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

BÚSQUEDA DE TEXTO EN MILLENNIUMDB

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

JOSÉ IGNACIO BERROCAL CONTRERAS

PROFESOR GUÍA:
AIDAN HOGAN

MIEMBROS DE LA COMISIÓN:
BENJAMÍN BUSTOS CÁRDENAS
JUAN BARRIOS NUÑEZ

SANTIAGO DE CHILE
2022

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN
POR: JOSÉ IGNACIO BERROCAL CONTRERAS
FECHA: 2022
PROF. GUÍA: AIDAN HOGAN

BÚSQUEDA DE TEXTO EN MILLENNIUMDB

Las bases de datos de grafos se han posicionado como alternativa válida para programadores que requieren de una base de datos más a la medida para su problema que lo que les ofrecen las bases de datos tradicionales, esto se debe, mayoritariamente, a que las base de datos de grafos proporcionan mayor libertad al momento de definir y modificar *schemas*, y mayor expresividad en su lenguaje de consulta.

Por otro lado, la poca estructuración que estas bases de datos ofrecen también traen problemas, en especial a sus usuarios finales, los que tendrán mayor dificultad para poder identificar elementos en ésta y así poder navegar estas base de datos de forma eficiente.

En este trabajo de título se busca implementar en MillenniumDB, una nueva base de datos de grafos, búsqueda de texto completo con índices invertidos para mitigar parte del problema previamente mencionado. Con esta nueva funcionalidad, entonces, un usuario no necesitaría conocer previamente identificadores internos para los elementos de una GraphDB, sino que tendrá la posibilidad de buscar y explorarla a través de texto parcial.

Para esté objetivo, se diseñó y desarrolló un índice de texto, para luego implementar su lógica en el motor de MillenniumDB.

Para el diseño y lógica interna del índice primero se realizó una búsqueda preliminar sobre bibliotecas de búsqueda de texto, donde se llego a la conclusión que Lucene++, un *port* de Lucene a C++, era el más adecuado. Luego se implemento la lógica interna del índice utilizando éste *framework*.

Al momento de la validación de la solución desarrollada se consideraron 4 métricas: el tiempo de creación del índice, el tamaño de éste, el tiempo de búsqueda dado el tamaño del índice y el tiempo de consultas en el grafo que incluyen consultas. Las últimas son las que más afectan la experiencia de un usuario final de esta base de datos. Sin embargo esta solución tiene limitaciones al no estar del todo conectada al lenguaje de consultas de MillenniumDB por lo que no es transparente para el usuario final.

Este trabajo presenta una solución a algunos de los problemas que conlleva la utilización de base de datos de grafos para sus usuarios. Así pues como trabajo futuro se propone modificar la sintaxis del lenguaje de consulta para que la utilización de esta solución sea transparente para el usuario final.

Tabla de Contenido

1. Introducción **1**
1.1. Problema . . . 1
1.2. Objetivos . . . 2
1.3. Estructura de la Memoria . . . 2

2. Marco Teórico **4**
2.1. Recuperación de la Información . . . 4
2.1.1. Índices Invertidos . . . 4
2.1.2. Relevancia . . . 6
2.1.3. Lucene . . . 7
2.2. Modelos de Base de Datos de Grafos . . . 7
2.2.1. Edge Labelled Graph . . . 7
2.2.2. Property Graph . . . 8
2.3. Motores y Lenguajes de Consulta para Base de Datos de Grafos. . . . 9
2.3.1. RDF y SPARQL . . . 10
2.3.2. Neo4j y Cypher . . . 11
2.4. MillenniumDB . . . 11
2.4.1. Modelo de grafo . . . 12
2.4.2. Lenguaje de consulta MillenniumDB . . . 12

3. Problema **13**
3.1. Objetivo . . . 13
3.2. Desafíos . . . 13

4. Bibliotecas de Índices Invertidos **15**
4.1. Características de la máquina utilizada . . . 15
4.2. Selección de Índice Invertido . . . 15
4.3. Datos Experimentales . . . 16
4.4. Resultados Experimentales . . . 16
4.4.1. Tamaño del Índice en Disco . . . 17
4.4.2. Tiempo de Búsqueda . . . 18

5. Integración con MillenniumDB **19**
5.1. Diseño del Índice . . . 19
5.2. Búsqueda en Lucene++ . . . 20
5.3. Creación del Índice . . . 20

5.4. Búsqueda sobre el Índice 21
5.5. Implementación de búsqueda de texto en MillenniumDB 21
5.6. JOINS 22

6. Experimentos y Resultados **25**
6.1. Datos Experimentales 25
6.2. Validación del índice 26
6.3. Búsqueda En El Índice 27
6.4. Validación MillenniumDB 28

7. Conclusión **29**
7.1. Trabajo Realizado y Objetivos 29
7.2. Trabajo Futuro 30

Índice de Tablas

2.1. Documentos a indexar 4
2.2. Índice invertido 5
2.3. Índice Invertido 5

4.1. Partición del Corpus 16

Índice de Ilustraciones

1.1. Ejemplo de una base de datos representada en grafo 2

2.1. Ejemplo de Edge Labelled Graph 8
2.2. Ejemplo de Property Graph Model 9

4.1. Tiempo de Construcción Índice 17
4.2. Tiempo de Construcción Índice para Lucene++. 17
4.3. Tamaño de Índice en Disco para CLucene. 18

6.1. Tamaño del Índice 26
6.2. Tiempo de Indexación Wikidata 27
6.3. Tiempo de Búsqueda: 100 más relevantes 27
6.4. Tiempo de Búsqueda: Todos los documentos relevantes 28
6.5. Tiempo de Creación en MillenniumDB con indexación 28

Capítulo 1

Introducción

Las bases de datos relacionales han sido la opción estándar para el almacenamiento de datos, pero el alza en el volumen y la diversificación en la estructura de estos ha permitido un aumento en el uso de base de datos no tradicionales [14]. Esta aseveración toma aún más prevalencia con bases de datos de grafos (GraphDB), las que, como su nombre lo indica, permiten representar datos en estructura de vértices y aristas [12].

Esta representación da una mayor expresividad sobre las relaciones entre entidades, lo que permite a un usuario poder modelar con mayor facilidad distintos elementos del mundo real. El modelo de GraphDB se separa aún más del esquema por el cual las bases de datos relacionales se conocen, si se considera su flexibilidad para adaptarse a datos en constante cambios al no tener schema fijo y su reducido tamaño comparado con otros modelos [13].

Aunque ha aumentado el uso de las GraphDB más populares, tales como Neo4j y ArangoDB [1], y el uso en general de las GraphDB está en alza, no queda duda que este desarrollo es reciente. Antes de este resurgimiento en el interés por tales sistemas, la atracción por investigar tanto este modelo de base de datos como sus lenguajes de consultas era limitado [10], efecto que aún se puede notar en el rendimiento de estos sistemas [18], los que aún tienen cabida para ser optimizados.

Es por esta razón que desde el Instituto Milenio de Fundamento de los Datos nace la iniciativa de crear un nuevo sistema de GraphDB *open source* con optimizaciones del estado del arte. Desde esta iniciativa nace MillenniumDB [25], del cual ya se encuentra un primer prototipo desarrollado con su respectivo lenguaje de consulta.

1.1. Problema

Debido a la estructuración más flexible de las GraphDB, no existe una forma amigable con la que un usuario pueda buscar datos; los usuarios finales de estos motores de base de datos solo podrían trabajar con los identificadores internos para cada elemento del grafo. Por ejemplo, no es posible buscar en el grafo de la Figura 1.1 a los jugadores con nombre "*david*", sino solo por el nombre completo: "*David Robinson*" o el identificador del nodo n_1.

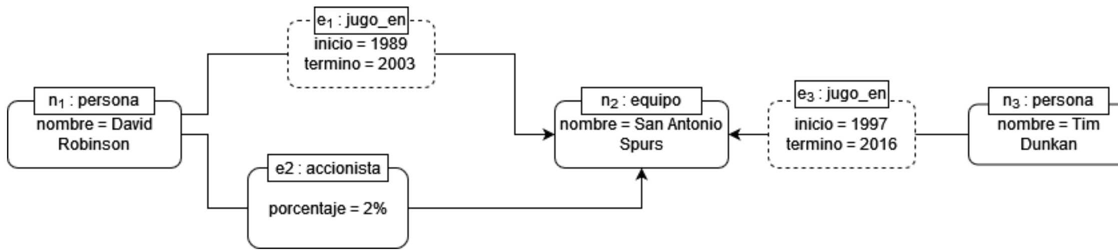

Figura 1.1: Ejemplo de una base de datos representada en grafo

Es por esto que surge otra feature característica de tales sistemas: la indexación y búsqueda sobre elementos de tal base de datos por su nombre o descripción en lenguaje natural. En este contexto, el objetivo del trabajo a realizar en esta memoria es implementar y evaluar búsqueda de texto sobre los elementos de MillenniumDB que tienen texto en lenguaje natural asociado. Por ejemplo, dada una búsqueda de texto como "*david*", en el contexto de la Figura 1.1, queremos poder devolver nodos como n_1, o los vecinos de estos nodos en el grafo, etc.

1.2. Objetivos

Objetivo General

El objetivo de esta memoria es implementar búsqueda de texto en MillenniumDB de forma eficiente, permitiendo integrar búsquedas de texto con consultas de grafo.

Objetivos Específicos

1. Elegir una biblioteca de búsqueda que permita implementar búsqueda por índice invertidos que, además, sea compatible con MillenniumDB.
2. Diseñar un índice para los elementos del grafo para permitir búsquedas eficientes sobre ellos que devuelva nodos asociados con texto que coincide con la búsqueda..
3. Desarrollar un software que pueda ordenar por relevancia los resultados de tales búsquedas.
4. Integrar la funcionalidad desarrollada con el prototipo existente de MillenniumDB.
5. Evaluar el rendimiento de la solución desarrollada (tiempo de respuesta, espacio de los índices creados, etc)

1.3. Estructura de la Memoria

Esta memoria está estructurada de la siguiente manera:

1. En el siguiente Capítulo se discute el marco teórico y los conocimientos previos necesarios para realizar este trabajo: Recuperación de la información, modelos conceptuales para estructurar las GraphDB, y motores de éstos.
2. En el Capítulo 3 se habla en más detalle sobre el problema y los desafíos que habrá que afrontar.

3. En el Capítulo 4 se discuten distintas herramientas para resolver el problema propuesto y se presenta un benchmark sobre ellas.
4. El Capítulo 5 presentar la manera de indexación que se utilizará en MillenniumDB. y el paso de búsqueda en este índice creado, además de explicar cómo esto se conectará al motor ya desarrollado al flujo de una consulta en MillenniumDB.
5. Por su parte, en el Capítulo 6 se discute la validación del trabajo realizado.
6. Y por último en el Capítulo 7 se discute el trabajo en su totalidad y las conclusiones.

Capítulo 2

Marco Teórico

En esta memoria se trabaja con conceptos de recuperación de la información y GraphDB; es por esto que en este Capítulo se detallan los elementos más importante de ambos tópicos. Primero se discutirá sobre herramientas de indexación y *ranking* de documentos con respecto a su texto; luego se mencionarán distintos modelos y lenguajes de GraphDB existentes. Por ultimo se explicará el modelo de MillenniumDB y sus diferencias con otros modelos de GraphDB.

2.1. Recuperación de la Información

Para poder obtener información sobre cualquier tipo de texto plano es necesario resolver dos problemas: El primero es tener una forma de guardar la información del texto que permita realizar búsquedas eficientes; y el siguiente problema es que los resultados de consultas sobre el índice sean relevantes a la consulta en sí.

2.1.1. Índices Invertidos

La forma más utilizada para resolver el problema de optimizar busqueda es índices invertidos [11] en el que se guardan strings y la ubicación de los documentos donde ocurre. Por ejemplo si se tienen los documentos de la Tabla 2.1, su índice sería como el que se ve en la tabla 2.2; y se interpreta de la siguiente manera: el string `"perro"` se encuentra en el documento 0 desde la posición del carácter 2 y en el documento 3 desde la posición 2.

$documento_0$:	"el perro ladra"
$documento_1$:	"el gato camina"
$documento_2$:	"josé camina por el parque"
$documento_3$:	"el perro muerde al gato "

Tabla 2.1: Documentos a indexar

Uno de los problemas que surgen de este modelo es su demanda en memoria, ya que crece rápidamente al aumentar la cantidad de documentos, el tamaño de estos y el tamaño del

el:	(0,0), (1,0), (2,15), (3,0)
perro:	(0,2),(3,2)
gato:	(1,2),(3,8)
camina:	(1,7),(2,4)
...	

Tabla 2.2: Índice invertido

vocabulario completo. Es por esto por lo que se han desarrollado métodos para poder reducir la sobrecarga que este sistema genera [20]. La más común es guardar solamente la información a nivel de documento y no la posición de la palabra en el documento. De esta forma, el tamaño del índice será limitado por la suma de las palabras únicas en cada documento, en vez de las palabras totales en cada documento. De esta forma se puede reducir el tamaño del índice, particularmente en el caso de indexar documentos muy largos (que tendrán más repeticiones de palabras). Pero se pierde la información de las posiciones y el orden de las palabras en el documento, lo cual significa que no se puede hacer búsquedas por frases exactas como `"gato camina"`.

Otras técnicas que contribuyen a la disminución del tamaño del índice es ignorar *stop words*; estas palabras son aquellas que no entregan mucha información sobre el texto mismo, pero que aparecen repetidamente en éste, como lo son algunos artículos, preposiciones, etc. Por ejemplo, si definimos los pronombres personales como *stop words* en el corpus de la tabla 2.1, el índice quedaría como se ve en la tabla 2.3.

perro:	(0,2),(3,2)
gato:	(1,2),(3,8)
camina:	(1,7),(2,4)
...	

Tabla 2.3: Índice Invertido

A priori este proceso no debería afectar las búsquedas: Un usuario buscando `"el perro"` esperaría obtener tanto $documento_0$ y $documento_3$. En la mayoría de los casos, quitar los *stop words* no afecta los resultados negativamente dado que tienen una frecuencia muy alta y así dan menos información sobre la intención de la busqueda.

Lematización y *stemming* también contribuyen a disminuir el tamaño total, aunque la finalidad de ambas técnicas no es disminuir tamaño, sino que es poder usar reglas básicas del lenguaje para extraer información más allá del literal que se está guardando.

Lematización hace referencia al proceso que trata de obtener el lema, o forma canónica, de una palabra desde sus derivados, por ejemplo, lematizar la palabra en inglés `"been"` resultará en `"be"` lo mismo para `"are"` `"is"` `"was"` etc. *Stemming*, por su parte, usa heurísticas para remover parte de una palabra con la intención de obtener la palabra raíz: aplicarle este proceso a `"replacement"` y `"replaced"` entonces, podría retornar `"replac"`. Dependiendo de las reglas impuestas, estos procesos puede enriquecer o entorpecer una búsqueda.

La compresión de los índices es otra característica común entre las distintas implementaciones de índices invertidos. No solo ayuda a reducir el espacio en memoria secundaria, sino también ayuda a reducir el tiempo necesario para evaluar una consulta dado que se puede guardar más información en memoria principal.. Existen distintas técnicas para este propósito; la mayoría se enfoca en codificar el índice en un arreglo de bits de tal forma que números muy altos sean comprimidos a un valor de menor tamaño [31].

2.1.2. Relevancia

Para el segundo problema planteado, se hace necesario definir ¿cuáles documentos son más relevantes para una búsqueda? Por este motivo definir más formalmente el problema se hace necesario:

Definición 2.1 ([21]) *Problema de Recuperación de Información*
Diremos que dado un set de documentos D, una consulta q como una secuencia de palabras, lo que se quiere obtener es un $D^* \subseteq D$ tal que maximice la siguiente probabilidad:

$$P(d|q, D) \text{ con } d \in D^*.$$

La probabilidad definida acá hace referencia a la probabilidad de que el usuario que haga la consulta q encuentre relevante el documento $d \in D^*$. Entonces el problema central es de definir medidas de relevancia que intentan aproximar esta probabilidad. Luego, una forma de estimar la eficacia de una medida de relevancia es comparar sus resultados con el número de usuarios que visitan d cuando busquen q; el último nos da una estimación de la probabilidad definida antes. El problema con este modelo es que solo dice si un documento es relevante o no. No dice qué tan relevante es el documento, ni cuál documento es lo más relevante.

Una solución a este problema que es simple de implementar es el ***Boolean Model***; en este modelo los documentos se manejan como conjuntos, los predicados son un conjunto específico de términos y las consultas son expresiones booleanas. Por ejemplo, si usamos los documentos de la tabla 2.1, el conjunto de términos será *{perro,gato,josé,parque,...}*. La consulta $q = perro \wedge gato$ daría como resultado $documento_3$.

Otra solución a este problema se encuentra en ***Vector Space Model*** o *VSM*. En este modelo tanto la consulta como los documentos son representados como vectores de un mismo espacio y cada término del índice corresponde a una dimensión específica de este espacio. Para resolver una consulta q se computa una medida de relevancia tal como la similitud entre la consulta y los distintos documentos y se ordenan según su resultado. En general se calcula la similitud coseno de la siguiente manera:

$$\textit{cosine-similarity}(q, d_i) = (q \bullet d_i)/(||q|| \cdot ||d_i||)$$

Donde $= (q \bullet d_i)$ indica el producto escalar de los dos vectores, y $||q|| \cdot ||d_i||)$ los módulos de los vectores.

Este fórmula nos da el coseno del angúlo entre ambos vectores. El valor específico de cada una de las coordenadas de un vector puede variar según la formulación que se esté usando, pero una de las soluciones más utilizadas es *tf-idf* o *term frecuency - inverse document frecuency.*

Term frecuency hace referencia a cuantas veces aparece en un documento un término en particular. Entre más veces aparece un término más relevante es ese documento para la consulta. A su vez, *inverse document frecuency* hace referencia a cuanta información una palabra entrega y se puede definir como sigue: Sea N el número de documentos en la colección, y n_i la cantidad de documentos que contienen el término t_i entonces *idf* de t_i es:

$$idf(t_i) \coloneqq log(N/n_i)$$

Por último para obtener *tf-idf*:

$$tf\text{-}idf(t_i, d) \coloneqq tf(t_i, d) \times idf(t_i)$$

2.1.3. Lucene

Lucene es una biblioteca *open-source* de Java que entrega capacidad para recuperación de información. Ha sido la biblioteca estándar en la industria para éste propósito; dentro de algunos de los proyectos más reconocidos que utilizan Lucene se encuentran ElasticSearch y Solr. El primero es un motor de búsqueda y análisis distribuido, y el segundo es una plataforma de búsqueda tanto para documentos en formato *pdf* o *docx* como para bases de datos.

Para utilizar Lucene se requiere definir documentos con campos específicos; estos documentos se indexarán y sobre ellos se realizará la consulta. Además se requiere especificar cuáles de estos campos serán contabilizados para el análisis de la consulta.

Para resolver una consulta, Lucene usa una combinación de *Boolean Model* y *VSM*; primero ejecutando *BM* y luego pasando los resultados de éste proceso al modelo vectorial. Acá Lucene procesa un score para cada documento con respecto al vector de la consulta.

2.2. Modelos de Base de Datos de Grafos

Los dos modelos de base de datos de grafos más utilizados son *Edge Labelled Graph Model* y *Property Graph Model* [8]. En ambos modelos un nodo representa una entidad, una arista es una relación entre entidades.

2.2.1. Edge Labelled Graph

En específico, *Edge Labelled Graph*, es un modelo que representa datos en un conjunto de tripletas: *subject*, *predicate*, y *object.* Cada una de estas tripletas representan un elemento en un grafo dirigido con aristas etiquetadas; *subject* y *object* representan nodos y los *predicates* representan las etiquetas de las aristas. La dirección de las aristas es de *subject* a *object.* En la Figura 2.1 se puede apreciar información de dos departamentos de la FCFM codificada en

un Edge Label Graph. Este grafo está descrito usando el *Resource Description Framework* (RDF) [28], que es un modelo de datos basado en *Edge Labelled Graphs*, propuesto para intercambiar datos sobre la web.

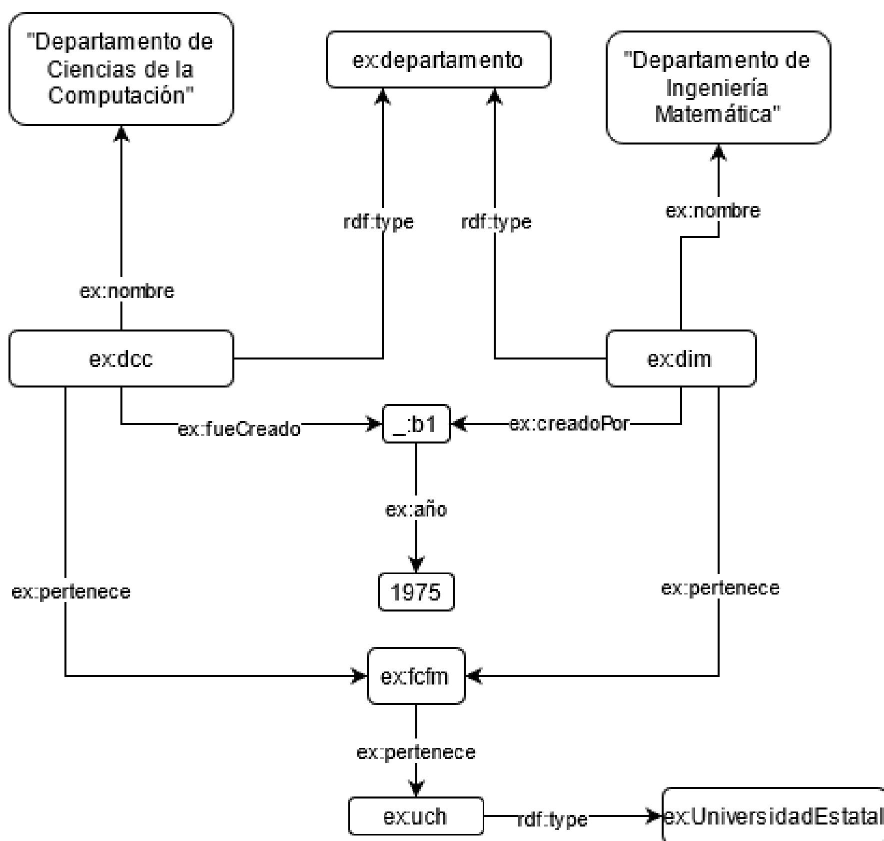

Figura 2.1: Ejemplo de Edge Labelled Graph

2.2.2. Property Graph

En un Property Graph los datos se pueden representar en un grafo dirigido con etiquetas donde, además, cada nodo y arista puede contener un conjunto de información en la forma de *{llave:valor}* llamados *properties*; estos elementos representan información específica de cada uno de los elementos del grafo [7]. Este modelo es usado especialmente si se tiene una gran cantidad de información en sistemas distribuidos [22]. En la Figura 2.2 se puede ver un ejemplo de un *property graph*.

En comparación con un *Edge Labelled Graph*, los *Property Graphs* son más complejos de manejar, pero permiten más flexibilidad en términos de poder agregar más información sobre las aristas. Por ejemplo, en la arista `e1` de la Figura 2.2, podemos especificar el año de la creación del Departamento de Ciencias de la Computación. Aunque existen mappings para expresar esta información en un Edge Labelled Graph [6], se requiere la conversión de una arista a un nodo, que constituye un cambio mayor al grafo [17].

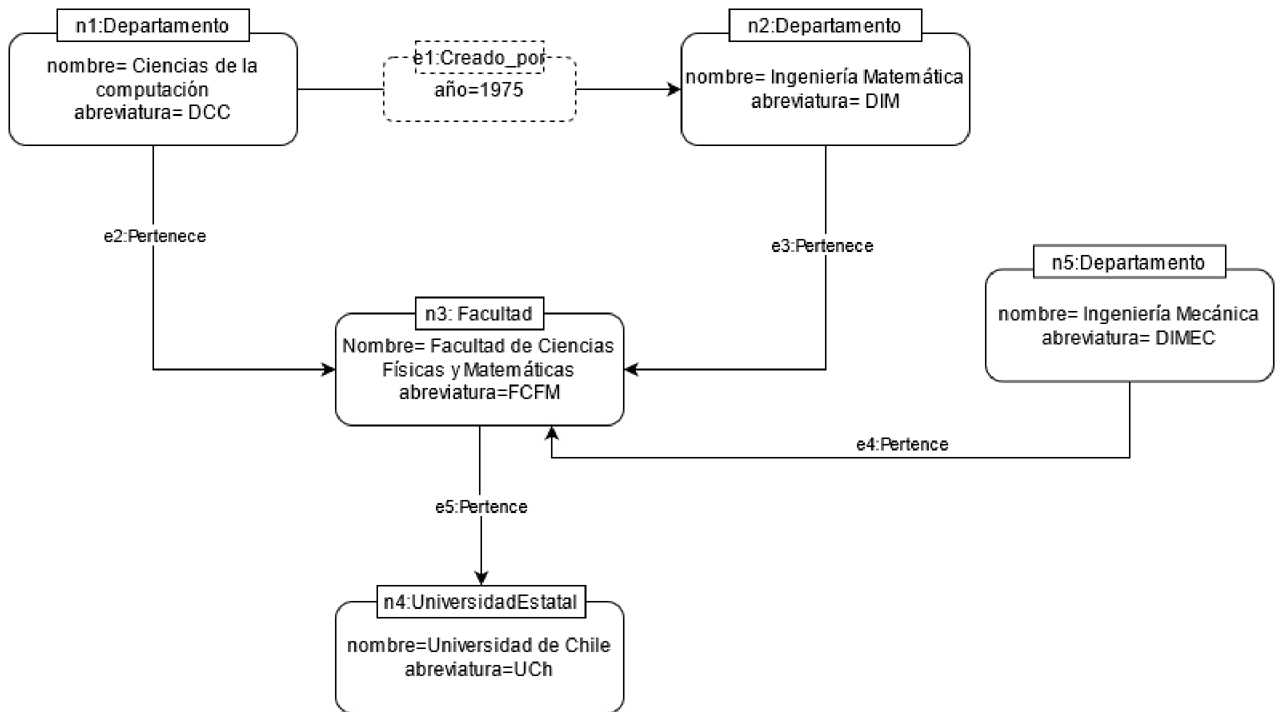

Figura 2.2: Ejemplo de Property Graph Model

2.3. Motores y Lenguajes de Consulta para Base de Datos de Grafos.

Para ambos modelos de grafos existen distintos motores y lenguajes de consultas asociados, pero gran parte de los motores exhiben un núcleo con operaciones similares [9]. Además de las operaciones compartidas con modelos relacionales como soportar álgebra relacional y agregaciones, los lenguajes de GraphDB presentan en su mayoría consultas de *Graph Patterns* y *Path Queries*. *Graph Patterns* es un tipo de consulta donde una variable aparece en lugar de cualquier elemento con la misma estructura que el modelo donde se encuentra [9].

Path queries, por su parte, son un tipo de consulta que permiten navegar la topología del grafo; la más simple de este tipo de consultas es saber si existe un camino entre dos nodos específicos de la base de datos [8].

En la práctica, los motores y lenguages de consulta para grafos pueden soportar otras funcionalidades más allá de Graph Patterns y Path Queries. Por ejemplo, pueden soportar consultas que usan el algebra relacional, agregaciones (como están incluidos en SQL), recursión, búsqueda de texto, etc.

En la siguiente sección, se presentan motores y lenguajes de consulta populares. Primero, hablaremos de motores para RDF (un modelo de Edge Labelled Graphs) y del lenguaje de consulta SPARQL. Luego hablaremos de Neo4j, un motor para Property Graphs, y su lenguaje de consulta Cypher.

2.3.1. RDF y SPARQL

Resource Description Framework [28] o *RDF* es un modelo de GraphDB basado en *Edge Labelled Graph*. Su lenguaje de consulta SPARQL se específica en [27]. Un ejemplo que combina *graph patterns* y *path queries* en SPARQL para el modelo definido en el grafo de la Figura 2.1 se presenta a continuación:

```
SELECT ?entidad
WHERE {
  ?entidad ex:pertenece+ ex:uch .
}
```

El resultado de tal consulta en el grafo de la Figura 2.1 sería el que sigue:

```
ex:dcc
ex:dim
ex:fcfm
```

Las variables en SPARQL se representan con un `"?"` al comienzo de su nombre. Por lo tanto en la consulta anterior se quiere buscar los nodos que tienen un camino a través de una o más aristas con etiqueta `"ex:pertenece"`, indicado con el operador `"+"`, al nodo `"ex:uch"`.

RDF, además, permite la utilización de *semantic extensions*, las que definen una serie de implicancias lógicas sobre los elementos del grafo de un RDF [29, 26]; el conjunto de elementos de un grafo de RDF se denomina *resources*.

Un ejemplo de *semantic extension* es RDF Schema la que provee mecanismos para describir grupos de *resources* relacionadas y las relaciones entre ellas [30]. Por ejemplo, en RDFS se puede definir que: *`ex:universidadEstatal rdfs:subClassOf ex:universidad`*. Si se agrega esta definición al grafo de la Figura. 2.1 entonces se puede concluir que *`ex:uch rdf:type ex:universidad`*.

Aunque SPARQL no ofrece búsqueda de texto completo, algunas de sus implementaciones, como *Virtuoso* [15], sí la implementan. Por ejemplo la siguiente consulta retorna todos los *subject* donde su *ex:nombre* sea relevante, como fue definida anteriormente, a la consulta `"Ingeniería Matemática"`.

```
SELECT ?x
FROM *
WHERE
  {
    ?x ex:nombre ?name .
    ?name bif:contains 'Ingeniería Matemática'.
  }
```

En este caso, el predicado `bif:contains` es interpretado como una búsqueda de texto por el sistema. El resultado de tal consulta aplicado a la Figura 2.1 sería:

```
ex:dim
```

2.3.2. Neo4j y Cypher

Un motor de base de datos de grafos basado en *Property Graph Model* es Neo4j; está implementado en Java y es la base de datos de grafos más utilizada en la práctica [1]. El lenguaje de consulta de Neo4j, llamado Cypher [16], soporta ambos tipo de consultas previamente mencionadas. A continuación se muestra un ejemplo de un *graph pattern* para el modelo de la Figura 2.2 donde se quiere buscar todo nodo con etiqueta `"Departamento"` que tenga una arista con etiqueta '`Pertenece`' al nodo con propiedad `{abreviatura:"FCFM"}`.

En Cypher, se puede expresar un *graph pattern* de la siguiente forma:

```
MATCH (x:Departamento) -[:Pertenece]-> (:Facultad{abreviatura:"FCFM"})
RETURN x.nombre;
```

Consulta que daría como resultado:

```
"Ciencias de la Computación"
"Ingeniería Matemática"
"Ingeniería Mecánica"
```

Además, Neo4j soporta búsqueda de texto con índices invertidos sobre las relaciones y los nodos del grafo de forma nativa, aunque estos índices no son creados al declarar el esquema, sino que el usuario tiene que declarar para qué elementos se quiere crear dichos índices [2]. Un ejemplo del proceso para indexar y buscar en Neo4j sobre el grafo descrito en Figura. 2.2 se presenta a continuación:

```
CALL db.index.fulltext.createNodeIndex
("FullNames", ["Departamento", "Facultad"], ["Nombre"])
CALL db.index.fulltext.queryNodes
("FullNames", "Ingeniería Matemática") YIELD node
RETURN node.nombre
```

Este código entregaría la siguiente información:

```
"Ingeniería Matemática"
"Ingeniería Mecánica"
"Facultad de Ciencias Físicas y Matemáticas"
```

La explicación del código es la siguiente: primero se crea el índice `"FullNames"` que indexará la propiedad `"Nombre"` de los nodos con etiqueta `"departamento"` y `"Facultad"`. Luego se busca en el índice creado el texto `"Ingeniería Matemática"`. El administrador de la base de datos puede indexar los datos consultados más frecuentemente antes de procesar las consultas.

2.4. MillenniumDB

MillenniumDB [25] es una base de datos de grafos que surge debido a que sus creadores en el Instituto Milenio Fundamento de los Datos piensan que las implementaciones de los

modelos existente de GraphDB no son suficientes para representar y evaluar eficazmente los diversos tipos de consultas de grafo que se encuentran en el mundo real; además tiene como objetivo ser una alternativa *open source* a diferencia de la mayoría de los sistemas de base de datos que se encuentran en el mercado que son comercializados. El sistema está implementado en C++.

2.4.1. Modelo de grafo

La estructura de grafos que utiliza MillenniumDB difiere tanto de *Property Graph* como de *Edge Labelled Graph*, aunque puede simular ambos de forma directa, en particular MillenniumDB usa *Domain Graph* como modelo, el que se define como sigue:

Definición 2.2 *Domain Graph*
Dado un universo Obj de objetos (strings, numeros, IRIs, etc)

$$Domain\ graph := (O, \gamma)$$

donde $O \subseteq Obj$ *y* $\gamma \colon O \longrightarrow O \times O \times O$

O se puede entender como los elementos de la base de datos, mientras que γ modela identificadores que mapean a aristas dirigidas y etiquetadas:

A modo de ejemplo, si se considera el grafo de la Figura 2.1 la tripleta *ex:fcfm ex:pertenece ex:uch* se puede traducir directamente a *domain graphs*: $\gamma(e) = (fcfm, pertenece, uch)$ donde e es un identificador para la arista.

De la misma manera, al considerar el grafo de la Figura 2.2,la arista entre los nodos n_1 y n_2 se codifica $\gamma(e_1) = (n_1, creado_por, n_2)$, mientras que la tupla (abreviatura,DCC) en el nodo n_1 se codifica $\gamma(e_2) = (n_1, abreviatura, DCC)$ y la tupla de la arista e_1 se traduce a $\gamma(e_3) = (e_1, año, 1975)$.

2.4.2. Lenguaje de consulta MillenniumDB

El lenguaje de consulta desarrollado para MillenniumDB es cercano a Cypher, aunque con modificaciones para mejor utilizar las fortalezas de *domain graphs*.

Entre estas modificaciones se encuentra tomar funcionalidades de otros lenguajes de consultas como SPARQL.

Un ejemplo de consulta, se puede ver a continuación.

```
SELECT ?x.name, ?y
MATCH (?x) -[:Pertenece]-> (?y)
WHERE ?x.abreviatura=="DCC"
```

En este caso si un usuario quisiera buscar por palabra clave a todos los elementos que en la propiedad "`name`" tengan texto relacionado a "`universidad`" no sería posible. Solo se puede consultar por valores completos y exactos.

Capítulo 3

Problema

Las GraphDBs al ofrecer mayor flexibilidad comparadas con las base de datos relacionales, pierde una forma amigable con la que los usuarios puedan buscar y encontrar datos más allá de las ofrecidas por su lenguaje de consulta. Toda consulta tiene que ser en base a los IDs internos de la base de datos o buscando literales de forma exacta.

Por esta razón se hace necesario poder recuperar información de los elementos de la GraphDBs que se encuentren en lenguaje natural de forma más flexible. Esta *feature* se hace aún más necesaria si se considera que las GraphDBs en general son utilizadas para procesar grandes cantidades de datos, por lo que buscar manualmente entre ellos los elementos relevantes no se hace posible.

3.1. Objetivo

El objetivo de forma general es permitir que MillenniumDB pueda resolver eficientemente consultas del tipo:

```
MATCH (?x) -[:Pertenece]-> (?y)
WHERE TEXTSEARCH(?y.name,"universidad")
RETURN ?y
```

En específico se requiere implementar la ejecución de la segunda línea del código anterior, para cualquier *property* del grafo. El resultado de este tipo de consulta debe ser tal que pueda ser operado por el lenguaje de consultas ya implementado por el equipo de MillenniumDB. Es importante notar que la consulta mostrada es solo un ejemplo. El objetivo es poder combinar búsqueda por texto con consultas de grafo generales.

3.2. Desafíos

Se identificaron cuatros desafíos que pueden afectar la solución final:

1. Encontrar una biblioteca de índices invertidos para C++, el lenguaje en el que se ha implemendado MillenniumDB, que se adecue a las necesidades del proyecto. Facilidad de integración, escalabilidad, volumen de datos posibles, etc.
2. Los grafos no son documentos como tal por lo que es necesario definir qué elementos de un grafo son los necesarios para indexar y obtener respuestas con sentido.
3. Modificaciones en el árbol de ejecución de las consultas: modificar la forma que MillenniumDB procesa las consultas puede traer consecuencias no intencionadas.
4. La solución implementada tiene que tener en consideración el volumen de los datos que se pretenden procesar en MillenniumDB: Grafos importantes como Wikidata tienen ciento de millones de elementos en su dataset [23].

Capítulo 4

Bibliotecas de Índices Invertidos

En este capítulo presentamos el trabajo hecho para seleccionar la implementación de índices invertidos más apta para ser integrada con MillenniumDB. Se muestran las distintas bibliotecas para búsqueda de texto y las métricas que se utilizaron para escoger entre éstos. También se discuten las limitaciones de estos sistemas.

4.1. Características de la máquina utilizada

Todos los experimentos realizados en esta sección y en el capítulo 6 fueron realizados en Windows Subsystem for Linux con Ubuntu 20.04, compilado en GCC 9.3.0 en una máquina con 8 GB de RAM, CPU con velocidad de reloj de 2.8 GHz y memoria secundaria HDD de 1TB.

4.2. Selección de Índice Invertido

Para poder integrar índices invertidos a MillenniumDB se debió explorar las distintas implementaciones de esto. Dado que el motor de MillenniumDB se implementa en C++, se buscaron los índices invertidos implementados en C o C++; encontrando dos clones de Lucene: Clucene y Lucene++. CLucene equivale a la version 2.3.2 de Lucene y Lucene++ o *lpp* [5] que implementan la mayor parte del funcionamiento de Lucene 3.0.

Ambas implementaciones tienen funcionalidades parecidas, y tienen las funcionalidades básicas que se necesitan para este trabajo, así que para elegir entre ambas, hicimos algunos experimentos para comparar su rendimiento y los recursos que usan empiricamente. En esta comparación, se consideran:

- *Los tiempos para construir el índice*: Tiempos menores son deseables para tener mejor escalabilidad y eficiencia en el momento de cargar datos grandes.
- *El tamaño en memoria secundaría el índice*: Tamaños menores son deseables para tener mejor escalabilidad y eficiencia en el momento de cargar datos grandes.

4.3. Datos Experimentales

Para seleccionar una alternativa se utilizó un corpus extraído de Project Gutenberg, un repositorio de más de 60.000 libros en el dominio publico; en particular se utilizó un corpus construido por Lahiri [19], el cual contiene 3036 libros manualmente preprocesados para remover meta-datos, información de licencia y notas de traductor.

Se realizó un survey sobre su velocidad de indexación y tamaño de los índices en ambos. Para estos experimentos se tomó un subconjunto del corpus de Lahiri de 2100 documentos, y para hacer experimentos a diferentes escalas, se separó el corpus en 15 conjuntos $\{D_i\}_0^{14}$ de tal forma que cada $D_i \subsetneq D_{i+1}$, es decir, que cada conjunto es más grande que el anterior. En la Tabla 4.1 se presenta información relevante sobre las particiones hechas al corpus para realizar los experimentos.

	número de documentos	tamaño (kB)	tamaño del vocabulario
D_0	10	2 321	37 646
D_1	20	5 952	87 004
D_2	25	10 185	126 835
D_3	50	13 385	193 551
D_4	75	22 369	327 837
D_5	100	29 051	389 267
D_6	200	93 609	840 600
D_7	400	174 505	1 590 504
D_8	500	243 945	2 276 856
D_9	600	311 226	2 750 946
D_{10}	900	376 106	3 196 054
D_{11}	1200	490 881	4 154 296
D_{12}	1500	700 537	5 971 654
D_{13}	1800	815 913	7 167 956
D_{14}	2100	855 758	7 570 726

Tabla 4.1: Partición del Corpus

4.4. Resultados Experimentales

En esta sección, revisaremos medidas relacionadas con la construcción del índice, en particular, el tiempo para indexar los documentos, y el tamaño del índice final. También revisaremos medidas relacionadas con el tiempo para hacer búsquedas en ambas implementaciones. Se usarán estas medidas para seleccionar la implementación de índices invertidos que será integrada con MillenniumDB.

Tiempo de Construcción del Índice

Se registró el tiempo de construcción de índice con respecto a la cantidad de archivos en CLucene y Lucene++.

Como se puede ver de las Figura 4.1 Lucene++ es en promedio más rápido que CLucene para construir el índice. Aunque la desviación estándar de éste ultimo es más significativa

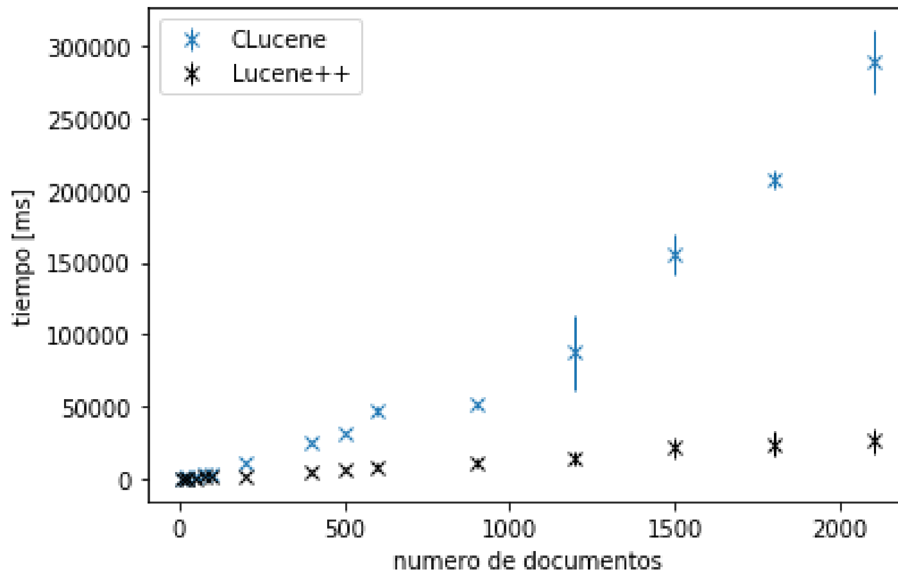

Figura 4.1: Tiempo de Construcción Índice

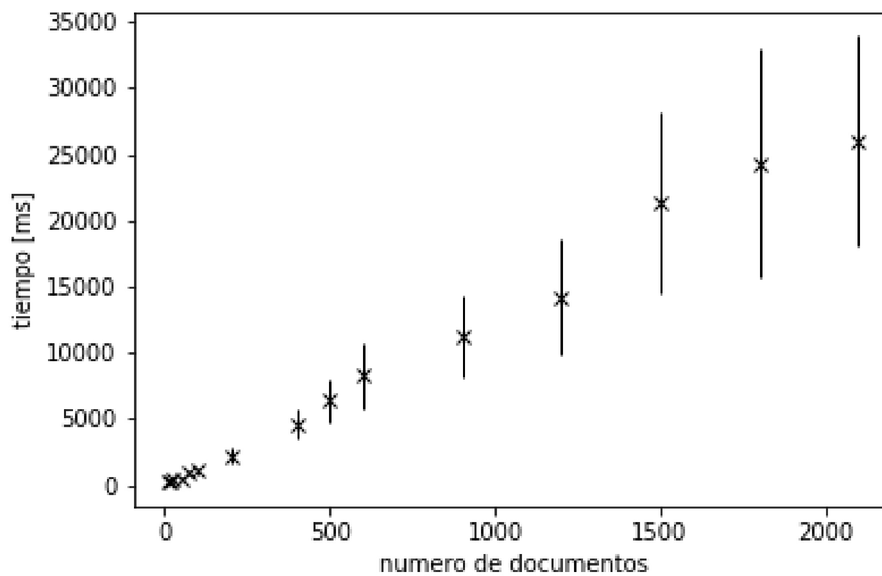

Figura 4.2: Tiempo de Construcción Índice para Lucene++.

comparativamente, la diferencia en rendimiento sigue siendo alrededor de 10 veces menor. En la Figura 4.2 se encuentra el tiempo de construcción solo de Lucene++ para mayor detalle. Se puede observar en esta última figura, además, que el tiempo de construcción del índice queda linealmente relacionado con el número de documentos, lo cual es una propiedad importante de escalabilidad.

4.4.1. Tamaño del Índice en Disco

De la misma manera, para cada construcción de índice, se midió el tamaño que éste ocupaba en disco. El resultado de esta medición se puede ver en la Figura 4.3.

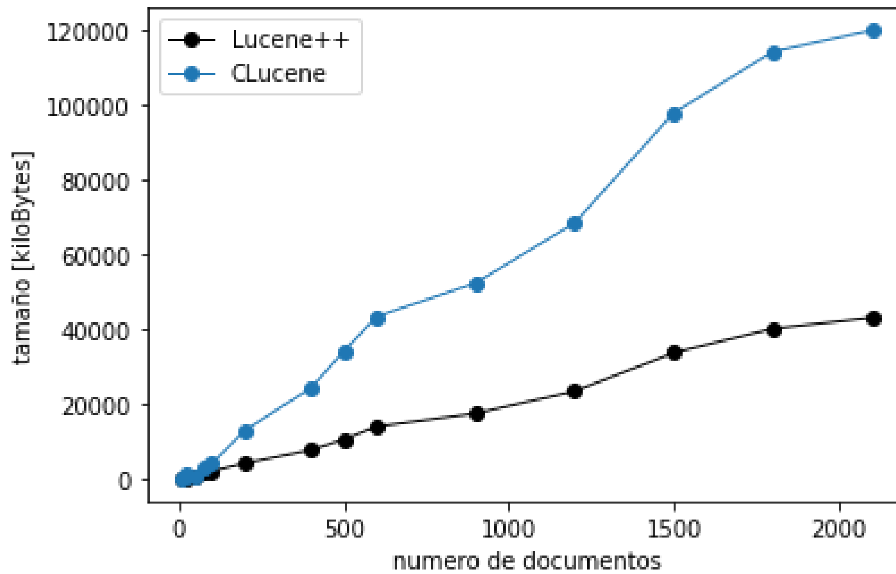

Figura 4.3: Tamaño de Índice en Disco para CLucene.

En esta métrica Lucene++ nuevamente supera a CLucene, pero la diferencia no es tan pronunciada como en el caso anterior. Estos resultados indican que Lucene++ cuenta con mejor compresión para texto que CLucene.

4.4.2. Tiempo de Búsqueda

Por último, se realizaron consultas sobre estos índices con palabras individuales aleatorias pertenecientes al corpus. Al realizar este experimento, se encontró que para ambas bibliotecas la primera consulta realizada se demoraba significativamente más que las consultas que la seguían, posiblemente relacionado con el tiempo tomado para cargar la estructura del índice en memoria principal. En CLucene este tiempo se encuentra en un intervalo de [280-320] ms independiente de la cantidad de documentos indexados; las consultas posteriores a esta tenían un tiempo de búsqueda de menos de 20 ms. Para Lucene++ el tiempo de la primera consulta se encuentra en un intervalo de [300-380] ms y el tiempo de las siguentes consultas fue menor a 15 ms.

Como se puede ver Lucene++ es más rápido y más liviano que Clucene al indexar, y solo un poco más lenta al momento de hacer la primera búsqueda, y más rápida al momento de seguir haciendo búsquedas. Por lo que se decidió seguir con Lucene++ para la implementación del índice.

Capítulo 5

Integración con MillenniumDB

En éste capitulo se discutirá la implementación de indexación y de búsqueda tanto a nivel de Lucene++, como a nivel de MillenniumDB.

5.1. Diseño del Índice

Una base de datos de grafo no indexa documentos, sino indexa grafos. En el caso particular de MillenniumDB, los grafos siguen el modelo de *domain graphs*. Por eso fue necesario diseñar un esquema de índice para los *domain graphs*, indicando cuáles serían los documentos indexados, qué contenido van a tener estos documentos, qué elementos se usarán para formar la lista de términos, y qué elementos deberían devolver de la lista de registros. Así que se definió el documento de Lucene++ con la información necesaria para poder implementarlo en el motor de la base de datos. Para definir la información que vamos a indexar, se organizó una reunión con los desarrolladores principales de MillenniumDB: Domagoj Vrgoc y Carlos Rojas. El propósito de la reunión fue definir más precisamente la funcionalidad deseada con respecto a la búsqueda de texto en el motor antes de diseñar el índice.

Se decidió que para cada par (key, propiedad) asociada a un nodo se crea el siguiente documento:

- node_id: La id del nodo donde se encuentra la *propiedad*.
- key_id: la id de la key de la propiedad.
- value_text: el texto plano asociado a la propiedad.

Solo *value_text* y *key_id* son indexados para que Lucene++ pueda buscar sobre ellos, el primero con índices invertidos en texto plano y *key_id* es indexado como un *numeric field*.

Entonces, por ejemplo, para el grafo de la Figura 2.2, el nodo n1 tendría dos documentos asociados:

- node_id:n1
- key_id: [id interna asociada a la propiedad *nombre*] .

- value_text: Ciencias de la computación

- node_id:n1
- key_id: [id interna de MilleniumDB asociada a la propiedad *abreviatura*] .
- value_text: DCC

Y para realizar una búsqueda del tipo `x.nombre=''computación''` se convierte el string "`nombre`" a la id interna que tiene MillenniumDB para este string, y se realiza una búsqueda conjunta sobre ese id y el texto. `''universidad''`.

5.2. Búsqueda en Lucene++

Antes de poder realizar consultas en un índice definido en Lucene++ se tienen que definir distintos objetos que nos permiten configurar las búsquedas en el índice para una solución más a la medida. Éstos objetos están condicionado tanto por el dataset como por el problema que se quiere solucionar.

En particular, al igual que la indexación, se tiene que definir un *analyzer* encargado de *tokenizar* nuestra consulta. También se tiene que definir un objeto de una clase que implemente la clase abstracta *collector*; éste objeto se encarga de ordenar y filtrar los documentos luego de que se le calcule un puntaje de relevancia para una consulta en específico.

Para nuestro índice la preferencia por un analyzer específico pasaba por o *StandardAnalyzer* o algún *Analyzer* en algún lenguaje en particular. El primero no provee mayor lógica asociada a algún idioma, por ende no provee lematización, aunque sí tiene normalización de texto eliminando puntuación y capitalización de palabras. Por otro lado un *analyzer* especifico como *EnglishAnalyzer* sí tiene lematización e ignora *stopword* específicas, esto como ya se mencionó, puede entorpecer o enriquecer la búsqueda dependiendo del dataset a indexar; por ejemplo si nuestra base de datos tuviese muchos literales en distintos idiomas. Es por eso que se decidió por *StandardAnalyzer* aunque cambiarlo no es mayor problema.

Para el *Collector* se decidió por *TopScoreDocCollector* el que permite entregar los n documentos con mejor score. Esto debido a que se definió que la búsqueda retornara inicialmente los primeros 100 resultados, y solo si el usuario lo requería entregara la totalidad de resultados.

5.3. Creación del Índice

MillenniumDB por el momento está desarrollada para migrar desde otras base de datos de grafos en un solo proceso: la clase *BulkImport*. El propósito de esta clase es el de leer los datos desde un archivo (por ejemplo, en formato JSON), y convertirlos e indexarlos (principalmente usando árboles B+) como un grafo, por lo que aquí es donde se implementa la creación del índice invertido.

En particular, Existían dos opciones viables donde implementar la indexación, por un lado se puede indexar todo luego de terminado todo el proceso de importado, esto implicaría reobtener los datos ya guardados en memoria secundaria para procesarlos nuevamente y crear

el índice, pero permite que el usuario tenga la opción de no crear un índice invertido si así lo desease, o crearlo en otro momento.

Por otro lado también se podía modificar el método *process_node*, el que procesa cada propiedad asociada a un nodo, creando el objeto correspondiente (tanto la key como el value) en la base de datos si no existe. Esta opción es más eficiente ya que no requiere reobtener datos, sino que los datos necesarios ya se encuentran en memoria, pero la indexación se realiza necesariamente en paralelo a la creación de la base de datos. Por como esta pensado MillenniumDB y *BulkImport*, es decir, que se cree todo en un solo proceso, se termino tomando la decisión de diseño de utilizar la opción más eficiente, que es la ultima opción descrita.

5.4. Búsqueda sobre el Índice

Al ejecutar una consulta en MillenniumDB, se va escribiendo en un segmento específico de memoria los resultados de aquella.

Por esto, al implementar la búsqueda sobre el índice creado, esta funciona de la siguiente manera: Busca en el índice la consulta dada y retorna los primeros 100 resultados más relevantes, si se requieren más documentos se retornarían todos los documentos con un score no nulo.

Aunque para la mayoría de las consultas esto es posible, el numero de documentos retornados está limitado por el tamaño de memoria primaria asignado. Para 8GB de memoria este limite era de aproximadamente 20 millones de documentos. Por otro lado para este volumen de resultados, los documentos con puntaje muy cercano a cero tienen poca relación con la consulta inicial.

5.5. Implementación de búsqueda de texto en MillenniumDB

MillenniumDB tiene distintos operadores para cada tipo de consultas:

1. **Operadores base** son los operadores que generan resultados base sobre el grafo. Incluyen *graph patterns*, *path queries*, etc.
2. **Operadores unarios** son operadores que reciben un operador como argumento, posiblemente con otros argumentos como constantes o condiciones; por ejemplo, selección (σ), proyección (π).
3. **Operadores binarios** son operadores que reciben dos operadores como argumentos, posiblemente con otros argumentos como constantes o condiciones; por ejemplo, join ($\bowtie$), únion ($\cup$), diferencia ($-$), etc.

Estos operadores luego se traducen a iteradores los que se usan para representar un flujo de resultados desde un operador de consulta. Estos iteradores transforman los resultados de entrega a resultados de salida según la semántica del operador; por ejemplo, un iterador de selección tomará un iterador de entrada y una condición, y escribirá al iterador de salida todos los elementos de la entrega que satisfagan la condición.

En nuestro caso, se desarrolló el *iterator TextSearch*. Esta clase implementa la interfaz *BindingIdIter* del motor; de esta interfaz nos interesan los 2 siguientes métodos:

1. *void begin()*: Como su nombre lo indica es ejecutado al comienzo de una consulta de MillenniumDB y configura todo lo necesario para que esta se lleve acabo.
2. *bool next()*: Este método escribe en una dirección dada de memoria la id de un objeto de MillenniumDB. Este id es el resultado de la operación que se estuviera realizando. Este método, además, se puede llamar múltiples veces para así obtener todos los resultados pertinentes. Si no quedan resultados retorna *false*, de lo contrario *true*.

En nuestro caso, *begin()* se encarga de realizar el pre-procesamiento descrito anteriormente, es decir, buscar los 100 mejores matches para la consulta dada, además, computa la cantidad total de resultados para esa consulta; y *next()* se encarga de retornar los resultados de la consulta de Lucene++ de forma secuencial y de procesar todos los resultados si se requieren más de 100 de ellos.

5.6. JOINS

Por último, para verificar el funcionamiento conjunto tanto de la búsqueda en Lucene++ y MillenniumDB se desarrolló una clase prototipo que realizará consultas en MillenniumDB con índices invertidos saltándose el lenguaje de consulta en sí y que actuara de forma similar al tipo de consultas que se desean realizar.

Por ejemplo, tomando en cuenta una consulta tipo:

```
MATCH (?x) -[:Pertenece]-> (?y)
WHERE TEXTSEARCH(?y.name,"universidad")
RETURN ?y
```

La podemos traducir, en pseudocódigo, a:

```
query="(?x) -[:Pertenece]-> (?y)"
key_id=get_object_id("name") //Método para obtener id \
//desde un objeto en la base de datos.
iterator=TextSearch(key_id,"universidad") //Búsqueda en Lucene++ \
//que entrega las ids de los elementos.
TextSearchJoin(iterator,query)
```

Para implementar este TextSearchJoin() se identificó dos opciones:

1. *Nested-loop join*: genera, para cada resultado de búsqueda, una consulta de grafo reemplazado la variable de búsqueda con el constante del resultado.
2. *Hash-join*: indexa los resultados de la búsqueda en memoria principal, evaluar el resto de la consulta una vez (sin reemplazar la variable de consulta), y para cada resultado generado acá, emitir solo los resultados donde la variable de búsqueda se encuentra en los resultados de búsqueda en memoria principal.

Entre las opciones de nested-loop y hash, se eligió nested-loop porque el hash-join requiere

generar todos los resultados para el resto de la consulta, algo que puede ser muy grande. En comparación, con el límite de n resultados por búsqueda, el número de subconsultas que habrá que evaluar con nested-loop es acotado.

En pseudocódigo, este iterador quedaría de la siguiente manera:

```
TextSearchLJoin implementa BindingIdIter{

// constante indicando el número de resultados dado para la búsqueda
int n

// variable de búsqueda en la consulta
var V

// palabras clave de la búsqueda
string Keywords

// plan del resto de la consulta de grafo sin la búsqueda de texto
Query q

// índice invertido implementado en Lucene++
TextIndex InvIndex

// iterador de resultados de consulta generado por Lucene++
SearchResultIter searchResults

// iterador de resultados de consulta generado por MillenniumDB
BindingIdIter queryResults = null

// el resultado actual
BindingId current = null

TextSearch(int n, str keywords, query q, TextIndex invIndex)

void begin()
  searchResults = TextSearch.begin();

bool next()
  if(queryResults!=null && queryResults.next())
    current = queryResults.current()
    return true
  else if(searchResults.next())
    r = searchResults.current();
    q_r = replace(q,V,r); // reemplaza variable v por constante r
    queryResults = evaluate(q_r); // evaluación por el motor de MillenniumDB
    next()
```

```
  else return false

BindingId current()
  return current()

  }
```

Capítulo 6

Experimentos y Resultados

En este capítulo se discutirá la validación de la solución creada, el objetivo de esto es tener una base experimental con la que comparar a otras bases de datos de grafo con búsqueda de texto invertida.

En particular primero se explica el dataset utilizado: *Wikidata*, y luego se discuten métricas considerando solo el índice invertido en Lucene++: tiempo de indexación, tamaño del índice en disco y búsqueda en éste. Y luego se discuten métricas conjuntas de MIllenniumDB con Lucene++.

6.1. Datos Experimentales

Como se mencionó, para estos experimentos se utilizó *Wikidata*, una base de conocimiento colaborativo elaborada para almacenar la información estructurada de todos los proyectos de Wikimedia [4]. Surge como respuesta a la necesidad de poder buscar, analizar y reutilizar información de Wikipedia y Wikimedia [24].

Wikidata se organiza en base a *items*, cada uno de estos puede representar cualquier tipo de objeto o concepto. Más allá de esto, la información se organiza en pares de *property-value*, así por ejemplo el *item* `Santiago` pudiese tener una *property* `founded by` con valor *Pedro de Valdivia*.

Wikidata pone frecuentemente a disposición un *datadump* con toda su información disponible, éste está estructurado como una base de datos en RDF. En particular el datadump es un archivo de texto donde cada tripleta se ve de la siguiente manera:

```
<http://www.wikidata.org/entity/Q31> <http://schema.org/description>

"country in Europe"@en .
```

los datos usados en estos experimentos ésta basada en un *datadump* del 2021-06-23 y es el mismo dataset que se utilizo para los *benchmarks* realizados por la publicación *Millen-*

niumDB: A Persistent, Open-Source, Graph Databaset [25].

6.2. Validación del índice

Como se mencionó, primero se evaluó el indice invertido solo desde Lucene++. Para cada tripleta con un literal se creo un documento de Lucene++ donde el *predicates* actuaría como key y *object* como value. Para el campo node_id ID utilizó el numero de tripletas vistas hasta el momento.

Entonces, para la tripleta descrita en la sección anterior el documento asociado es:

- node_id: 500.
- key_id: -.
- value_text: country in Europe.

El valor *key_id* es proporcionado al guardar elementos en la base de datos por lo que para efectos de estos experimentos se simuló como un valor entero aleatorio.

En la Figura 6.1 se muestra la evolución del tamaño del índice, el cual es comparable al tamaño del conjunto de datos completo; esto condice con lo esperado ya que se esta guardando aproximadamente la misma información.

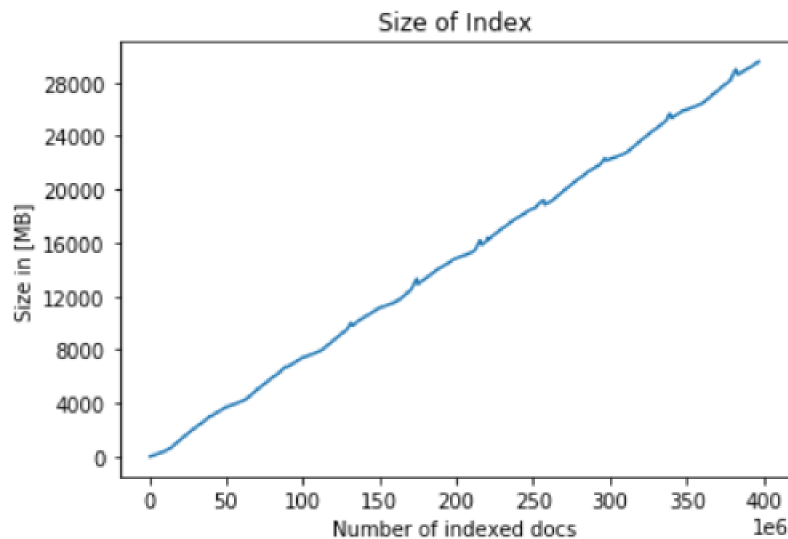

Figura 6.1: Tamaño del Índice

El tiempo total de indexación de las tripletas se puede ver en la Figura 6.2.

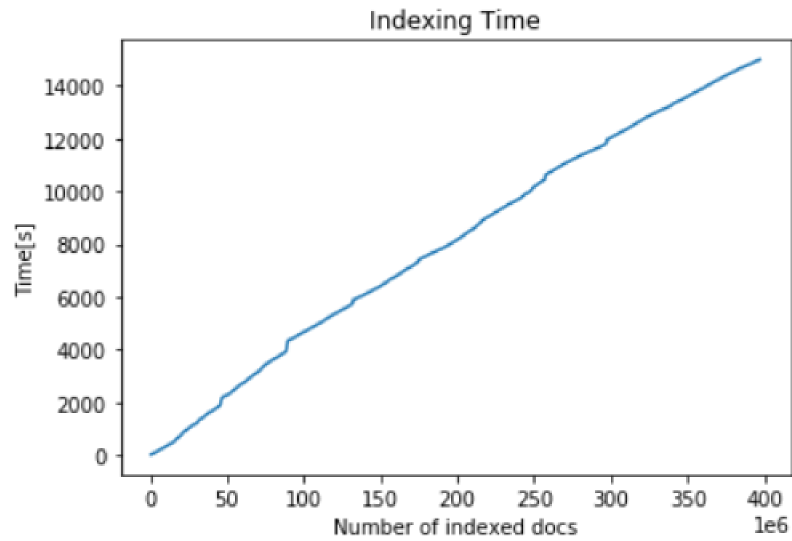

Figura 6.2: Tiempo de Indexación Wikidata

6.3. Búsqueda En El Índice

Para buscar en el índice creado se utilizó un subconjunto del dataset TREC 2009 Million Query Track [3] con 40 000 consultas en lenguaje natural.

Se estudió tanto el tiempo que necesitaba para recuperar los primeros 100 hits más relevantes, con los resultados en la Figura 6.3, y el tiempo para buscar y ordenar todos los documentos con puntaje mayor a cero, con los resultados en la Figura 6.4.

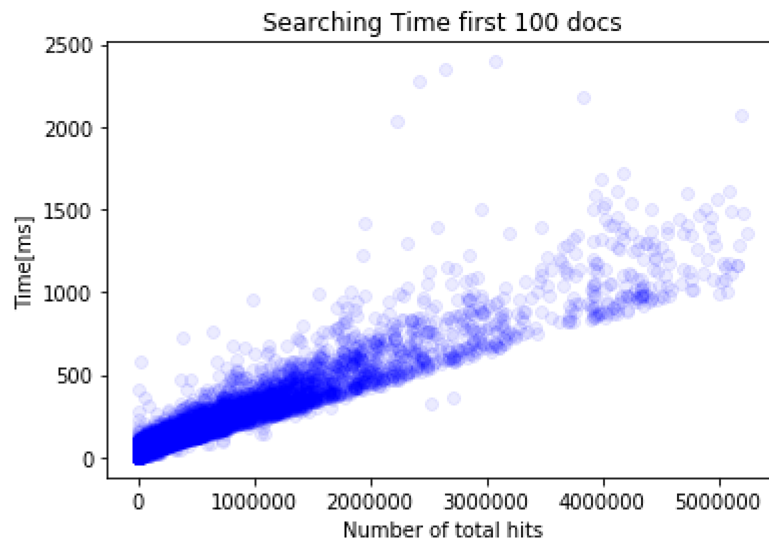

Figura 6.3: Tiempo de Búsqueda: 100 más relevantes

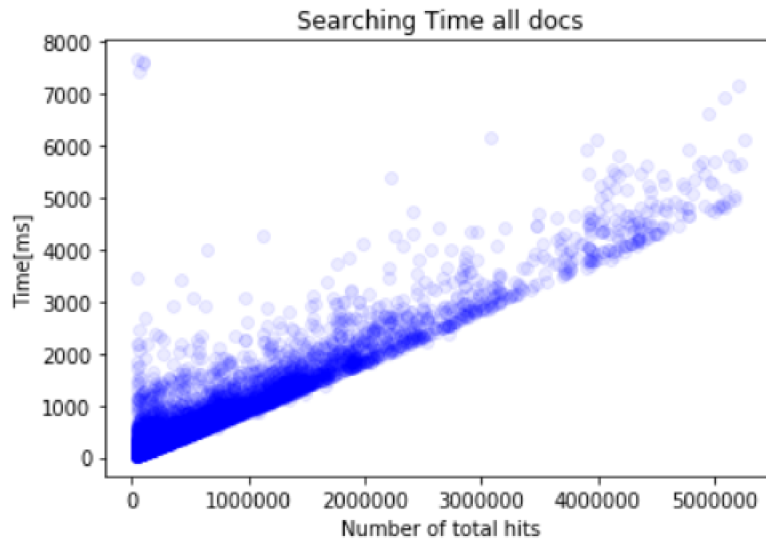

Figura 6.4: Tiempo de Búsqueda: Todos los documentos relevantes

6.4. Validación MillenniumDB

Luego, se indexó nuevamente *Wikidata* esta vez en MillenniumDB utilizando el método *bulkImport*, como se mencionó la indexación ocurre paralelamente a la creación de la base de datos con todos sus elementos (*nodes*, *edges*, *ids*,*literales* , etc).

Como se puede concluir de la Figura 6.5 y Figura 6.2 el *bottle neck* del proceso no se encontraría en la creación del índice invertido sino en otra parte de éste.

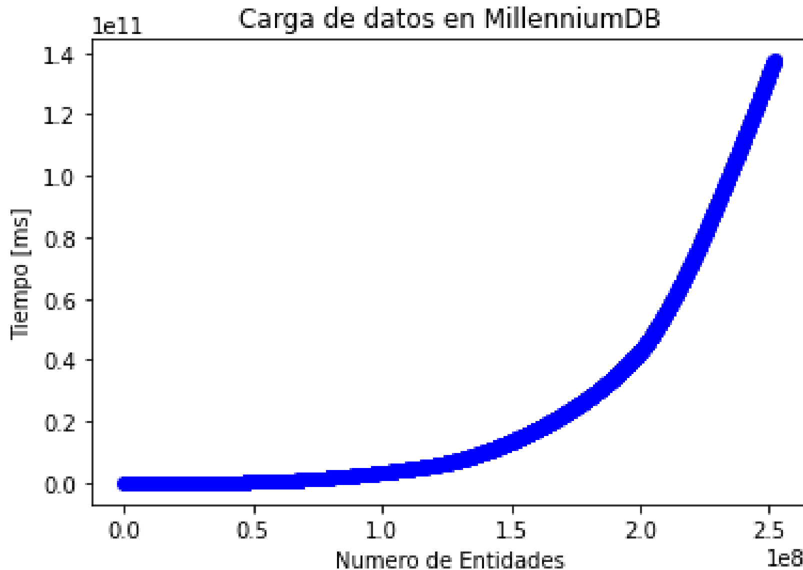

Figura 6.5: Tiempo de Creación en MillenniumDB con indexación

Capítulo 7

Conclusión

En esta sección se discute a grandes rasgos todo el trabajo realizado, los resultados experimentales y el trabajo futuro a realizar.

7.1. Trabajo Realizado y Objetivos

El objetivo de éste trabajo era implementar búsqueda de texto completo en el motor de base de datos MillenniumDB. Retomando, entonces los objetivos específicos definidos en el capitulo 1:

1. **Elegir una biblioteca de búsqueda que permita implementar búsqueda por índice invertidos**: Se determinó a través de experimentos de rendimiento una biblioteca y se llego a la conclusión que Lucene++ era el *framework* que mejor se adapta al problema.
2. **Diseñar un índice para los elementos del grafo para permitir búsquedas eficientes sobre ellos**: Tomando en cuenta los requisitos del equipo de MillenniumDB se diseñó un índice que fue implementado utilizando la biblioteca seleccionada. Cada documento de este índice recolecta la información necesaria para una búsqueda.
3. **Poder ordenar por relevancia los resultados de tales búsquedas**: Se logró tener tanto una forma de obtener los 100 documentos más relevantes dada una consulta, como obtener todos los documentos restantes, si así lo solicita el usuario.
4. **Integrar la funcionalidad desarrollada con el prototipo existente**: Se integró la funcionalidad de indexación y se implementó la interfaz definida para cada tipo de búsqueda en el lenguaje de consulta, pero no se modificó el lenguaje de consultas en sí basado en el consejo del equipo de MillenniumDB.
5. **Validar el rendimiento de la solución desarrollada (tiempo de respuesta, complejidad de espacio de los índices creados, etc)**: Se logró validar el índice tanto desacoplado de MillenniumDB como dentro de éste.

Tomando en cuenta que se completaron todos los objetivos excepto el cuarto que se encuentra inconcluso, el objetivo de este trabajo principal de este trabajo de titulo se completo casi en su totalidad. Esta falta se debió a la no modificación de la sintaxis del lenguaje

de consulta de MillenniumDB. La que requiere más consideración por parte del equipo de MillenniumDB.

7.2. Trabajo Futuro

Como se mencionó, quedó pendiente la modificación de la sintaxis del lenguaje de consultas de MilleniumDB para aceptar las keywords asociadas a búsqueda por índices invertidos, además de la modificación en la ejecución de tales consultas para que llamen a las clases implementadas. En el futuro, seria interesante, también, comparar la solución utilizada con base de datos de grafos populares como Neo4j.

Otra línea de trabajo puede ser relacionado con incorporar un análisis de enlaces en el grafo (como, por ejemplo, PageRank) para subir el puntaje de nodos con alta centralidad, e incorporar información del tipo de propiedad para dar más peso a una coincidencia en un campo más importante (por ejemplo, una coincidencia en el nombre de un nodo puede pesar más que una coincidencia en su biografía). El trabajo actual puede servir como la base para estos futuros avances.

Bibliografía

[1] Db-engines ranking. https://db-engines.com/en/ranking. Accessed: 2022-04-04.

[2] Neo4j documentation. https://neo4j.com/docs/cypher-manual/current/administration/indexes-for-full-text-search/. Accessed: 2020-05-08.

[3] Trec 2009 million query track. https://trec.nist.gov/data/million.query09.html. Accessed: 2022-04-11.

[4] Wikidata main page. https://www.wikidata.org/wiki/Wikidata:Main_Page. Accessed: 2022-03-15.

[5] Alan Wright. Luceneplusplus. https://github.com/luceneplusplus/LucenePlusPlus.

[6] R. Angles, H. Thakkar, and D. Tomaszuk. Mapping rdf databases to property graph databases. *IEEE Access*, 8:86091–86110, 2020.

[7] Renzo Angles. The property graph database model. In *AMW*, 2018.

[8] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)*, 50(5):1–40, 2017.

[9] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, 2017.

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

[11] Ricardo Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval - the concepts and technology behind search, Second edition*. Pearson Education Ltd., Harlow, England, 2011.

[12] Angela Bonifati, George H. L. Fletcher, Hannes Voigt, and Nikolay Yakovets. *Querying Graphs*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2018.

[13] Mike Buerli and CPSL Obispo. The current state of graph databases. *Department of*

Computer Science, Cal Poly San Luis Obispo, mbuerli@ calpoly. edu, 32(3):67–83, 2012.

[14] Ali Davoudian, Liu Chen, and Mengchi Liu. A survey on nosql stores. *ACM Comput. Surv.*, 51(2):40:1–40:43, 2018.

[15] Orri Erling. Virtuoso, a hybrid rdbms/graph column store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.

[16] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1433–1445, 2018.

[17] Daniel Hernández, Aidan Hogan, and Markus Krötzsch. Reifying RDF: what works well with wikidata? In Thorsten Liebig and Achille Fokoue, editors, *Proceedings of the 11th International Workshop on Scalable Semantic Web Knowledge Base Systems co-located with 14th International Semantic Web Conference (ISWC 2015), Bethlehem, PA, USA, October 11, 2015*, volume 1457 of *CEUR Workshop Proceedings*, pages 32–47. CEUR-WS.org, 2015.

[18] Florian Holzschuher and René Peinl. Performance of graph query languages: Comparison of cypher, gremlin and native access in neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, page 195–204, New York, NY, USA, 2013. Association for Computing Machinery.

[19] Shibamouli Lahiri. Complexity of Word Collocation Networks: A Preliminary Structural Analysis. In *Proceedings of the Student Research Workshop at the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 96–105, Gothenburg, Sweden, April 2014. Association for Computational Linguistics.

[20] Giulio Ermanno Pibiri and Rossano Venturini. Techniques for inverted index compression, 2019.

[21] Juan Ramos. Using tf-idf to determine word relevance in document queries. 01 2003.

[22] Marko A. Rodriguez and Peter Neubauer. Constructions from dots and lines, 2010.

[23] Denny Vrandecic and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, 2014.

[24] Denny Vrandečić and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.

[25] Domagoj Vrgoc, Carlos Rojas, Renzo Angles, Marcelo Arenas, Diego Arroyuelo, Carlos Buil Aranda, Aidan Hogan, Gonzalo Navarro, Cristian Riveros, and Juan Romero. Millenniumdb: A persistent, open-source, graph database. *CoRR*, abs/2111.01540, 2021.

[26] World Wide Web Consortium. Sparql 1.1 entailment regimes. 2013.

[27] World Wide Web Consortium. Sparql 1.1 query language. 2013.

[28] World Wide Web Consortium. Rdf 1.1 concepts and abstract syntax. 2014.

[29] World Wide Web Consortium. Rdf 1.1 semantics. 2014.

[30] World Wide Web Consortium. Rdf schema 1.1. 2014.

[31] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6–es, July 2006.