J., & Ladner, R. E. (1979). Propositional dynamic logic of regular programs. Journal of Computer and System Sciences, 18(2), 194-211.

[20] Graph Database. Wikipedia. http://en.wikipedia.org/wiki/Graph_database, 2013.

[21] Graves, M., Bergeman, E. R., & Lawrence, C. B. (1995). Graph database systems. Engineering in Medicine and Biology Magazine, IEEE, 14(6), 737-745.

[22] Guava. The Guava project. http://code.google.com/p/guava-libraries/, 2013.

[23] Gyssens, Marc, Laks VS Lakshmanan, and Iyer N. Subramanian. Tables as a paradigm for querying and restructuring. Proceedings of the fifteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. ACM, 1996.

[24] HyperGraphDB. Home. http://www.hypergraphdb.org/index, 2012.

[25] InfiniteGraph. Home. http://www.infinitegraph.com, 2012.

[26] Introducing the Knowledge Graph: things, not strings. http://googleblog.blogspot.co.uk/2012/05/introducing-knowledge-graph-things-not.html, 2012.

[27] Kasneci, Gjergji, et al. The YAGO-NAGA approach to knowledge discovery. ACM SIGMOD Record 37.4 (2009): 41-47.

[28] Kleinberg, J. M., Kumar, R., Raghavan, P., Rajagopalan, S., & Tomkins, A. S. (1999). The web as a graph: Measurements, models, and methods. In Computing and combinatorics (pp. 1-17). Springer Berlin Heidelberg.

[29] Kuper, G. M., & Vardi, M. Y. (1993). The logical data model. ACM Transactions on Database Systems (TODS), 18(3), 379-413.

[30] Levene, Mark, and Alexandra Poulovassilis. An object-oriented data model formalised through hypergraphs. Data & Knowledge Engineering 6.3 (1991): 205-224.

[31] Libkin, Leonid, Wim Martens, and Domagoj Vrgoc. Querying Graph Databases with XPath. ICDT, 2013.

[32] Martínez-Bazan, Norbert, et al. Dex: high-performance exploration on large graphs for information retrieval. Proceedings of the sixteenth ACM conference on Conference on information and knowledge management. ACM, 2007.

[33] Mackenzie, Charles E. Coded-Character Sets: History and Development. Addison-Wesley Longman Publishing Co., Inc., 1980.

[34] Nadeau software consulting. Java tip: How to read files quickly. http://nadeausoftware.com/articles/2008/02/java_tip_how_read_files_quickly, 2008.

[35] Neo4j. Home. http://neo4j.org, 2013.

[36] NoSQL Databases. Home. http://nosql-database.org/, 2013.

[37] Olson, Michael A., Keith Bostic, and Margo Seltzer. Berkeley db. Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference. 1999.

[38] OWL. Web Ontology Language. http://www.w3.org/TR/owl-features/, 2009.

[39] Poulovassilis, Alexandra, and Stefan G. Hild. Hyperlog: A graph-based system for database browsing, querying, and update. Knowledge and Data Engineering, IEEE Transactions on 13.2 (2001): 316-333.

[40] RDF. Resource Description Framework. http://www.w3.org/RDF, 2013.

[41] Schmidt, Albrecht R., et al. The XML benchmark project, 2001.

[42] SPARQL. Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/, 2013.

[43] Sparsity-Technologies. Home. http://www.sparsity-technologies.com, 2013.

[44] Trinajstiac, N. Chemical graph theory. Boca Raton, FL: CRC press, 1983.

[45] Wasserman, Stanley, and Katherine Faust. Social network analysis: Methods and applications. Vol. 8. Cambridge university press, 1994.

[46] Ramas, Mauro San Martın, Claudio Gutierrez, and Peter T. Wood. SNQL: Social Networks Query Language. (2011).

[47] World Data Center for Climate. Statistics. http://www.dkrz.de/daten-en/wdcc/statistics, 2012.

[48] World Wide Web Consortium. Home. http://www.w3.org/, 2013.