UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Proyecto de grado

Aceleración de consultas en BD de grafos mediante el uso de operaciones de álgebra lineal

Bruno Amaral Ribeiro
Juan Manuel San Martín

Ingeniería en Computación
Facultad de Ingeniería
Universidad de la República

Montevideo – Uruguay
Diciembre de 2019

UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Proyecto de grado

Aceleración de consultas en BD de grafos mediante el uso de operaciones de álgebra lineal

Bruno Amaral Ribeiro
Juan Manuel San Martín

Proyecto de grado presentada en la Facultad de Ingeniería de la Universidad de la República, como parte de los requisitos necesarios para la obtención del título de grado en Ingeniería en Computación.

Directores:
Lorena Etcheverry
Pablo Ezzatti

Montevideo – Uruguay
Diciembre de 2019

Juan Manuel San Martín, Bruno Amaral Ribeiro

Proyecto de grado / Bruno Amaral Ribeiro Juan Manuel San Martín. - Montevideo: Universidad de la República, Facultad de Ingeniería, 2019.

XIV, 81 p.: il.; 29, 7cm.

Directores:

Lorena Etcheverry

Pablo Ezzatti

Proyecto de Grado – Universidad de la República, Ingeniería en Computación, 2019.

Referencias bibliográficas: p. 55 – 81.

I. Etcheverry, Lorena, Ezzatti, Pablo, . II. Universidad de la República, Ingeniería en Computación. III. Título.

INTEGRANTES DEL TRIBUNAL DE DEFENSA DE PROYECTO

Fernando Carpani

Fernando Fernández

Federico Gómez

Montevideo – Uruguay
Diciembre de 2019

RESUMEN

Las denominadas bases de datos de grafos (BDG) permiten consultar y almacenar información estructurada en grafos. En los últimos años estos sistemas han ganado popularidad y madurez, permitiendo gestionar grafos de gran tamaño y con tiempos de cómputo cada vez menores. De todas formas, el procesamiento eficiente de consultas sobre grafos sigue siendo un tema relevante en la industria y la academia. Una línea promisoria en este aspecto es transformar las consultas sobre BDG en operaciones de álgebra lineal numérica (ALN) y, de esta forma, valerse del amplio conjunto de herramientas que ofrece dicho campo. Adicionalmente, el uso de operaciones de ALN posiciona a las GPUs (del inglés *Graphics Processing Units*), como una alternativa eficiente y de bajo costo para desarrollar tareas vinculadas al procesamiento y optimización de consultas sobre grafos, en particular SPARQL (*SPARQL Protocol And RDF Query Language*) el lenguaje de consulta estándar sobre grafos en RDF (*Resource Description Framework*).

Representar grafos relevantes (como los que modelan redes sociales) con matrices tiene la particularidad de que, en general, genera matrices con muchos coeficientes sin información. Mapearlas de forma directa entonces no es una buena opción, ya que se desperdicia mucha memoria en datos que no son relevantes y el cómputo de las operaciones implica trabajo innecesario. Esta situación motiva el uso de matrices dispersas.

En este trabajo se estudia la composición de operaciones de ALN que resuelven consultas utilizando una matriz para representar el grafo que modela la base de datos y se realiza un prototipo utilizando C++ y CUDA.

Se busca entonces la mejor forma de cargar y operar con las matrices, para esto se compararon distintos formatos de almacenamiento resultando el de mejor performance respecto a nuestros intereses el denominado CSR (*Compressed Storage Row*).

Para realizar las pruebas se utilizan conjuntos de datos generados de *Berlin SPARQL Benchmark*. Estos datos deben ser transformados al formato de entrada que utiliza el prototipo, para esto se implementa un traductor utilizando Java. Luego de realizado el prototipo, se ejecutan pruebas en él y en otras implementaciones de motores de bases de datos RDF los resultados fueron comparados mostrando que existen ventajas al utilizar la GPU para consultas cuyo grafo resultado es grande.

Palabra clave:
Bases de datos de Grafos, Álgebra lineal numérica, *Graphics Processing Unit*

Tabla de contenidos

Capítulo 1

Introducción

El término *Big Data* se usa tanto para caracterizar datos según su volumen, velocidad de creación, variedad, etc. como para hablar de posibilidades de análisis y extracción de conocimiento a partir de datos con estas características. En este marco los grafos son una abstracción muy utilizada, en particular para representar ciertos escenarios donde se quiere modelar las relaciones entre entidades o conceptos. Un caso paradigmático de esto son los datos provenientes de las redes sociales.

Los llamados sistemas de bases de datos de grafos (BDG) permiten almacenar y consultar grafos. En los últimos años estos sistemas han ganado madurez, permitiendo gestionar grafos de gran tamaño. De todas formas, el procesamiento eficiente de consultas sobre grafos se ha transformado en un tema relevante en la industria y la academia.

Una línea promisoria combinando estos puntos anteriores es transformar las consultas sobre BDG en operaciones de álgebra lineal numérica (ALN) y, de esta forma, valerse del amplio conjunto de herramientas que ofrece dicho campo.

Existen en el mercado diversos motores de BDG. Algunos de ellos almacenan los grafos usando motores relacionales, y otros implementan mecanismos nativos para el almacenamiento y procesamiento de las consultas. Pocos de éstos explotan la relación natural que existe entre los grafos y su representación mediante matrices (dispersas), y la consecuente relación entre las operaciones

de consulta sobre grafos y operaciones algebraicas sobre las matrices que los representan.

El campo de ALN ha concentrado en las últimas décadas importantes esfuerzos para la inclusión de técnicas de HPC (del inglés *High Performance Computing*) en la resolución de las operaciones básicas del álgebra matricial. Esta situación, aunado con lo expresado en el párrafo anterior, permite vislumbrar importantes aceleraciones en el manejo de BDG en el caso de poder definir las principales consultas de estas bases en términos de operaciones de ALN.

Coincidentemente, la necesidad de mejorar la performance de distintas aplicaciones, y en especial operaciones de ALN, han llevado a explorar nuevas arquitecturas de hardware que exploten, entre otros aspectos, la naturaleza paralelizable de los métodos de resolución. En este contexto las GPUs aparecen como una alternativa de bajo costo para desarrollar algunas de las tareas que implican grandes volúmenes de datos.

1.1. Objetivos

Las bases de datos de grafos son una herramienta que está ganando popularidad en los últimos años debido a la abundancia de datos que pueden ser modelados de esta forma. Por otro lado el uso de operaciones de ALN posiciona a las técnicas de HPC en general y a los aceleradores gráficos (GPUs) en particular, como una alternativa eficiente y de bajo costo para desarrollar algunas de las tareas vinculadas al procesamiento y optimización de consultas.
A partir de lo expresado anteriormente, el objetivo principal del proyecto es lograr implementar de forma eficiente un conjunto de consultas sobre BDG en términos de operaciones de ALN y, en este contexto, evaluar sacar partido del poder de cómputo que ofrecen las GPUs modernas para el procesamiento de este tipo de operaciones.

Para la consecución de los objetivos principales del proyecto se establecen los siguientes objetivos específicos intermedios:

- Revisión del estado del arte de BD de grafos, con especial atención en

algunos de los motores de almacenamiento para RDF con soporte para consultas en SPARQL.

- Investigación sobre operaciones de ALN relacionadas con las consultas de BD de grafos y los distintos esfuerzos tendientes a formalizar dicha relación.
- Crear una definición de estructuras de datos asociados a los grafos y las consultas sobre ellas que permitan ser utilizadas por operaciones de GPUs.
- Identificación de un sub-conjunto de los operadores más utilizados de SPARQL para ser definidos en términos de operaciones de ALN.
- Implementar en forma eficiente las operaciones de ALN que permitan resolver las consultas de SPARQL antes identificadas.
- Finalmente, evaluar el uso de las GPU para acelerar estas consultas con respecto a sus implementaciones en CPU.

1.2. Resultados esperados

- Documento del estado del arte de BD de grafos, con especial atención en los motores de almacenamiento para RDF con soporte para consultas en SPARQL.
- Documento de actualización del estado del arte sobre operaciones de ALN relacionadas con las consultas de BD de grafos y los distintos esfuerzos tendientes a formalizar dicha relación.
- Implementar, al menos, un prototipo, que incluyen un conjunto de operaciones que permitan acelerar las consultas en SPARQL, que saque partido del poder de cómputo de dispositivos con arquitectura masivamente paralela.

1.3. Estructura del documento

El resto del documento se estructura de la siguiente forma: el Capítulo 2 presenta conceptos preliminares, así como un análisis sobre motores de almacenamiento y consulta existentes para RDF con soporte a SPARQL además de investigación sobre otros esfuerzos por utilizar operaciones de álgebra para resolver consultas SPARQL.

Luego, en el Capítulo 3, se presenta por un lado un análisis de los datos y del uso de algunas bases de datos de grafos. Se analizan también propuestas de representación lógica y física de las estructuras necesarias y por otro, la implementación de un subconjunto de consultas SPARQL relevantes en términos de operaciones de ALN sobre la representación matricial elegida.
Seguidamente, en el Capítulo 4, se plantean los casos de prueba utilizados en la evaluación experimental, así como la forma en que fueron generados estos datos, para culminar este capítulo se exponen los resultados obtenidos de ejecutar estas pruebas en la implementación realizada y en herramientas disponibles en el mercado.
Finalmente en el Capítulo 5 se resumen las conclusiones arribadas en el proyecto y posibles extensiones del trabajo realizado.

Capítulo 2

Conceptos preliminares

A continuación se presenta una breve introducción a conceptos preliminares, así como una breve revisión de los motores de almacenamiento y consultas que los soportan, conocidos como RDF *Triplestores*.

2.1. *Resource Description Framework*

Resource Description Framework (RDF) [1] es un modelo de datos para expresar afirmaciones sobre recursos identificados por un identificador universal (URI del inglés *Universal Resource Identifier*).

Las afirmaciones son expresadas como triplas sujeto - predicado - objeto, donde el sujeto y el predicado siempre es un recurso y el objeto puede ser un recurso o un string. Los nodos blancos son utilizados para representar recursos anónimos o recursos sin URI, que típicamente tienen una función estructural como por ejemplo agrupar un conjunto de afirmaciones. Los valores en RDF son llamados literales, y sólo puede aparecer como objeto en las triplas.

Un conjunto de triplas RDF puede verse como un grafo dirigido donde los sujetos y objetos son nodos, y los predicados son arcos. Existen varios formatos para la serialización de RDF, en este trabajo utilizamos Turtle [2].

2.2. SPARQL *Protocol and RDF Query Language*

SPARQL (SPARQL *Protocol and RDF Query Language* [3]), es el lenguaje de consulta estándar y el protocolo para *Linked Open Data* en la web o en una base de datos de grafos (también llamada RDF *triplestore*). SPARQL, permite a los usuarios consultar información de bases de datos o cualquier fuente de datos que se pueda mapear a RDF. El estándar SPARQL está diseñado y respaldado por el W3C y ayuda a los usuarios y desarrolladores a centrarse en lo que les gustaría saber en lugar de cómo está organizada la base de datos.

La mayoría de las formas de consulta en SPARQL contienen un conjunto de patrones de tripleta (*triple patterns*) denominadas patrón de grafo básico. Los patrones de tripleta son similares a las tripletas RDF, excepto que cada sujeto, predicado y objeto puede ser una variable. Un patrón de grafo básico concuerda con un subgrafo de datos RDF cuando los términos RDF (*RDF terms*) de dicho subgrafo pueden ser sustituidos por las variables y el resultado es un grafo RDF equivalente al subgrafo en cuestión.

La Figura 2.1 muestra una consulta SPARQL para encontrar el título de un libro en el grafo de datos dado en la parte superior de la Figura. La consulta consta de dos partes: la cláusula SELECT identifica las variables que aparecen en los resultados de la consulta, y la cláusula WHERE proporciona el patrón de grafo básico para la concordancia con el gráfo de datos. El patrón de grafo básico de este ejemplo consiste en un único patrón de tripleta con una sola variable (?title) en la posición del objeto [3].

```
<http://ex.org/b1> <http://purl.org/dc/1.1/title> "SPARQL" .


SELECT ?title
WHERE
{
  <http://ex.org/b1> <http://purl.org/dc/1.1/title> ?title .
}
```

Figura 2.1: Ejemplo de consulta de SPARQL.

La consulta de la Figura 2.1, efectuada sobre los datos indicados anteriormente, tiene la siguiente salida:

```
title
"SPARQL Tutorial"
```

2.3. Bases de datos de grafos

En el área de la computación, una base de datos orientada a grafos es aquella que permite almacenar la información como nodos de un grafo y sus respectivas relaciones con otros nodos, permitiendo así aplicar la teoría de grafos para recorrer la base de datos.

El uso de este tipo de bases de datos depende fuertemente de la lógica de negocio donde se encuentre involucrada la información a almacenar, ya que no puede aplicar en todos los escenarios, o tal vez no se podría aprovechar su potencial en unos u otros contextos.

En contraste, en las bases de datos relacionales el vínculo entre dos datos, muchas veces está dado por claves foráneas [4], esta relación es una convención del usuario, es decir, no está formalmente definida sin otra fuente más que la estructura de la base de datos. Además, generalmente, tiene dependencias de problemas de la estructura de datos inherente a una base de datos relacional, como por ejemplo cadenas de registros físicas, o la forma en la que estos datos son persistidos, teniendo como consecuencia que una misma consulta en una base de datos relacional puede tener asociado un tiempo de ejecución más elevado que una en una base de datos de grafos.

La forma en que las bases de datos de grafos organizan la información puede variar. Una forma es persistir una lista, con todos los nombres de los nodos, asociados a un identificador y, por otra parte, tener una tabla, donde se almacenen las relaciones entre los nodos [5]. Otra manera es usar una estructura clave-valor, haciendo que este tipo se parezca más a los usados en bases NoSQL [6]. Algunas propuestas utilizan además conceptos extras, como tags o propiedades, los cuales pueden resumirse a un tipo especial de relación como las marcadas anteriormente, pero que permiten realizar filtros de datos más

rápidamente [7].

Para poder realizar consultas sobre estas bases, se requiere un lenguaje de consultas distinto a SQL, ya que este está fuertemente vinculado a bases de datos relacionales [8]. Sin embargo, hasta 2008 no hubo un estándar o lenguaje [9].

2.4. *General-Purpose Computing on Graphics Processing Units*

GPGPU (*General-Purpose Computing on Graphics Processing Units* [10]) es un concepto reciente dentro de informática que trata de estudiar y aprovechar las capacidades de cómputo de las GPUs para resolver problemas de propósito general.

Una GPU es un procesador diseñado para los cómputos implicados en la generación de gráficos 3D interactivos. Algunas de sus características (bajo precio en relación a su potencia de cálculo, gran paralelismo, optimización para cálculos en coma flotante), se consideran atractivas para su uso en aplicaciones fuera de los gráficos por computadora, especialmente en el ámbito científico y de simulación. Así, se han desarrollado técnicas para la implementación de simulaciones de fluidos, bases de datos, algoritmos de clustering, etc.

Debido a las diferencias fundamentales entre las arquitecturas de la GPU y la CPU, no cualquier problema se puede beneficiar de una implementación en una GPU. En concreto, el alto grado de paralelismo necesario para alcanzar desempeños razonables plantea las mayores dificultades. Las CPU están diseñadas para el acceso aleatorio a memoria. Esto favorece la creación de estructuras de datos complejas, con punteros a posiciones arbitrarias en memoria. Por otro lado, el acceso irregular a memoria en general deteriora enormemente el desempeño de las GPUs.

La tarea del diseñador de algoritmos para GPUs consiste principalmente en adaptar los accesos a memoria y las estructuras de datos a las características de la GPU. Pese a que cualquier algoritmo que sea implementable en una

CPU lo es también en una GPU, esas implementaciones no serán igual de eficientes en las dos arquitecturas. En concreto, los algoritmos con un alto grado de paralelismo, sin necesidad de estructuras de datos complejas y con una alta intensidad aritmética son los que mayores beneficios obtienen de su implementación en GPU.

2.5. Manejadores de datos en RDF

Existen dos enfoques típicos en el diseño de los sistemas manejadores de datos en RDF, también conocidos como *Triplestores*. Por un lado, los enfoques basados en el modelo relacional que representan los datos RDF en forma tabular siguiendo diferentes técnicas, y hacen un uso extensivo de índices sobre las tablas para resolver las consultas. En algunos casos, las consultas SPARQL son traducidas a consultas SQL sobre esta representación. Por otro lado, existen enfoques basados en grafos, que buscan representar a los datos RDF como un grafo, y usualmente emplean técnicas basadas en homomorfismos para resolver las consultas SPARQL. Una revisión exhaustiva del estado del arte en estos sistemas puede encontrarse en [11].
En esta sección se presentan las características principales de dos herramientas actuales que corresponden al enfoque relacional Virtuoso (utilizada comercialmente) y RDF-3X (*open source*), y una que corresponde al enfoque orientado a grafos llamada gStore (*open source*).

2.5.1. Virtuoso

Virtuoso es un *middleware* y motor de base de datos híbrido que combina la funcionalidad de un sistema de gestión de bases de datos relacionales (RDBMS), base de datos relacional de objetos (ORDBMS), base de datos virtual, RDF, XML, texto libre, servidor de aplicaciones web y servidor de archivos funcionalidad en un solo sistema. En lugar de tener servidores dedicados para cada una de las funcionalidades antes mencionados, Virtuoso es un "servidor universal". La edición gratuita y de código abierto de Virtuoso *Universal Server* también se conoce como OpenLink Virtuoso. El software ha sido desarrollado por OpenLink Software con Kingsley Uyi Idehen y Orri Erling como principales arquitectos de software [12].

Virtuoso se desarrolló por primera vez como un RDBMS orientado a transacciones de filas utilizando clusters de servidores. Fue sucesivamente reorientado como una base de grafos RDF con SPARQL incorporado e inferencia. Por último, el producto ha sido revisado para aprovechar el almacenamiento comprimido en columnas y la ejecución vectorizada. En el artículo de Erling (2012) [13] se discuten las opciones de diseño tomadas para:

- Aplicar técnicas de almacenamiento por columnas bajo los requisitos de performance necesario.
- Manipular los datos RDF semi-estructurados e impredecibles.
- Soportar las cargas de trabajo de BI (Business Intelligence) más típicas.

Las aplicaciones de Virtuoso más extendidas están en el espacio RDF, con terabytes de tripletas RDF que usualmente no caben en la RAM. La excelente eficiencia espacial de la compresión en el almacenamiento por columnas fue el mayor incentivo para la transición hacia este tipo de almacenamiento. Además, la combinación de un modelo de datos sin esquemas con buen rendimiento analítico es atractivo para la integración de datos donde el esquema de datos es volátil.

2.5.2. RDF-3X

En el informe de Neumann y WeikumThe (2009) [14] se presenta el motor RDF-3X, una implementación de SPARQL que afirman lograr excelente rendimiento, persiguiendo una arquitectura de estilo RISC con una indexación simplificada y mejoras en el procesamiento de consultas.

RDF-3X sigue la lógica de los sistemas de gestión de datos diseñados y personalizados para los dominios de aplicación específicos pueden superar a los sistemas genéricos por dos órdenes de magnitud.

Algunos de los factores que influyen en este argumento son:

- Opciones adaptadas de estructuras de datos y algoritmos en lugar de una amplia variedad de métodos.
- Software más ligero sin sobrecarga.
- Simplificación de la configuración y autoadaptación más fácil a ambientes cambiantes.

Se basa en los siguientes tres principios claves:

- El diseño físico es independiente de la carga de trabajo. Esto, afirman, lo logran construyendo índices sobre las seis permutaciones de las tres dimensiones que constituyen una tripleta RDF y, adicionalmente, los índices sobre las variables *count-aggregated* para las tres agrupaciones bidimensionales y las tres proyecciones unidimensionales. Cada uno de estos índices se pueden comprimir, el almacenamiento total para todos los índices juntos afirman es menor que el tamaño de los datos primarios.
- El procesador de la consulta es de estilo RISC basado principalmente en hacer merge joins sobre listas de índices ordenas. Esto dicen es posible por la indexación "exhaustiva"de la tabla de tripletas. También afirman que de hecho, todo el procesamiento es sólo de índices y la tabla de tripletas es meramente virtual. Los árboles de operadores los construyen preservando órdenes interesantes para subsiguientes joins de la mayor extensión posible, sólo cuando esto ya no es posible, RDF-3X cambia a procesamiento de combinación basado en hash. Expresan también que este enfoque puede ser altamente optimizado a nivel de código, y tiene una sobrecarga mucho más baja que los procesadores de consulta tradicionales. Al mismo tiempo, es suficientemente versátil para apoyar también las diversas opciones de eliminación de duplicados de SPARQL, patrones disyuntivos en las consultas y todas los demás características que SPARQL requiere.
- El optimizador de consultas lo centran principalmente en el orden de los joins en su generación de planes de ejecución. Emplean programación dinámica para la enumeración del plan, con un modelo de costos basado en estadísticas específicas de RDF. Estas estadísticas incluyen contadores de predicado-secuencias frecuentes en trayectorias del grafo de datos, estos caminos son potenciales patrones. Declaran que en comparación con el optimizador de consultas en un sistema universal, el optimizador RDF-3X es más sencillo pero mucho más preciso en sus estimaciones de selectividad y decisiones sobre planes de ejecución.

Aunque es razonable suponer que la mayoría de las bases de datos RDF son de consulta mayormente, es decir son principalmente de lectura, tiene que soportar al menos carga incremental e idealmente incluso actualizaciones en línea como la inserción de una nueva tripleta para anotar los datos existentes. Ale-

gan que el reto de hacer esto eficientemente implica lidiar con la indexación agresiva que RDF-3X emplea para consultas rápidas. Por eso las actualizaciones las recopilan en espacios de trabajo e índices diferentes y las fusionan posteriormente con los índices principales de la base de datos mediante procesos batch.

Intentan que esto sea transparente a los programas, sostienen que el procesador de consultas se puede extender para proporcionar esta comodidad con bajo overhead. Mientras este enfoque no proporciona transacciones ACID completas, incluye formas de control de la concurrencia y soporta el nivel read-committed [14].

2.5.3. GStore

GStore [15] propone un enfoque basado en gráfos para almacenar y consultar datos RDF. En lugar de mapear tripletas de RDF en una base de datos relacional como la mayoría de los métodos existentes, almacena los datos RDF como un gran grafo. Entonces una consulta SPARQL se convierte en una consulta a un subgrafo.

Con el fin de acelerar el procesamiento de consultas, desarrollaron un nuevo índice, junto con algunas reglas de poda y algoritmos de búsqueda que buscan aumentar la eficiencia. Ese método afirman puede responder consultas exactas de SPARQL y consultas con "comodines" de una manera uniforme. También proponen un algoritmo de mantenimiento para manejar actualizaciones en línea sobre repositorios RDF. Específicamente, modela un conjunto de datos RDF como un grafo dirigido con múltiples aristas, donde cada vértice corresponde a un sujeto o un objeto. Cada tripleta representa una arista dirigida de un sujeto a su objeto correspondiente. Dado un sujeto y un objeto, puede existir más de una propiedad entre ellos, es decir, múltiples aristas pueden existir entre dos vértices.

Proponen también un nuevo esquema de indexación que intenta acelerar el procesamiento de consultas. En primer lugar, almacenan un grafo RDF como una tabla de lista de adyacencias basadas en almacenamiento. Luego, para cada entidad o vértice de clase en el grafo, de acuerdo con las etiquetas de sus

adyacencias y las etiquetas de vértices vecinos asignan una cadena de bits como la firma del vértice. De esta manera, un grafo RDF se convierte en un grafo de firma de datos G*. Entonces, proponen un nuevo índice (llamado VS*-tree) sobre G*.

En tiempo de ejecución, también codifican todos los vértices de Q en firmas de vértices, y luego convierten Q en su correspondiente grafo de firma de consulta Q*. Encontrar todas las coincidencias de Q* sobre G* afirman conducirá a todos los resultados candidatos sin ningún falso negativo [15].

En resumen, dicho trabajo intenta realizar las siguientes contribuciones:

- Adoptar el modelo de grafo como el esquema de almacenamiento físico para datos RDF. Específicamente, almacenan datos RDF en listas de adyacencia disk-based.
- Transformar un grafo RDF en un grafo de firma de datos mediante la codificación de cada entidad y vértice de clase. Propone un nuevo índice (VS*-tree) sobre el grafo de firma de datos.
- Desarrollar una regla de filtrado para la consulta de subgrafos sobre el grafo de firma de datos, que puede incrustarse en el algoritmo de consulta que intenta responder a consultas SPARQL exactas y consultas con comodines de una manera uniforme.
- Demostrar a través de experimentos que el rendimiento de este método es superior a los métodos existentes y soporta updates online con bajo overhead.

2.6. Trabajo relacionado

A continuación se presenta TensorRDF [16] un estudio cuyo objetivo fue crear una representación en tensores de consultas SPARQL para poder operar sobre RDF de manera análoga a operaciones de álgebra lineal.

El autor indica el interés en mejorar la performance de consultas que requieren muchos recursos computacionales. Se enuncia que apróximadamente el 48 % de las consultas analizadas no eran sencillas y llevaban asociadas un alto nivel de procesamiento. Estas consultas incluyen los operadores UNION y OPTIONAL

y estos son llamados patrones no conjuntivos dentro de la consulta (los patrones conjuntivos, como lo indica la palabra, hace referencia a operadores AND y FILTER).

El trabajo presenta las siguientes definiciones.

RDF: Toda la información RDF es almacenada en tripletas, donde se tienen 3 elementos, sujeto, objeto y predicado, que pueden ser modeladas como 2 nodos unidos por una arista con valor.

SPARQL Query: Puede ser visto como una tupla (qt, RC, DD, Gp, SM) donde qt es el tipo de consulta (SELECT, ASK, etc.); RC es la cláusula resultado, es decir indica que información del resultado es presentada. DD es el dataset definition, el cual es opcional y es la fuente seleccionada para utilizar en la consulta, SM es *solution modifier* e indica si es necesario realizar operaciones sobre los resultados, como por ejemplo ordenar los resultados u operaciones similares.
Por último se tiene Gp, el cual es el patrón del grafo que se busca, a su vez, este es una tupla (T, f, OPT, U), donde T son tripletas, iguales a las definidas en la sección RDF, pero donde cualquiera de los elementos de la tripleta puede ser una variable, se dice además que presentan una intersección si una misma variable ocurre en más de una tripleta. f es un conjunto de FILTERS con condiciones booleanas, que se pueden aplicar sobre los resultados de la consulta. OPT es un conjunto de OPTIONAL, patrones opcionales, los cuales son buscados en las posibles soluciones, y por último U es un conjunto de patrones UNION modelados como un grafo.
La respuesta de esta consulta, es una lista L de resultados, los cuales dependen de qt (el tipo de consulta).
Modelo general: Se define un conjunto $\mathcal{E}$, con $\mathcal{E}$ finito. Una propiedad, en el modelo, de un elemento $e \in \mathcal{E}$, es una función $\pi : \mathcal{E} \to \Pi$, donde Π representa un codominio apropiado. Entonces $\pi(e) := \langle \pi, e \rangle$ donde $e \in \mathcal{E}$. La familia de propiedades, π_i, para $i = 1, ..., n < \infty$ y sus correspondientes conjuntos codominios Π. Con estas definiciones se puede modelar el siguiente producto vectorial $\mathcal{E} \times \Pi_1 \times ... \times \Pi_n$.
Representación en tensores: Dado un grafo RDF (G) definió el conjunto S como el conjunto de todos los recursos (nodos) en G, donde S es finito.

Una propiedad de un recurso $s \in S$ se define como $\pi : S \to \Pi$, donde Π es un codominio apropiado. Se define $\pi(s) := \langle \pi, s \rangle$, por lo tanto se puede imaginar que π asocia un recurso s con su valor de propiedad dado por el grafo RDF. Se puede ver la representación de G como el producto de conjuntos $S \times \Pi_1 \times ... \times \Pi_{k-1} \times \Pi_k \times ... \times \Pi_{k+d}$. Separando de esta manera y llamando B al producto desde S hasta $k-1$, y H desde k a $k+d$, entonces se separan los índices en dos. Al primer conjunto se le llama B, por *Body* y al segundo H, por *Head*. Falta ver la representación, la cual es una forma multilineal $\gamma : B \to H$ Por lo tanto, se puede representar un grafo RDF como un tensor de rango k con valores en H. Observando los codominios y siendo estos el conjunto que tiene a los sujetos y objetos y el otro como el que contiene los predicados. Ahora, como S, O y P son finitos, se puede definir una función $idx : S \to N$ la cual lleva estos conceptos a números enteros.

No se ahondará en como funciona la implementación, pero se basa en operar con tensores utilizando una equivalencia entre estos y consultas de SPARQL.

El trabajo presenta resultados de la evaluación experimental, ejecutando en un ambiente de cuatro computadoras de similares características (para aprovechar las técnicas de des-centralización) y comparó los resultados contra una única de esas computadoras corriendo su algoritmo. Como bases de datos tomó DBPedia, UNIPROT y BILLION. Los resultados que obtuvo fueron excelentes para su solución, dejando un amplio margen con la competencia, en particular con RDF-3X, que, según el autor, es la más óptima dentro de las otras soluciones.

Capítulo 3

Propuesta

En esta sección se estudia cual debería ser el objetivo y rango del prototipo a implementar. Para ello se analizan qué consultas son las más usuales en SPARQL, como también qué representaciones posibles existen para matrices a nivel de estructura de datos (esto permitirá elegir una representación que haga buen uso de los recursos de las GPUs). Recién luego de haber realizado estos análisis se podrá definir un prototipo a construir y compararlo con otras soluciones ya existentes.

3.1. Representación lógica

Como se expresó antes existe una relación entre los grafos y su representación mediante matrices la cual será utilizada en esta propuesta. A continuación se detallará esta relación y como utilizando ALN, se puede realizar consultas a una base de datos de grafos.

El grafo de la Figura 3.1 será utilizado para explicar las representaciones analizadas.

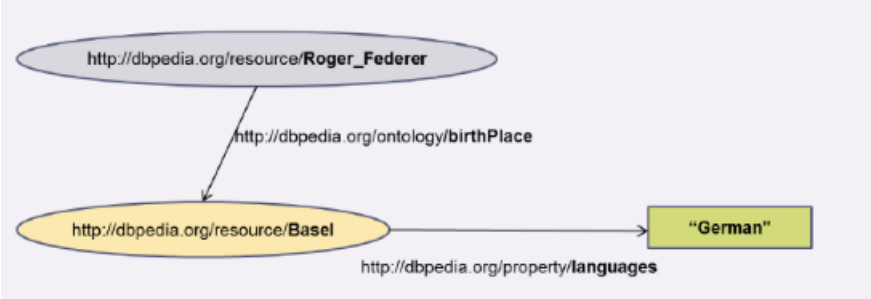

Figura 3.1: Grafo RDF de ejemplo

La representación lógica tradicional [17] consiste de una matriz de $M \times N$ siendo M la cantidad de Sujetos y N la cantidad de Objetos, donde cada entrada es una lista de los predicados que se cumplen. Se asigna a cada sujeto y objeto un identificador que se corresponderá a una columna de la matriz, una posible asignación para el ejemplo podría ser $RogerFederer = 1$, $Basel = 2$ y $German = 3$ (llamaremos a este mapeo hashObj). También se le asigna un identificador a cada predicado, por ejemplo: $BirthPlace = 1$ y $Languages = 2$. La representación lógica para este caso sería la presentada en la Tabla 3.1

	RogerFederer	Basel	German
RogerFederer		1	
Basel			2

Tabla 3.1: Ejemplo de representación lógica.

Obtener el lugar de nacimiento de RogerFederer equivale a la siguiente consulta en SPARQL:

Consulta 3.1 Consulta SPARQL (obtener lugar de nacimiento de sujeto)

```
1 PREFIX dbr:<http://dbpedia.org/resource/>
2 PREFIX dbo:<http://dbpedia.org/ontology/>
3
4 SELECT ?place
5 WHERE
6 {
7   dbr:RogerFederer dbo:birthPlace ?place.
8 }
```

Considerando la representación anterior, obtener el lugar de nacimiento del sujeto equivale a realizar las siguientes operaciones:

- $v \times A$
  siendo $v = (1, 0)$ y A la matriz ya descripta. Esta es una manera sencilla de obtener una fila de la matriz original utilizando operaciones matriciales. Una vez que se multiplica un vector con un 1 en el índice de la fila deseada y 0 en caso contrario se obtiene la fila del índice seleccionado. Es decir, se obtiene la fila que contiene todas las triplas que tienen el sujeto RogerFederer.
- El resultado de la operación anterior será $w = (\_, 1, \_)$ dicho resultado tendrá que ser recorrido entrada por entrada buscando que dentro de cada lista se encuentre la propiedad buscada.

Consulta 3.2 Pseudocódigo del algoritmo de ejemplo

```
funcionBusqueda(predicadoBuscado)
    for i = 1 to M
        while w[i].HasNext;
            predicado = w[i].next;
            if (predicado == predicadoBuscado) then
            result.add(i)
        fin while
    fin for
```

El resultado será 2, buscando este índice en el hashObj se puede obtener que la ciudad de nacimiento de RogerFederer es Basel.

Durante el proceso de idear una solución se propuso también una representación no tradicional en donde se tiene un Matriz multidimensional (tensor) donde cada nivel está asociado a un predicado, al igual que en la representación tradicional se le asigna a cada sujeto y objeto un identificador que se corresponderá a una columna de las matrices y también se asigna un identificador a cada predicado pero este definirá un nivel de la matriz tridimensional y la existencia de una arista entre un sujeto y objeto sera dada por el valor 1 en el nivel correspondiente. Esto se vio como una potencial optimización, ya que cada celda contendría estrictamente un 1 o un 0 en lugar de una lista de índices. La representación para este caso sería la resumida en las Tablas 2 y 3

	RogerFederer	Basel	German
RogerFederer	0	1	0
Basel	0	0	0

Tabla 2: Nivel BirthPlace.

	RogerFederer	Basel	German
RogerFederer	0	0	0
Basel	0	0	1

Tabla 3: Nivel Languages.

Con esta representación la consulta ya estudiada (Consulta 3.1) se corresponde a realizar la siguiente operación:

$(v \times M_0) \times hashObj$

Siendo $v = (1, 0)$ y M_0(matriz bidimensional) la matriz del nivel BirthPlace, esta última matriz se puede obtener de multiplicar la matriz M (tridimensional) por una matriz de igual dimensión con todas las entradas en 0, salvo el nivel correspondiente a BirthPlace en donde todas las entradas serán 1; para luego aplanar el resultado sumando la misma entrada para cada nivel llamado a esta operación $ML(X)$ siendo X el nivel a obtener. Siguiendo con el ejemplo se tiene entonces que:

$(v \times M_0) \times hashObj = ((1, 0) \times ((0, 1, 0), (0, 0, 0))) \times hashObj$
$(v \times M_0) \times hashObj = (0, 1, 0) \times (RogerFederer, Basel, German)$
$(v \times M_0) \times hashObj = Basel$

Para comparar las dos representación antes descriptas se plantearan nuevas consultas a resolver y la implementación de las consultas en términos de operaciones de álgebra lineal sobre ambas.

Consulta 1

```
SELECT ?lenguaje
WHERE
```

```
{
    RogerFederer BirthPlace ?ciudad. ?ciudad lenguaje ?lenguaje
}
```

Los pasos en la representación tradicional son:

- $v \times A$ siendo $v = (1, 0)$ y A la matriz ya descripta. Esta operación ya fue analizada y el resultado es 2.
- Luego se realiza $w \times A$
  siendo $w = (0, 1)$ (porque el resultado de la operación anterior fue 2, si hubiese sido 1, w seria (1,0)). El resultado de la operación anterior será $z = (\_, \_, 2)$ igual que antes. Dicho resultado tendrá que ser recorrido entrada por entrada buscando que dentro de cada lista se encuentre la propiedad buscada.

  El pseudocódigo sería similar al visto en la Consulta 3.2, usando $predicadoBuscado = 2$
  Que resultará en $3 = German$.

En la representación n-dimensional:

siendo $v = (1, 0)$
$((v \times M_0) \times M_1) \times hashObj = (((1, 0) \times M_0) \times M_1) \times hashObj$
$((v \times M_0) \times M_1) \times hashObj = ((0, 1) \times M_1)) \times hashObj$
$((v \times M_0) \times M_1) \times hashObj = ((0, 0, 1)) \times hashObj$
$((v \times M_0) \times M_1) \times hashObj = German$

Consulta 2

```
SELECT ?p WHERE { RogerFederer ?p Basel}
```

En enfoque tradicional:
$(v \times A) \times w$ con $v = (1, 0)$ y $w = (0, 1, 0)$

En enfoque propuesto:

Equivale a iterar por nivel realizando $(v \times M_i) \times w$ con $v = (1, 0)$, $w = (0, 1, 0)$

Observaciones

En esta última operación es que el enfoque tradicional tiene una gran ventaja ya que la cantidad de operaciones necesarias es mucho menor cuando se deja libre el predicado (?p). En el enfoque matricial se tienen que hacer dos multiplicaciones y recorrer una lista que suele tener pocos elementos (intuitivamente se deduce esto dado que un sujeto suele estar conectado por una o ninguna propiedad) en cambio en el enfoque multidimensional se tienen que realizar dos multiplicaciones por nivel, en este ejemplo son solo dos niveles pero, en un caso real se tendría que iterar por tantos niveles como predicados lo cual sería más costoso que el enfoque tradicional donde la cantidad de operaciones no cambia sin importar cuantas propiedades existan en el sistema.

En resumen, la diferencia entre ambos enfoques es que para verificar una relación en el multidimensional se debe realizar una operación entre una matriz y un vector, mientras que en el enfoque que hemos llamado tradicional implica recorrer la lista de predicados que potencialmente unen un sujeto y un predicado.

3.2. Consultas más frecuentes en SPARQL

En el trabajo de Arias et al. (2011) [18] se estudia cómo se compone la mayor parte de las consultas SPARQL. Para ello tomaron como base 3 millones de consultas obtenidas de DBPedia. En particular obtuvieron los logs de consultas relacionados al USEWOD2011 *challenge*, el cual consistía en varios meses de recolección de consultas de usuarios de DBPedia y *Semantic Web Dog Food*(SWDF). Se consideraron estas fuentes como válidas, ya que tenían una composición suficientemente heterogénea, dada por consultas generadas por computadora y creadas por humanos.

Primero separaron las consultas entre qué tipo era más común, donde las consultas SELECT resultaron ser el 96.9 % y 99.7 % de DBPedia y SWDF respectivamente.

Luego, miraron las funciones del lenguaje, la más común es FILTER, con 49 % en ambas fuentes. Por último, miraron las funciones comparativas, encontraron que el de igualdad es el que tiene una utilización mayor, con 23 % de las consultas teniendo al menos uno, donde en casi la totalidad de los usos, este afecta solamente a una variable.

En general el resto de las funciones de SPARQL no son demasiado utilizadas, a excepción del UNION, el cual si fue encontrado en aproximadamente el 10 % de las consultas.

Pasando a otra estadística interesante, observaron que las consultas relacionadas a los tipos de tuplas que se registraban eran las siguientes:

- Tuplas con Sujeto y Predicado constantes y Objeto variable eran las más comunes, con 66 % en DBPedia y 48 % en SWDF.
- Seguidas por Sujeto constante y Predicado y Objeto variables con 22 % en DBPedia y 0.52 % en SWDF.
- Mientras que Predicado y Objeto constantes y Sujeto variable obtuvo 7 % en DBPedia y 46 % en SWDF.

Viendo estos datos, observaron que la diferencia entre el segundo y tercer puesto en relación a la fuente SWDF y DBPedia se diferencian fuertemente en estas. Suponen que esto se debe a la naturaleza semántica de estas bases de datos.

También estos datos sirven como resultados estadísticos a la hora de crear un sistema de indización, donde se ve pueden observar los mejores índices a crear para acelerar las consultas (en estos casos, Sujeto-Predicado, Sujeto y Predicado-Objeto serían los más recomendables).

Luego de investigar como estos patrones detectados se combinan entre ellos, en particular, lo primero que estudiaron es cuántas tuplas existen por consulta. Lo más común fue una única tupla por consulta, con resultados variados, 97 % en SWDF y 66 % en DBPedia. A partir de estos número resolvieron estudiar únicamente el caso de DBPedia, ya que las consultas de SWDF son casi en su totalidad de una tupla.

Estudiaron el JOIN con una variable común entre dos tuplas, es interesante ver

qué combinaciones de JOIN son las más utilizadas, es decir, en qué posiciones se encuentra la variable en ambas tuplas.

Las posibles combinaciones son: Sujeto-Sujeto, Predicado-Predicado, Objeto-Objeto, Sujeto-Predicado, Sujeto-Objeto y Predicado-Objeto. Observaron que el JOIN más común es el Sujeto-Sujeto (60 %), seguido luego por Sujeto-Objeto (34,5 %), y el tercer lugar por Objeto-Objeto (4,5 %).

Determinaron que los grafos son de tipo estrella o incluyen cadenas largas. Para verificar la teoría de cadenas largas analizaron los grafos resultados de las consultas, en particular, analizaron el camino más largo que estos presenten. Encontraron que el 98 % de ellos tienen solo largo 1, con lo cual se descartó esta teoría. Por otro lado, para analizar la teoría que los grafos resultado son de tipo estrella analizaron dichos grafos. Encontraron que en SWDF la gran mayoría son de tipo nodo central con una hoja. Pero para DBPedia, si bien este mismo tipo de grafo es el más usual, tiene un 66 % de ocurrencias, es seguido por grafos con un nodo central y 3 hojas con el 27 % de las ocurrencias. Los siguientes tipos de grafos encontrados ya tienen al menos un grado de magnitud menor, sin embargo, la mayor parte de estos otros tipos también son de tipo estrella. Por lo cual los autores concluyen que si bien depende de la base de datos, aproximadamente un 33 % de los grafos resultados son de tipo estrella.

3.3. Operaciones

Luego de analizado el artículo anterior de Arias et al. (2011), se decidió que las 6 operaciones más comunes serían las que se resolverían. Se consideran entonces consultas con uno y dos parámetros, y se incluye una consulta de tipo JOIN con el objetivo de observar su comportamiento en la representación matricial utilizando GPU. A continuación se presenta el conjunto de consultas objetivo del presente trabajo y su implementación mediante operaciones de ALN.
Para la especificación de operaciones en ALN se utilizan las siguientes definiciones:

- M matriz que modela grafo, donde la fila representa el sujeto, la columna el objeto y dentro de la celda se enumeran los predicados que los vinculan

(siendo el orden irrelevante).

- $v(s) = v(0, ...0, 1, 0, ...0)$ donde la posición del 1 corresponde al índice que le corresponde al sujeto s en la matriz M (análogo para los objetos).
- n cantidad de objetos en el grafo (sujetos, predicados y objetos).
- *objects* vector que contiene todos los elementos que son objeto en alguna tripla.
- *subjects* vector que contiene todos los elementos que son sujeto en alguna tripla.
- *predicates* vector que contiene todos los elementos que son predicado en alguna tripla.

3.3.1. Consulta 1

Consulta 1 Sujeto y predicado constantes

```
1 SELECT ?o
2 WHERE { s p ?o }
```

Esta es la consulta con el patrón más común, apareciendo en un 66 % de las consultas de DBPedia y un 48 % de las de SWDF. En el caso del prototipo, involucra hacer una selección de la fila del sujeto especificado (multiplicar por el vector que tiene 1 en la fila indicada) y luego filtrar todos los resultados (usar la función cumplePredicado) de la fila por el identificador del predicado seleccionado. Las celdas que queden con al menos un predicado indican la columna de el o los objetos que deben ser devueltos. Su traducción a ALN consiste en multiplicar por el vector que tiene 1 en la fila correspondiente al sujeto en cuestión y luego filtrar. Las celdas que queden con al menos un predicado indican la columna de el o los objetos que deben ser devueltos. A continuación se presenta un pseudocódigo.

Pseudocódigo 1 Consulta 1 en ALN

Require: M es la matriz que representa al grafo, s y p son el sujeto y predicado que se buscan
Ensure: o es un conjunto de objetos que satisfacen el patrón
1: $w \leftarrow v(s) \times M$
2: **for all** $i \in 1..n$ **do**
3: **if** $p \in w[i]$ **then**
4: $o \leftarrow o \cup objects[i]$
5: **end if**
6: **end for**

3.3.2. Consulta 2

Consulta 2 Sujeto y predicado constantes más JOIN

```
1   SELECT ?o2
2   WHERE { s p1 ?o1.
3           ?o1 p2 ?o2
4           }
```

En esta consulta se busca probar que tan eficiente puede ser el prototipo para componer más operaciones. Se construyó utilizando el mismo patrón que para la Consulta 1, es decir, fijando sujeto y predicado (a excepción obvia del segundo sujeto, que debía ser el objeto obtenido en la primer tripla). Analizando la consulta usando álgebra se observa que se reduce a la primer consulta para la primer mitad, una vez obtenidos los objetos, estos son usados nuevamente como en la consulta anterior, es decir, para todos los objetos que cumplen en el primer paso (análogo a lo mostrado en la Consulta 1) se vuelve a hacer exactamente lo mismo. Su implementación en ALN consiste en realizar lo mismo que para la consulta 1, y una vez obtenidos los objetos correspondientes a la variable ?o1 estos son ingresados como parámetro de entrada nuevamente. A continuación se presenta un pseudocódigo.

Pseudocódigo 2 Consulta 2 en ALN

Require: M es la matriz que representa al grafo, s, $p1$ y $p2$ son el sujeto y los dos predicados parámetros
Ensure: o es un conjunto de objetos que satisfacen ambos patrones

```
1:  w ← v(s) × M
2:  for all i ∈ 1..n do
3:      if p ∈ w[i] then
4:          μ ← v(i) × M
5:          for all j ∈ 1..n do
6:              if p ∈ μ[j] then
7:                  o ← o ∪ objects[j]
8:              end if
9:          end for
10:     end if
11: end for
```

3.3.3. Consulta 3

Consulta 3 Sujeto y objeto constantes

```
1  SELECT ?p
2  WHERE { s ?p o }
```

Si bien este patrón es uno de los menos comunes (según Arias et al. (2011)) junto con la Consulta 4 fue creada para completar los tres casos bases fijando dos elementos. Además, supone un caso interesante, dado que la variable libre es la tercer dimensión de nuestra matriz. En el caso de esta consulta, primero se debe fijar una fila y una columna, para ello se multiplica la matriz por un vector seleccionando el sujeto. El vector que se obtiene es multiplicado por otro vector para seleccionar el objeto, lo que produce una única celda que contiene el/los predicados que cumplen la consulta. A continuación se presenta un pseudocódigo.

Pseudocódigo 3 Consulta 3 en ALN

Require: M es la matriz que representa al grafo, s, y o son el sujeto y objeto parámetros
Ensure: p es un conjunto de predicados que satisfacen el patrón
1: $w \leftarrow v(s) \times M$
2: $i \leftarrow w \times v(o)$
3: **for all** $p_i \in predicate[i]$ **do**
4: $\quad p \leftarrow p \cup p_i$
5: **end for**

3.3.4. Consulta 4

Consulta 4 Predicado y objeto constantes

```
1  SELECT ?s
2  WHERE { ?s p o }
```

Este patrón no se encuentra en muchas consultas de DBPedia (solo 7 %) pero sin embargo en SWDF aparece en un 46 % de las mismas. Se estima que esto se debe principalmente a la diferencia en tipo de contenido entre ambas bases de datos. Por lo cual es interesante ver el desempeño de esta consulta. Esta consulta es análoga a la Consulta 1 desde el punto de vista del prototipo, en lugar de multiplicar para seleccionar un sujeto, se multiplica para seleccionar un objeto, luego de eso, se validan los predicados de igual manera que se hace en la Consulta 1. A continuación se presenta un pseudocódigo.

Pseudocódigo 4 Consulta 4 en ALN

Require: M es la matriz que representa al grafo, p y o son el predicado y objeto que se buscan
Ensure: s es un conjunto de sujetos que satisfacen el patrón
1: $w \leftarrow v(o) \times M$
2: **for all** $i \in 1..n$ **do**
3: **if** $p \in w[i]$ **then**
4: $s \leftarrow s \cup subjects[i]$
5: **end if**
6: **end for**

3.3.5. Consulta 5

Consulta 5 Sujeto constante

```
1  SELECT ?p ?o
2  WHERE { s ?p ?o }
```

Comenzando con las consultas donde hay dos variables, es interesante observar que pasa cuando se fija el sujeto. Este patrón aparece en un 22 % de las consultas de DBPedia, lo cual no es insignificante, sin embargo solo aparece en un 0.5 % de las de SWDF, nuevamente, marcando la diferencia entre ambas bases. A la hora de analizar esta consulta de manera algebráica basta pensar que se están pidiendo todas las triplas que tienen un sujeto fijo, por lo cual lo único que se debe hacer es multiplicar la matriz por un vector que seleccione el sujeto provisto. el vector resultante contendrá todas las triplas que se deben seleccionar. El vector resultante contendrá todas las parejas (predicado, objeto) que se deben retornar. A continuación se presenta un pseudocódigo.

Pseudocódigo 5 Consulta5 en ALN

Require: M es la matriz que representa al grafo, s es el sujeto que se busca
Ensure: w representa parejas (predicado, objeto)
1: $w \leftarrow v(s) \times M$

3.3.6. Consulta 6

Consulta 6 Objeto constante

```
1  SELECT ?s ?p
2  WHERE { s? ?p o }
```

Por último, y continuando con lo observado en la Consulta 5, ahora se fija el objeto. Si bien en este caso tanto en DBPedia como en SWDF no se registran demasiadas consultas que contengan este patrón, generar esta consulta no es muy diferente a generar la Consulta 5. Además, se hizo ya que en el caso de las bases de datos RDF basadas en columnas (Virtuoso por ejemplo), se espera que esta consulta tenga una diferencia marcada con la anterior. Por lo cual, tanto por conveniencia, completitud y para confirmar que se van a observar tiempos similares en el prototipo se introdujo. Análoga a la consulta anterior, en este caso, en lugar de multiplicar la matriz por un vector seleccionando el sujeto, se multiplica un vector que selecciona el objeto por la matriz, obteniendo un vector que nuevamente tiene todas las parejas (sujeto, predicado) que deben ser devueltas. A continuación se presenta un pseudocódigo.

Pseudocódigo 6 Consulta 6 en ALN

Require: M es la matriz que representa al grafo, o es el objeto que se busca
Ensure: w representa parejas (sujeto, predicado)
1: $w \leftarrow M \times v(o)$

3.4. Otras posibles consultas

Con las consultas anteriores se abarcan los tres patrones que se utilizan mayormente en DBPedia y SWDF, según Arias et al. (2011). Además de estos patrones de consultas sencillas, se agrega una encadenada, nuevamente usando el patrón más utilizado (Consulta 2). Con el objetivo de poder analizar si el prototipo podía hacer un mejor trabajo que otras alternativas que no usan GPU.

Por otro lado ya que según Arias et al. (2011), todas las consultas que usan otros operadores se utilizan menos del 3 % en DBPedia y menos de 1 % en SWDF se decidió dejarlas por fuera del área a estudiar.

3.5. Representación física

Representar un grafo interesante (como los que modelan redes sociales) en una matriz tiene la particularidad de generar matrices dispersas y de grandes dimensiones. Esto es dado que la probabilidad de que exista un vínculo entre 2

nodos es poca, y la cantidad de nodos es alta. Mapearlas de forma directa por lo tanto no es una buena opción, ya que se desperdiciaría mucha memoria en datos que no son relevantes (valores nulos) y las operaciones implican grandes volúmenes de trabajo.

Se busca entonces la mejor forma de cargar y operar con las matrices en algún formato que ahorre espacio y disminuya tiempo de procesamiento. Para esto se analizarán tres alternativas de formato de almacenamiento de matrices dispersas, en particular: CSR (*Compressed Storage Row*), CDS (*Compressed Diagonal Storage*) e HYB (*Hybrid*). Se mide y compara la performance de estas representaciones para tres operaciones fundamentales, cargar la matriz desde un archivo, multiplicar la matriz por un vector y multiplicar un vector por la matriz, con el objetivo de elegir el óptimo para este problema.

La matriz de la Figura 2 será usada de ejemplo para explicar los distintos formatos.

$$E = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 0 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

Figura 2: Matriz E de ejemplo.

Cabe mencionar que en una matriz de dimensiones reducidas como la de la Figura 2 no tiene sentido buscar un mejor formato de mapeo ya que el espacio ahorrado es insignificante, se utiliza únicamente con el objetivo de visualizar las distintas estructuras necesarias en los formatos a analizar.

3.5.1. Formato *Compressed Storage Row*

En este formato se tienen tres vectores: Uno almacena los valores de las entradas no nulas (vector-datos) (las entradas nulas son las de valor 0) es de tamaño N, siendo N la cantidad de elementos no nulos de la matriz. Otro vector (vector-columnas) que guarda los índices de las columnas en que se encuentran los elementos del vector-datos, también de tamaño N y el tercero (vector-índices) almacena el número acumulado de elementos distintos de cero

por filas dejando el primer valor en 0. Este último vector es de tamaño $K+1$, siendo k la cantidad de filas de la matriz.

Para la matriz de ejemplo la representación en este formato quedaría:

- vector-datos : [1,2,3,4,5,6];
- vector-columnas : [0,2,2,0,1,2];
- vector-índices : [0 2 3 6];

3.5.2. Formato *Compressed Diagonal Storage*

En este formato se tiene una matriz en donde la primer entrada de cada fila representa el índice de la diagonal a la que corresponde esa fila. La diagonal principal tiene índice 0, el índice crece para las diagonales que están por encima de la diagonal principal y decrece para las que están debajo. Además solo se representan las diagonales que tienen algún valor no nulo. La representación de la matriz ejemplo en forma gráfica se muestra en la Figura 3.

$$\begin{pmatrix} -2 & 0 & 0 & 4 \\ -1 & 0 & 5 & 0 \\ 0 & 1 & 0 & 6 \\ 1 & 0 & 3 & 0 \\ 2 & 2 & 0 & 0 \end{pmatrix}$$

Figura 3: Matriz E en formato CDS.

3.5.3. Formato *Hybrid*

Como su nombre lo indica esta estrategia de almacenamiento es una mezcla de dos formatos, por un lado el CDS presentado en la sección anterior y, por otro, el denominado simple o elemental, en donde se tienen tres vectores, uno contienen los valores no nulos, otro las columnas de cada valor no nulo y un tercero que indica la fila de cada valor no nulo.

En el formato CDS si una diagonal tiene un solo valor válido, por ejemplo la diagonal -2, se tiene que guardar toda la diagonal, esto no parece muy eficiente, el formato HYB intenta resolver este problema en donde las diagonales

con pocos valores válidos son almacenadas en el formato elemental. Se tiene que definir un criterio que determine qué diagonal almacenar con formato CDS y cual con formato elemental.

Utilizando como criterio por ejemplo que la diagonal tiene que tener más de la mitad de sus entradas válidas para que se almacenen en formato diagonal, la representación quedaría como se muestra en las Figuras 4 y 5. En la Figura 4 se muestra la sección de la matriz E almacenada en formato elemental mientras que en la Figura 5 se muestra la sección de la matriz E almacenada en formato CDS.

$$\begin{pmatrix} filas & columnas & datos \\ 0 & 2 & 2 \\ 1 & 2 & 3 \\ 2 & 0 & 4 \\ 2 & 1 & 5 \end{pmatrix}$$

Figura 4: Sección de la matriz E almacenada en formato elemental.

$$\begin{pmatrix} 0 & 1 & 0 & 6 \end{pmatrix}$$

Figura 5: Sección de la matriz E almacenada en formato CDS.

3.5.4. Selección de la representación física basada en experimentos

La evaluación experimental incluye, cargar la matriz desde un archivo, multiplicar la matriz por un vector y multiplicar un vector por la matriz, con juegos de datos variados obtenidos de *The SuiteSparse Matrix Collection [19].* Se muestran los resultados en las Tablas 4, 5 y 6. Los resultados son el tiempo promedio de cinco ejecuciones independientes.

Entradas no nulas	CSR	CDS	HYB
6	0,000072	0,000077	0,000086
35	0,000117	0,000126	0,000128
88627	0,426136	0,644117	0,447438
412148	0,512749	0,640551	0,601455
1891669	2,386610	-	2,799282

Tabla 4: Tiempos de ejecución (en seg.) de cargar la matriz.

Como se observa en la Tabla 4 con una cantidad de entradas no nulas muy pequeña los tiempos son muy similares, pero a medida que la cantidad de entradas aumenta se empieza a notar una diferencia en favor de los formatos CSR e HYB. Con una cantidad de entradas de 1891669, el formato CSR obtiene una performance un poco mejor que HYB. El tiempo del formato CDS no figura dado que requería más recursos computacionales que los recursos disponibles en la plataforma utilizada.

Entradas no nulas	CSR	CDS	HYB
6	0,073590	0,074808	0,079638
35	0,085577	0,088232	0,081770
88627	0,083897	0,408101	0,095506
412148	0,080070	0,324645	0,096726
1891669	0,101785	-	0,149929

Tabla 5: Tiempos de ejecución (en seg.) de $Matriz \times Vector$.

En la Tabla 5 se puede observar que con 6 o 35 entradas los tiempos son casi idénticos pero a partir de 88627 entradas el formato CDS demora 5 veces más mientras que CSR no ve afectado su tiempo e HYB aumenta solo un poco. La diferencias entre CSR e HYB son despreciables salvo en la última entrada de la tabla donde HYB demora 50 % más aproximadamente.

Entradas no nulas	CSR	CDS	HYB
6	0,081819	0,076892	0,081486
35	0,079974	0,080255	0,075778
88627	0,085115	0,365529	0,093926
412148	0,119318	0,336381	0,104435
1891669	0,095897	-	0,142981

Tabla 6: Tiempos de ejecución (en seg.) de $Vector \times Matriz$.

En la Tabla 6 se repite la misma situación que en las anteriores, el formato CSR resulta ser el de mejor desempeño, seguido por el formato HYB. Una vez observados estos resultados también se puede hacer la presunción de que el buen desempeño del formato HYB se debe a que muchos de sus elementos se almacenaron en formato elemental, ya que si se hubieran encontrado en el formato CDS los tiempos serían más similares a los obtenidos para esa estructura.

Dados los resultados obtenidos y el análisis efectuado parece certero utilizar la representación CSR para llevar a cabo el prototipo.

Capítulo 4

Evaluación experimental

En este capítulo se describen los casos de prueba generados para comparar el desempeño del prototipo contra soluciones ya existentes. Se explica como se generaron dichos casos de prueba, como también las diferencias de desempeño encontradas luego de ejecutados estos casos en las distintas soluciones.

4.1. Generación de casos de prueba

Para realizar pruebas se necesita crear grafos de millones de elementos dado que son los grafos sobre los que interesa poder efectuar operaciones. Para esto se analizaron las distintas herramientas disponibles en la actualidad. A continuación se describe cuál fue la elegida y la manipulación que tuvo que efectuarse para adaptar la salida de la herramienta en cuestión.

4.1.1. Formato de entrada

Es necesario definir un formato de entrada para el prototipo, de manera de poder realizar la carga de una base de datos de grafos. Para ello vale la pena observar que el objetivo del prototipo es tratar el grafo como si fuese una matriz, por lo cual se utiliza una función biyectiva, que haga corresponder un elemento del grafo a un número entero, el cual se utiliza como índice posicional en la matriz.

Para esto se separa la entrada en varios archivos, el primero de ellos contiene la matriz propiamente dicha. Esta se encuentra escrita en triplas, es decir, cada entrada en este archivo se compone de 3 elementos, el primero y el segundo son

los índices del sujeto y el objeto de la tripla, mientras que el tercer elemento está compuesto por todos los índices de los predicados que los unen.

Luego se tiene por otro lado la salida de la función biyectiva entre los elementos y sus índices. Este formato de datos hace que la carga en memoria por parte del prototipo de la base de datos sea rápida.

Berlin SPARQL Benchmark (BSB)

Para comprobar el desempeño del prototipo es necesario compararlo con otras soluciones, para ello se utilizó el generador de datos que es parte del Berlin SPARQL Benchmark (BSB) [20], este fue seleccionado debido a su popularidad y a que su modelo de datos es sencillo.

El modelo de datos presentado por BSB consiste en ocho tipos de entidades. El diagrama de estos modelos puede ser visto en la Figura 6.

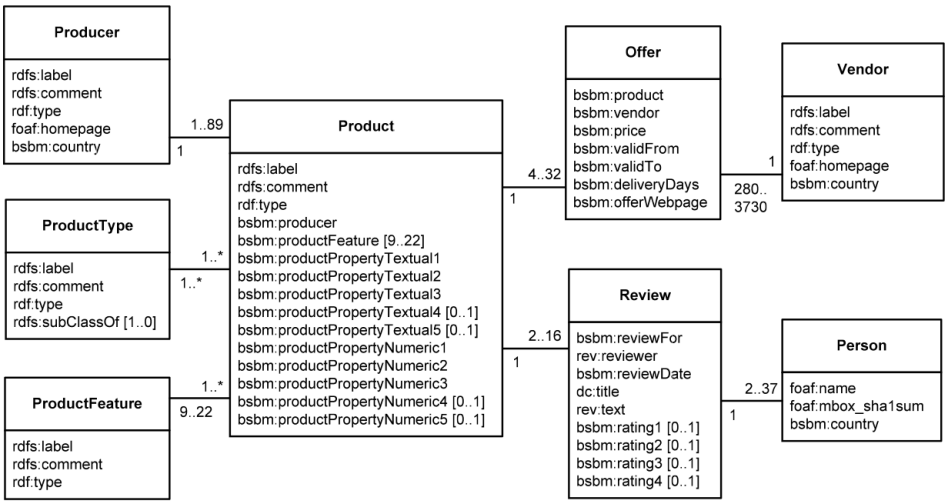

Figura 6: Modelo de Berlin SPARQL Benchmark.

Para generar estas bases de datos se utiliza una herramienta, la cual deja especificar cuantos Productos debe tener el grafo generado. La salida de esta herramienta es un archivo .nt, el cual se compone de triplas en texto plano, donde los sujetos y predicados están rodeados de < y > indicando que se trata de una URI, mientras que los objetos dependen de su tipo, si son de tipo URI,

llevan los mismos indicadores que los anteriores, caso contrario se rodea de comillas el valor y opcionalmente, si el objeto es de algún tipo conocido se sigue del indicador $\wedge\wedge$ el cual vincula este valor a un tipo que está indicado en formato URI.

A continuación se muestra un ejemplo de cada tipo:

- `<http://www4.wiwiss.fu-berlin.de/instances/ProductType1>`
- `"microprocessor mops reigned"`
- `"2000-07-05"^^<http://www.w3.org/2001/XMLSchema#date>`

Preprocesamiento

Se observa entonces que el formato de salida de BSB y el de entrada del prototipo son distintos. Por lo cual, es necesaria la creación de un algoritmo traductor. Este debe ser capaz de generar un índice único para cada elemento del grafo y además asociar todas las triplas que compartan sujeto y objeto. Se utiliza JAVA como lenguaje para este traductor por el balance entre portabilidad, potencia y rapidez del mismo. El algoritmo básico está descrito a continuación y sobre el mismo se itera hasta terminar de leer todo el archivo línea a línea.

- Leer línea del archivo original (formato **Sujeto Objeto Predicado**).
- Buscar cada elemento dentro de un diccionario, si el mismo existe, tomar el índice asociado, caso contrario crear un índice e insertarlo en el mapa.
- Buscar la pareja **Sujeto-Objeto** dentro de la estructura que almacena las triplas, si existe, tomarla y agregarle el índice del **Predicado**, caso contrario crearla, asociarle el **Predicado** y almacenarla en la estructura mencionada.

Una vez terminados estos pasos solo resta escribir en los archivos descritos con anterioridad las estructuras que quedaron en memoria.

Este algoritmo permite que el archivo de entrada sea recorrido una única vez y se procese línea a línea. A simple vista podemos ver que el primer potencial cuello de botella serán los diccionarios, tanto por inserción/búsqueda (en relación al resto del algoritmo) como también por espacio ocupado en memoria (ya que los elementos son almacenados como cadenas de caracteres). Con respecto

al índice generado, se utiliza una variable de tipo entero, la cual se inicializa en 0 y se incrementa en uno cada vez que es necesario.

4.2. Carga de datos

Como se expresó anteriormente, el formato utilizado es CSR pero se le agregaron otras estructuras necesarias para resolver las operaciones en cuestión y para mostrar los resultado en una forma más parecida a las aplicaciones similares al prototipo desarrollado. Entonces, la carga de datos se realiza en dos partes, por un lado se realiza la carga de la matriz que indica las conexiones de los nodos y por el otro con la carga de los sujetos, objetos y predicados.

En ambas cargas se leen los cabezales de los archivos que indican los tamaños de estos de manera de poder inicializar las estructuras, luego se procede a leer línea por línea agregando este nuevo dato a la estructura.

En un principio se pedía memoria nueva para cada entrada pero esto provocaba que en pocos minutos se consumiera toda la memoria del sistema lo cual atentaba contra nuestra intención de realizar pruebas con grandes cantidades de datos. Luego se optó por el uso del operador de C *realloc* que resultó ser más eficiente en el manejo de memoria.

4.3. Hipótesis previa a las pruebas

En base al diseño del prototipo, antes de realizar las pruebas se hicieron las siguientes hipótesis.

Analizando las posibles ventajas que puede ofrecer el prototipo, se prevé que en situaciones donde el resultado sea un sub-grafo de gran cantidad de nodos/aristas se verán las principales ventajas. Esto de debe a que las implementaciones basadas en CPU deberán iterar para recorrer todos los nodos, mientras que en el prototipo, todos los nodos serán obtenidos al realizar una operación sobre la GPU.

Contra implementaciones basadas en grafos, como la de RDF-3X es más difícil

saber qué tipo de ventajas se pueden alcanzar, ya que depende en gran medida como hace esta solución para decidir el orden en que se deben recorrer los nodos para obtener el sub-grafo solución.

4.4. Conjuntos de datos de prueba generados

Las entradas para ser probadas en RDF-3X, Virtuoso y en el prototipo, fueron elegidas deliberadamente. Si bien se podría haber automatizado, se cree que haber generado las pruebas de esta forma da la oportunidad de crear ciertas operaciones interesantes, sobretodo algunas donde el sub-grafo resultante es grande, o en casos tan pequeño que solo involucra un nodo y una o más aristas.

Estas son generadas tomando nodos aleatorios en el dataset, de manera de poder promediar mejor los resultados y observar si existe alguna diferencia entre una tupla que fuese numéricamente (según el índice generado por el traductor) más grande que otra. Algunas operaciones se prestan más, dado el modelo de datos de BSB, a tener más de un resultado, mientras que otras no, por lo cual mirando el conjunto de datos de prueba se puede observar que hay operaciones en las cuales predominan los resultados con sub-grafos grandes mientras en otras mayormente serán sub-grafos de dos nodos y una arista.

Como se mencionó antes, se utilizan corpus de datos generados por la herramienta de generación de modelos de BSB. En particular, se analiza si hay alguna tendencia a medida que crece el dataset, por lo cual se generan datasets crecientes explotando la herramienta que pide la especificación de cuántos Productos deben existir en el conjunto generado.

- 100.000 Productos (32 millones triplas aprox.).
- 150.000 Productos (49 millones triplas aprox.).
- 200.000 Productos (65 millones triplas aprox.).
- 250.000 Productos (98 millones triplas aprox.).
- 500.000 Productos (195 millones triplas aprox.).

Luego de generado y traducido el dataset de 500.000 Productos, y haber generado el conjunto de pruebas del mismo, la memoria que requiere no es com-

patible con el prototipo en el equipo de pruebas. Por lo tanto, se abandona el uso de este dataset.

4.5. Hardware utilizado

Antes de detallar los resultados obtenidos es necesario describir el hardware utilizado para generar y probar el sistema. Para generar los datos se utilizó un hardware con las siguientes especificaciones:

- SO: Ubuntu 16.0.
- Procesador: Intel® Core™ i7-6700 CPU @ 3.40GHz.
- RAM: 64 GB DDR4.

Para realizar las pruebas se utilizó el siguiente hardware:

- SO: Windows 10.
- Procesador: Intel® Core™ i7-8550U CPU @ 1.80GHz.
- RAM: 12 GB DDR4.
- Tarjeta grafica: NVIDIA® GeForce® MX150 2GB VRAM.

4.6. Discusión de los resultados

A continuación, para cada consulta de la sección 3.3 se presentan los tiempos de ejecución en segundos, también se muestra el número de operación (agregado para referenciar a la ejecución de la que se está hablando) y la cantidad de triplas devueltas por la consulta. Los resultados se agrupan por consulta. A continuación se describen los encabezados de las tablas 7-12:

- `#Prods`
  Especifica cuantos productos contiene el dataset.
- `NOp`
  Declara el número de operación (conjunto de entrada particular).
- `#Res`
  Es el número de triplas resultado de la consulta específica.
- `TProt`
  Tiempo en segundos del prototipo para resolver la consulta.
- `TRDF3X`
  Tiempo en segundos de RDF-3X para resolver la consulta.
- `TVirt`
  Tiempo en segundos de Virtuoso 7 para resolver la consulta.
- `TProdSImpr`
  Tiempo en segundos del prototipo para resolver la consulta, sin contabilizar tiempo de impresión en pantalla del resultado.

#Prods	NOp	#Res	TProt	TRDF3X	TVirt	TProtSImpr
100.000	1	1	0,673	0,105	0,016	0,653
	2	19	0,733	0,141	0,016	0,668
	3	1	0,673	0,109	0,016	0,653
	4	1	0,667	0,094	0,016	0,657
150.000	1	1	0,985	0,129	0,032	0,967
	2	1	0,996	0,140	0,015	0,964
	3	1	0,992	0,252	0,063	0,962
	4	1	0,998	0,271	0,016	0,963
200.000	1	22	1,320	0,189	0,582	1,273
	2	19	1,306	0,180	0,109	1,270
	3	1	1,312	0,183	0,891	1,281
	4	1	1,309	0,190	0,563	1,271
250.000	1	1	1,635	0,223	1,281	1,564
	2	1	1,627	0,323	0,047	1,572
	3	1	1,615	0,235	0,500	1,562
	4	1	1,634	0,234	0,312	1,560

Tabla 7: Tiempos de ejecución de Consulta 1 Datasets 1, 2, 3, 4 (en segundos)

#Prods	NOp	#Res	TProt	TRDF3X	TVirt	TProtSImpr
100.000	1	1	1,450	0,940	0,015	1,299
	2	1	2,047	0,125	0,016	1,299
	3	1	1,357	0,109	0,015	1,316
	4	1	1,423	0,109	0,016	1,316
150.000	1	21	1,984	0,244	0,328	1,922
	2	17	2,02	0,149	0,031	1,922
	3	1	2,023	0,472	0,500	1,949
	4	24	2,013	0,453	0,079	1,921
200.000	1	19	2,611	1,861	0,703	2,563
	2	13	2,605	1,718	0,156	2,531
	3	14	2,634	0,280	0,062	2,542
	4	2	32,152	0,500	2,535	31,187
250.000	1	1	3,272	0,315	0,187	3,153
	2	1	3,530	0,250	0,317	3,122
	3	1	3,601	0,535	0,640	3,124
	4	1	3,906	0,231	0,266	3,176

Tabla 8: Tiempos de ejecución de Consulta 2 Datasets 1, 2, 3, 4 (en segundos)

#Prods	NOp	#Res	TProt	TRDF3X	TVirt	TProtSImpr
100,000	1	1	0,467	0,109	0,016	0,429
	2	2	0,430	0,125	0,016	0,434
	3	3	0,440	0,094	0,016	0,430
	4	4	0,440	0,109	0,016	0,427
150.000	1	3	0,647	0,451	0,047	0,645
	2	1	0,628	0,423	0,047	0,625
	3	2	0,681	0,476	0,047	0,635
	4	1	0,873	0,481	0,093	0,633
200.000	1	3	1,231	0,277	0,172	0,833
	2	1	1,235	0,185	0,078	0,837
	3	2	1,068	0,452	0,078	0,831
	4	2	0,836	0,183	0,125	0,826
250.000	1	1	1,040	0,260	0,203	1,026
	2	2	1,021	0,250	0,185	1,019
	3	3	1,092	0,220	0,156	1,020
	4	4	1,034	0,236	0,188	1,030

Tabla 9: Tiempos de ejecución de Consulta 3 Datasets 1, 2, 3, 4 (en segundos)

#Prods	NOp	#Res	TProt	TRDF3X	TVirt	TProtSImpr
100.000	1	2.429	0,747	0,266	0,141	0,629
	2	2.649	0,730	0,293	NA?	0,633
	3	508.117	16,307	32,818	34,000	0,633
	4	1	0,670	0,108	0,015	0,626
150.000	1	1.500.000	40,559	103,691	117,375	0,988
	2	12.224	1,342	0,971	0,938	0,962
	3	87.268	4,279	6,573	9,047	0,926
	4	4.038	3,723	Error	1,156	0,929
200.000	1	6.970	4,900	8,571	16,890	1,229
	2	2.183	1,331	0,350	0,562	1,226
	3	100	1,269	0,181	0,250	1,236
	4	60.843	3,439	4,270	8,172	1,223
250.000	1	1917	1,892	2,818	1,218	1,539
	2	2.500.000	89,208	194,418	184,172	1,985
	3	51.518	1,567	5,917	9,766	1,533
	4	1	1,576	0,195	0,313	1,545

Tabla 10: Tiempos de ejecución de Consulta 4 Datasets 1, 2, 3, 4 (en segundos)

#Prods	NOp	#Res	TProt	TRDF3X	TVirt	TProtSImpr
100.000	1	9	0.397	0,115	0,016	0,359
	2	39	0.400	0,125	0,015	0,365
	3	35	0.397	0,118	0,015	0,361
	4	6	0.393	0,125	0,016	0,356
150.000	1	10	0,581	1,772	0,265	0,534
	2	12	0,593	1,712	0,140	0,538
	3	6	0,585	0,142	0,031	0,540
	4	5	0,584	0,144	0,047	0,543
200.000	1	33	1,062	0,185	0,281	0,729
	2	9	0,772	0,174	0,375	0,708
	3	9	0,764	0,190	0,282	0,717
	4	11	0,797	0,186	0,156	0,708
250.000	1	40	0,958	0,224	0,266	0,898
	2	39	0,988	0,223	0,203	0,895
	3	34	0,981	0,257	0,234	0,895
	4	6	0,958	0,232	0,141	0,893

Tabla 11: Tiempos de ejecución de Consulta 5 Datasets 1, 2, 3, 4 (en segundos)

#Prods	NOp	#Res	TProt	TRDF3X	TVirt	TProtSImpr
100.000	1	10	0,373	0,114	0,015	0,344
	2	355	0,413	0,138	0,047	0,347
	3	1	0,383	0,112	0,015	0,337
	4	2066	0,610	0,412	0,375	0,346
150.000	1	1.500.000	159,787	118,831	172,641	0,510
	2	11.182	1,447	0,967	1,422	0,507
	3	859.166	94,693	65,481	108,313	0,514
	4	76.848	6,911	7,301	9,469	0,510
200.000	1	63.745	6,455	5,049	20,844	0,691
	2	42	0,761	0,181	0,062	0,689
	3	1,672	0,988	0,403	0,422	0,677
	4	4.000.000	385,350	312,986	472,016	0,674
250.000	1	1.917	1,207	0,542	0,485	0,842
	2	2.500.000	233,428	235,820	276,828	0,839
	3	51,518	7,080	5,020	21,187	0,841
	4	1	0,890	0,193	0,094	0,845

Tabla 12: Tiempos de ejecución de Consulta 6 Datasets 1, 2, 3, 4 (en segundos)

Es importante aclarar que los tiempos en general no son comparables estrictamente, esto se debe a que en los tiempos de RDF-3X y Virtuoso se incluyen tiempos de parseo y planificación de las consultas, esto no ocurre en el prototipo, ya que el mismo tiene pre-planificadas las consultas debido a la forma en la que fue programado. Cabe destacar además que RDF-3X y Virtuoso tampoco son comparables entre sí, ya que RDF-3X no se encuentra levantado en memoria (diferente a Virtuoso que corre en memoria como un DBMS común).

Lo primero que se puede observar es que en las Consultas 4 y 6 se obtiene una mejoría notoria respecto a otros motores de bases de datos. Esto se puede deducir ya que a pesar de como se mencionó previamente, y los tiempos no son directamente comparables, si se puede extrapolar a partir de los tiempos de otras consultas. Esta mejoría es gracias al tamaño del subgrafo resultado, al ser este muy grande, una implementación basada en CPU tiene un orden mayor que en GPU.

Otro detalle interesante, el tiempo del prototipo crece con el tamaño del resultado a pesar de lo mencionado en el punto anterior, sin embargo podemos observar gracias a la columna del tiempo del prototipo sin tomar en cuenta la impresión que la cantidad de resultados en general no afecta el tiempo de procesamiento en sí, por lo cual llegamos a la conclusión de que dicho crecimiento de tiempo se debe a la impresión en pantalla y no al procesamiento.

Es interesante comparar la Consulta 4 con la Consulta 6, observando la Consulta 6 los tiempos para consultas equivalentes son mayores. Cuando se analiza la implementación, la Operación 4 lleva una operación de GPU más que la Operación 6. Esto, luego de analizarlo, se debe a la forma que se utiliza para imprimir resultados.
Se puede observar que la mayor parte del tiempo se gasta en mostrar al usuario un resultado legible (esto lo podemos ver si comparamos el tiempo total del prototipo contra el tiempo sin tomar en cuenta la impresión).

Una cualidad de Virtuoso que se puede observar es la utilización de algún tipo de cache para valores ya utilizados en consultas anteriores o durante la ejecución de la misma. Se puede ver esto analizando diferentes operaciones de la misma consulta (por ejemplo en la Consulta 1). También tiene un menor

tiempo de impresión en pantalla en general (esto aplicaría a RDF-3X y Virtuoso). En el caso del prototipo no se le dió relevancia a la optimización de este proceso. Luego de ver estas optimizaciones se puede pensar en que algunas pueden aplicar al prototipo. Una muy sencilla sería tener precargada la matriz en la memoria de la GPU, esto se debe a que en el hardware utilizado para las pruebas, el tiempo total entre la copia de RAM a GPU, ejecución del código en GPU y copia de GPU a RAM está dominado por los tiempos de copia.

Además se puede ver que el prototipo (y en menor medida Virtuoso) sufren en la Operación 4 de la Consulta 2, del dataset de 200.000 productos, ya que dicha consulta está creada con el objetivo de que la primer tripla de la consulta obtenga muchos resultados previos. Esto implica que el prototipo debe iterar para cada una de ellas buscando segundas triplas que cumplan las condiciones de la consulta. Esto genera que haya una diferencia de tiempo considerable, Virtuoso debe operar de una manera similar ya que también se ve un incremento de su tiempo.

Como conclusión primaria luego de ver estos resultados se puede observar que las técnicas expresadas con anterioridad, para la resolución de consultas de bases de datos de grafos tienen utilidad en ciertos contextos. Fundamentalmente donde los resultados (finales o intermedios) de las consultas sean grandes o también en caso de poseer hardware especializado, donde la comunicación entre CPU y GPU sea considerablemente más rápida (por ejemplo utilizando tecnologías como NVLink [21]). También cabe destacar que el hardware que fue utilizado es de bajo porte, donde se utilizó una GPU dedicada dentro de una computadora portátil, siendo esta alrededor de 20 veces menos potente que una única GPU especifica para este uso (NVIDIA MX150 [22] vs NVIDIA Tesla V100 [23]).

Capítulo 5

Conclusiones y trabajo futuro

Al inicio de este proyecto se planteó la idea de explotar la relación natural que existe entre los grafos y su representación mediante matrices (dispersas), y la consecuente relación entre las operaciones de consulta sobre grafos y operaciones algebraicas sobre las matrices que los representan. Pudiendo aprovechar de esta forma la potencialidad de las GPUs para realizar operaciones de ALN.

Se definió entonces una estructura de datos que permite cargar y operar con matrices de forma eficiente en la GPU, para esto se midió y comparó el desempeño de tres representaciones diferentes. También se detalló una secuencia de operaciones de ALN que resuelven de forma efectiva algunas consultas sobre estos grafos y se implementó un prototipo que implementa esta secuencia.

El prototipo desarrollado carga un grafo RDF y permite interactuar para ejecutar distintas consultas (las más comunes como fue presentado en la Sección 3.2), mostrando los resultados en pantalla de forma exitosa. Luego de ejecutadas las pruebas tanto en el prototipo como en otras implementaciones de motores de bases de datos RDF se pueden comparar, al menos en gran escala, estos resultados.

Los resultados experimentales muestran que existe una ventaja al utilizar una GPU para los casos de consultas con muchas triplas, obviamente esto se debe a que la implementación basada en la representación del grafo de la base de datos RDF como una matriz y la utilización de operaciones de álgebra lineal para resolver estas consultas no depende (en gran medida) del tamaño del

resultado, es decir, no importa si el grafo resultante está compuesto por 1 o 1.000.000 de triplas, al no tener que iterar sobre estos resultados (salvo claro para mostrarlos), el tiempo de ejecución es fijo.

En gran medida, estos casos fueron para los cuales se generó este prototipo, ya que en otras implementaciones de bases de datos RDF, las triplas se almacenan utilizando otras estructuras en las cuales por lo general hay que iterar para encontrar triplas, tanto sea por columnas (en el caso de Virtuoso por ejemplo), como en un grafo indizado (RDF-3X sería un ejemplo de esto).

Finalmente a partir del presente trabajo se realizó un artículo el cual fue presentado en el XLV Latin American Computing Conference (CLEI 2019). Este articulo se puede encontrar en el Anexo 0.3.

5.1. Trabajo futuro

El prototipo tiene un funcionamiento básico, el cual permitió probar las hipótesis planteadas. Es decir que ciertas consultas se podrían hacer en base a operaciones de ALN, y además estas implementarlas en GPU y, dependiendo del volumen que implicasen los resultados, estas podrían ser realizadas más rápidamente que en cualquiera de las implementaciones actuales en CPU. Sin embargo, el esfuerzo fue planteado como un prototipo, en otras palabras muchas características pueden ser mejoradas. Algunas de estas mejoras posibles se enumeran a continuación.

Incorporación del prototipo a un motor de base de datos abierto. No se cree que sea más conveniente implementar todo lo que falta para que el prototipo se convierta en un motor de base de datos RDF. Sin embargo, sí sería mucho más sencillo el camino opuesto. Tomar un motor existente y, en el momento donde se realiza la planificación de la consulta, analizar si utilizar el prototipo para resolverla o no.

Utilización de cache. El prototipo no utiliza ningún tipo de técnica para acelerar la ejecución, fuera de explotar el poder de cómputo de la GPU para hacer las operaciones necesarias para resolver consultas. Existen varias técnicas y se podrían complementar con varias otras particulares a la implementación

presentada. Específicamente es fácil ver que en todas las operaciones se comienza operando sobre la matriz, esta es bastante grande, por lo cual tenerla pre-cargada en la memoria de la GPU sería una forma sencilla de optimizar tiempos.

Extender las consultas. Si bien el análisis realizado muestra que con las consultas cubiertas se alcanza gran parte de los requerimientos en bases de datos RDF, parece interesante avanzar hacia una propuesta que abarque por completo la especificación de SPARQL 1.1.

Otra línea de investigación que puede ser aprovechada son las operaciones bitwise en GPU, muchos de los resultados intermedios de los algoritmos que resuelven consultas son 0 o 1 esto podría aumentar la eficiencia de las operaciones necesarias.

Por último, corresponde realizar evaluaciones experimentales del prototipo realizado en GPUs de alta gama.

Lista de figuras

Lista de tablas

APÉNDICES

0.1. Dificultades

0.1.1. Traducción

Como se ha mencionado, una de las dificultades importantes es la conversión de datos, desde los formatos estándar (.nt, .n3, etc.), hasta llegar a la estructura interna que se utiliza. En este caso, una matriz en formato CSR y tablas de hashes internos para identificar sujetos, objetos y predicados.

Los formatos normales consisten de archivo o archivos de texto, donde las triplas están impresas en forma plana, separando los elementos de la tripla mediante un carácter (normalmente un espacio en blanco). Como es normal, este proceso no es sencillo ni rápido cuando están involucradas muchas triplas y URIs.

Por ello se decidió que se iba a comenzar con el prototipo basado en la existencia de cuatro archivos. Uno que contiene, además de cabezales que indican cantidad de filas/columnas y elementos, todas las triplas, separadas por espacios (en este caso en lugar de en texto plano, estas URIs ya se encuentran identificadas por un UID).

A su vez, se cuenta con otros tres archivos, donde además de cabezales, que indican la cantidad de elementos del archivo y el largo máximo de la cadena de caracteres más larga, todos los elementos vinculados con su UID. Se comenzó con 3 archivos separados, uno para sujetos, otro objetos y por último uno para predicados.

Para comenzar con el desarrollo del prototipo se generó manualmente una pequeña base de datos. Una vez que el prototipo estuvo funcional con al menos una operación y la misma era correcta sobre este conjunto de datos de prueba, surgió la segunda dificultad, la cual fue generar un programa capaz de cargar el archivo de salida de la herramienta de Berlin.

En principio la tarea básica parece sencilla, tomar el archivo, a cada URI asignarle un UID nuevo, y almacenar las triplas por otro lado. El problema de esto, es que a medida que el programa va avanzando es necesario verificar

que la tripla no se encuentra ya asociada a un UID. Existirían 2 soluciones a esto, almacenar datos en memoria o almacenar en disco, la solución elegida fue almacenar en RAM, la ventaja de esto es que se aceleran los tiempos, la desventaja obviamente es que en general en cualquier sistema no hay tanta memoria RAM como almacenamiento. Por lo cual fue necesario aplicar varias técnicas para disminuir el espacio que ocupaban estas cadenas de caracteres en memoria.

También asociado a este problema, se tiene la dificultad de que a pesar de todas las optimizaciones que tiene el traductor, para grandes conjuntos de datos la ejecución de este es bastante lenta. Por lo cual si se genera un conjunto de datos, y luego se detecta un problema, es necesario corregir el programa y volver a ejecutarlo para obtener una nueva versión corregida de los datos. Todo esto conlleva tiempo y llevó varias semanas dentro del proyecto.

0.1.2. Índices únicos

Otro problema encontrado es que no se estaba pudiendo ejecutar la Operación 2 correctamente (esta operación hace JOIN de dos triplas utilizando un comodín en distintas posiciones de la tripla). Luego de analizar el problema fue evidente que cuando se hizo el traductor, quién está encargado de asignar un UID a cada URI y proveer la matriz que es cargada por el prototipo, no se tomó en cuenta que podían existir sujetos que fuesen objetos o viceversa (también potencialmente puedan existir predicados que fuesen sujetos u objetos, pero no se resolvió este problema ya que el modelo de datos utilizado no lo tenía).

Esto causó que se tuviesen distintos UID para la misma URI. Una vez detectado esto, se decidió unir los archivos que almacenan la relación entre UID y URI de sujeto y objeto en uno solo. Esto perjudica el desempeño en parte, ya que si bien se ahorra memoria al cargar la matriz (ya que no hay URIs duplicadas), la búsqueda sobre estos hashes es más laboriosa. Tanto en el prototipo, como también en el traductor (tener en cuenta que la mayor parte del tiempo de uso de CPU durante la ejecución de este se pasa buscando si una URI ya está registrada y dándole un UID nuevo). A pesar de perjudicar el desempeño, parece la forma más lógica y entendible de arreglar el problema. Luego de esto,

la operación 2 funcionó correctamente.

0.1.3. Versiones de Virtuoso

Inicialmente se había instalado Virtuoso desde los repositorios de Ubuntu (los cuales tienen Virtuoso 6.1 disponible), desconociendo la última versión (Virtuoso 7). En este ambiente Virtuoso era considerablemente más lento de lo que resultó ser en los casos de prueba, para tener una idea era más lento que RDF-3X en casi todos los escenarios, salvando claro en las operaciones donde hacía uso de su cache.

Más adelante, Lorena, nos informó que Virtuoso 7 tenía varias mejoras, en especial en lo que respecta a la eficiencia. Por lo cual, se comenzó nuevamente con las pruebas y como se nos había dicho, Virtuoso tuvo una mejora notoria. Por lo que se pudo investigar esto se debe a que Virtuoso cambió la manera en la que maneja los índices generados sobre las columnas, estructura que utiliza para almacenar grafos RDF.

0.1.4. Manejo de memoria en el prototipo

En un comienzo la carga de bases de datos grandes en nuestro prototipo era un proceso largo, esto se debía a que cada vez que se carga desde disco se piden grandes cantidades de memoria. Para realizar esta carga, a nivel de código, primero se genera una estructura vacía, esta se crea con los cabezales de los archivos antes mencionados. Luego, se recorre el archivo y se cargan los elementos necesarios. Un gran tiempo se pasaba asignando nueva memoria, para ingresar estos elementos, por lo cual para solucionar esto, en lugar de usar operadores de pedido de nueva memoria se usó el operador "realloc" de C, el cual tiene un tiempo de ejecución mucho menor, ya que no es un pedido de memoria nueva, sino que el sistema operativo agranda el tamaño de un puntero de memoria ya asignado. Con este cambio se bajó el tiempo de carga de matrices en gran medida, alrededor de un factor de mejora de 50×.

0.1.5. Debugging

Poder analizar los errores con bases de datos tan grandes fue un gran enemigo durante la implementación del prototipo. Esto se debe más que nada al volu-

men de datos que se maneja. Muchas veces ejecuciones con conjuntos de datos pequeños no fallan mientras que cuando se pasa a la misma ejecución pero con una base de datos más grande el prototipo caía debido a errores. Por esta razón, el uso del Debugger tanto como de herramientas de manejo de memoria (Valgrind por ejemplo) es bastante más complejo, sobretodo dado que estas herramientas de por sí tienen un overhead considerable. Además de esto, CUDA hasta hace poco tiempo no era amigable a la hora de usar un Debugger, por lo cual también existe un tema de falta de madurez de la tecnología para poder usar estas herramientas sobre código que se ejecuta en GPU. Nuestras soluciones a estos problemas fueron en general indirectas. Analizar uso de memoria y CPU del proceso de nuestro prototipo. Análisis manual del código, en ocasiones uso de otras técnicas como peer-programming o rubber-duck-debugging. En general estas técnicas indirectas llegaron a buen puerto.

0.1.6. Detección de cuellos de botella y técnicas para acelerar dicho algoritmo

Para detectar cuellos de botella en este programa se utilizaron varias técnicas, entre ellas se utiliza la herramienta VisualVM la cual permite analizar no solo la cantidad de memoria utilizada por un programa, como también el tiempo de CPU utilizado y, en particular, por qué clases y métodos son consumidos estos respectivamente. Otra técnica consiste en probar con varias alternativas empíricamente (por ejemplo a la hora de utilizar bibliotecas para leer archivos de disco o almacenar cadenas de caracteres en memoria) y optar por el más eficiente.

Durante las primeras pruebas del algoritmo la falta de optimización se observaba en los resultados, en su primera iteración el desempeño era medido en decenas de triplas procesadas por segundo, mientras que en la última iteración esta se elevó a miles de triplas procesadas por segundo. A continuación se explican algunas de estas técnicas.

La primera técnica fue la paralelización en hilos, ya que el algoritmo descrito anteriormente es paralelizable. El punto de entrada de estos hilos es una cola, la cual es alimentada por el hilo principal, el cual tiene como responsabilidad leer el archivo de entrada e insertar en esta cola cada tripla por separado. A

partir de esta cola cada hilo secundario puede realizar ejecuciones del algoritmo anteriormente mencionado.

Es importante notar que para evitar colisiones se utilizaron diccionarios capaces de ser accedidos de manera concurrente, y también fue necesario bloquear el acceso concurrente a ciertas otras partes del algoritmo. Una vez finalizada la ejecución de todos los hilos (una vez que la cola queda vacía y el archivo se termina de recorrer), es posible reanudar la ejecución por parte del hilo principal, que continúa como si el programa hubiese sido ejecutado completamente por un solo hilo.

Otra mejora realizada sobre el programa fue la inclusión de arreglos de diccionarios. Como se mencionó en el párrafo anterior, al utilizar hilos había que ser cuidadoso sobre las operaciones y su orden sobre los diccionarios, esto estaba causando muchas demoras ya que mientras un hilo operaba sobre el diccionario podrían haber uno o más esperando turno para hacer la misma tarea. Para esto se creó un arreglo de diccionarios, donde la unión de estos es equivalente al diccionario original. Para decidir sobre cual de estos diccionarios debía operar un hilo particular se utiliza el elemento sobre el cual estaba operando éste para generar un identificador, el cual simplemente selecciona que diccionario utilizar.

Luego de acelerar considerablemente el procesamiento fue necesario optimizar la cantidad de espacio ocupado en memoria por el programa. Esto se realizó de dos maneras, la primera fue sencilla y consistió en reemplazar cadenas de caracteres largas por otras más pequeñas (obviamente siendo estas simbólicas). La mayor parte de las URI de los elementos tienen un prefijo común largo, reemplazando este por unos pocos caracteres que no se puedan confundir se achica considerablemente el tamaño de la variable que lo aloja.

Otra optimización está relacionada con la máquina virtual de JAVA y sus configuraciones, a partir de JAVA 9 se introdujo la posibilidad de almacenar caracteres utilizando distintas codificaciones, previo a esto se utilizaban 2 bytes por cada carácter, mientras que luego se redujo a 1 único byte, logrando casi una mejora de 50 % en nuestro caso (aproximadamente un 60 % si se toma en conjunto con la mejora anterior). Esto no solo es una mejora a la hora de

observar el espacio ocupado en memoria RAM, sino que además se aceleraron considerablemente las comparaciones entre cadenas de caracteres que se ejecuta a la hora de buscar.

0.1.7. Operación 7

Incluido en el plan del prototipo estaba la Operación 7, la cual es una consulta que incluye el operador FILTER. Este funciona para los operadores <, <=, ==, !=, >= y >. Para esto fue necesario también implementar un operador similar al "atoi" encontrado en C, pero que fuese capaz de ejecutarse en GPU. Este operador convierte una cadena de caracteres (que representa un número) en un número entero. Esto es necesario, ya que las URIs vienen únicamente como cadenas de caracteres, y para comparar números es necesario convertir los tipos. Por desgracia, y vinculado al punto anterior, no se pudo hacer que esta operación funcionase correctamente con los conjuntos de datos de prueba generados (si corre con conjuntos de datos creados a mano). Y nuevamente, debido a la imposibilidad de usar herramientas para analizar el problema y que el alcance del proyecto ya estaba logrado en nuestra opinión, se decidió dejar fuera de este.

0.2. Código

A continuación se exponen los fragmentos principales de código del sistema.

0.2.1. CSR

La estructura utilizada contiene los siguientes elementos:

```
int *indicefila // array que indica la cantidad de valores
validos por filas

int sizeIndice // indica el largo de la estructura anterior

int size // cantidad de entradas no nulas

int *columnas // indices de columnas
```

```
int ** datos // array de arrays que almacena las relaciones que se
cumplen para la entrada actual. Ejemplo si la primera linea de la
matriz indica que A se relaciona con B por las propiedades 5 y 6
entonces datos[0] = [2][5][6] donde el primer elemento indica
el tamaño de este array

int * referencias // indica donde se almacenan los valores en la
estructura datos para la entrada i-esima

int * arrayDatos // almacena las relaciones al igual que la
estructura datos pero en este caso es un array continuo

int sizeArrayDatos // tamaño de estructura anterior

int * arrayDatosIndice // indica donde empiezan las propiedades
que se cumplen para el elemento i-esimo
```

0.2.2. Cargar Matriz

```
CSR cargarMatrizCRS(FILE *file, Cabezal cabezal, bool modoDebug) {

  \\inicializa estructuras con la medidas correspondientes
  CSR csr = CSR(cabezal);

  int i, *filasCooAux;
  filasCooAux = (int*)malloc(
  cabezal.getCantidadEntradasNoNulas() * sizeof(int));

  // almaceno los datos y sus coordenadas
  char datos[20];
  char* dato;
  int count = 0; // contador para indice de datos en formato array
  int oldPercentage = 0;

  for (i = 0; i < cabezal.getCantidadEntradasNoNulas(); i++) {
    int newPercentage = i*100/cabezal.getCantidadEntradasNoNulas();
```

```
if( modoDebug && newPercentage != oldPercentage){
  printf("Reading hash file: %d/100\n", newPercentage);
  oldPercentage = newPercentage;
  fflush(stdout);
}

fscanf(file, "%d %d %s\n", &filasCooAux[i],
&csr.getColumnas()[i], &datos);
dato = strtok(datos, ",");
count = 0;

while (dato != NULL) {
  int newDato = atoi(dato);

  // insert in datos
  csr.getDatos()[i] = (int *) realloc(csr.getDatos()[i],
                   (csr.getDatos()[i][0] + 2) * sizeof(int));

  // aumento cantidad de elementos
  csr.getDatos()[i][0] = csr.getDatos()[i][0] + 1;

  csr.getDatos()[i][(csr.getDatos()[i][0])] = newDato;

  // insert in datos en formato array
  count++;
  csr.setSizeArrayDatos(csr.getSizeArrayDatos() + 1);
  if(csr.getArrayDatos() == NULL) {
    csr.setArrayDatos(new int[1]);
  } else {
    csr.increaseArrayDatosSize(csr.getSizeArrayDatos());
  }
  csr.setArrayDato(newDato);

  // leo proximo dato
  dato = strtok(NULL, ",");

}
```

```
    delete dato;

    csr.getArrayDatosIndice()[i + 1] = csr.getArrayDatosIndice()[i]
                                        + count;
    csr.getReferencias()[i] = i + 1;

  }

  printf("\n");

  //inicializo indiceCSR todo en cero
  for (i = 0; i <= cabezal.getFilas(); i++) {
    csr.getIndicefila()[i] = 0;
  }

  // cuento la cantidad de elementos por fila y guardo ese valor en el
  // indice correspondiente
  // el +1 en el indice es porque primer valor en indices es 0
  for (i = 0; i< cabezal.getCantidadEntradasNoNulas(); i++) {
    csr.getIndicefila()[filasCooAux[i] + 1]++;
  }

  // sumo los elementos de las filas anteriores
  for (i = 0; i < cabezal.getFilas(); i++) {
    csr.getIndicefila()[i + 1] += csr.getIndicefila()[i];
  }

  return csr;

}
```

0.2.3. Matriz × Vector

```
__global__ void multiplicarMVCRS(int* vector, int *indicefilaCSR,
    int *columnasCSR, int *referenciasCSR_dev, int filas,
    int * resultado) {
    int indice = blockDim.x * blockIdx.x + threadIdx.x;
```

```
    int inicio, fin;
    int count;
    if (indice < filas) {
        inicio = indicefilaCSR[indice];
        fin = indicefilaCSR[indice + 1];
        count = 0;
        int i;
        for (i = inicio; i < fin; i++) {
            count += referenciasCSR_dev[i] * vector[columnasCSR[i]];
        }
        resultado[indice] = count;
    }
}
```

0.2.4. Vector × Matriz

```
__global__ void multiplicarVMCRS(int* vector, int *indicefilaCSR,
    int *columnasCSR, int *referenciasCSR_dev, int filas,
    int * resultado) {
    int fila = blockDim.x * blockIdx.x + threadIdx.x;
    int inicio, fin;
    if (fila < filas) {
        inicio = indicefilaCSR[fila];
        fin = indicefilaCSR[fila + 1];
        int i;
        for (i = inicio; i < fin; i++) {
         atomicAdd(
             &resultado[columnasCSR[i]],
             vector[fila] * referenciasCSR_dev[i]
         );
        }
    }
}
```

0.2.5. Cumple predicado

```
__global__  void cumplePredicado(
            int* resultadoIntermedio, int sizeResultadoIntermedio,
            int idPredicado, int *arrayDatos,
```

```
            int *arrayDatosIndice, int* resultado) {

    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int inicio, fin;
    if (i < sizeResultadoIntermedio) {
    int entrada = resultadoIntermedio[i];
    if (entrada > 0) {
        inicio = arrayDatosIndice[entrada - 1];
        fin = arrayDatosIndice[entrada];
        for (int j = inicio; j < fin; j++) {
            if (arrayDatos[j] == idPredicado) {
                resultado[i] = 1;
            }
        }
    }
}
```

0.2.6. Operación 1

```
    vector = Vector(sizeFilas);
    vector[indiceSujeto] = 1;
    // se crea un vector de tamaño igual a la cantidad de filas
    // con todos los valores en 0 y se setea en 1 el que corresponde
    // al sujeto por el cual se consulta.

    resultadoIntermedio = Vector(sizeColumnas);
    // se crea un vector que almacenará el resultado de multiplicar
    // el vector anterior por la matriz.

    transferir_y_multiplicar(vector, csr, resultadoIntermedio);
    // se llama a la operación que transfiere los datos a la GPU
    // y realiza la multiplicación obteniendo la fila del sujeto con
    // indiceSujeto.

    resultado = Vector(sizeColumnas);
    // se crea un vector que almacenara el resultado de validar
    // para todos los objetos cual cumple con la propiedad dada.
```

```
cumplePredicado (resultadoIntermedio, indicePredicado, csr,
                resultado)
// se llama a operación que verifica que se cumpla la propiedad
// indicada. Cabe mencionar que esta operación se implementó
// en CPU y GPU, se realizaron mediciones de performance
// resultando mejor la implementación en GPU.

// se imprime resultado
Para cada valor en el vector resultado
    si resultado[i] > 0
        entonces imprimo resultado
```

0.3. Articulo presentado en el XLV Latin American Computing Conference

Aceleración de consultas en bases de datos de grafos mediante el uso de operaciones de Álgebra Lineal Numérica

Bruno Amaral
Instituto de Computación
Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay
bruno.amaral@fing.edu.uy

Juan Manuel San Martin
Instituto de Computación
Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay
juan.san.martin@fing.edu.uy

Lorena Etcheverry
Instituto de Computación
Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay
lorenae@fing.edu.uy

Pablo Ezzatti
Instituto de Computación
Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay
pezzatti@fing.edu.uy

***Resumen*—The application of graph databases to different domains is gaining momentum. The Resource Description Framework (RDF) is one of the data models supported by graph databases, and SPARQL is the standard query language for RDF graphs. These databases are also known as RDF triplestores. Many triplestores are implemented over the relational data model, using tables to store graphs and translating SPARQL queries into SQL queries, and this approach can lead to unnecessary overheads. On the other hand, in the context of High-Performance Computing (HPC), implementations over hybrid hardware platforms using Numerical Linear Algebra (NLA) operations have become an effective and efficient computing strategy in the last decade. In particular, Graphics Processing Units (GPUs) have been adopted to perform general-purpose computations due to their high performance, reasonable prices, and an attractive relationship between computing capacity and energy consumption. In the context described above, this paper presents an initial study on the efficient implementation of a set of SPARQL queries in terms of NLA operations. Additionally, we evaluate the performance of implementing these operations on GPUs.**

***Index Terms*—graph data bases, RDF, SPARQL, Numerical Linear Algebra, GPUs.**

I. Introducción

En los últimos años, el uso de las bases de datos de grafos se ha extendido a diferentes dominios. Acompasando el gran desarrollo de las áreas relacionadas con la gestión de grandes volúmenes de datos (*big data*), estas bases de datos han cobrado una notable importancia a la hora de manejar de manera estructurada datos que pueden modelarse como grafos. Un caso emblemático son los datos provenientes de las redes sociales.

Existen diferentes modelos de datos para grafos, siendo el *Resource Description Framework* (RDF) [12] uno de ellos. SPARQL (*SPARQL Protocol And RDF Query Language*) [6] es el lenguaje de consulta estándar sobre grafos en RDF. Muchos de los sistemas de gestión de datos RDF y los motores de consultas SPARQL existentes utilizan en forma subyacente el modelo relacional para almacenar datos y resolver consultas. Esto implica una transformación entre modelos, la cual no necesariamente redunda en eficiencia. Otros enfoques almacenan los grafos RDF en estructuras que reflejan los conceptos de nodos y aristas, y proponen mecanismos para la resolución de consultas SPARQL basándose en homomorfismos entre grafos.

Por otro lado, el Álgebra Lineal Numérica (ALN) es un área de la computación donde existe un amplio conjunto de herramientas orientadas al cómputo eficiente de operaciones sobre matrices. Se destacan entre ellas las técnicas de computación de alto desempeño (HPC, por su sigla en inglés), y en particular el uso de plataformas de hardware híbridas, las cuales se han posicionado en la última década como una alternativa eficiente para computar operaciones de ALN.

Las plataformas de hardware híbridas equipadas con uno o mas procesadores multicore y aceleradores de hardware, han experimentado una importante evolución. En particular las GPUs (del inglés *Graphics Processing Units*), diseñadas originalmente para acelerar la presentación de gráficos, han sido adoptadas para realizar cómputos de propósito general debido a su gran rendimiento, precios razonables y una atractiva relación entre capacidad de cómputo y consumo energético. Este auge en la utilización de GPUs para la resolución de problemas generales fomentó el desarrollo del área que se conoce como GPGPU (del ingés *General-purpose computing on graphics processing units*) [8].

En el contexto descripto anteriormente, en este trabajo se presenta un estudio inicial sobre la implementación de forma eficiente de un conjunto de consultas SPARQL en términos de operaciones de ALN. Adicionalmente, se evalua el desempeño

de utilizar dichas técnicas en una GPU para acelerar el cómputo de las operaciones. Específicamente, se proponen diferentes representaciones matriciales para los datos en RDF, y se realiza un evaluación experimental preliminar para seleccionar aquella que ofrece mejores resultados en las operaciones de carga y multiplicación (matriz por vector y vector por matriz). Luego, se mapea un subconjunto de consultas SPARQL relevantes a operaciones de ALN sobre la representación matricial elegida, y por último se desarrolla un prototipo basado en GPUs que implementa las ideas antes descriptas.

La estructura del resto del documento es la siguiente: La Sección II presenta conceptos preliminares sobre RDF y SPARQL, así como un análisis sobre motores de almacenamiento y consulta para este tipo de datos. Luego, en la Sección III se presenta por un lado una propuesta para la representación de datos RDF en formatos matriciales y por otro, la implementación de un subconjunto de consultas SPARQL relevantes en términos de operaciones de ALN sobre la representación matricial elegida. La Sección IV presenta los resultados de la evaluación experimental del prototipo implementado sobre GPUs, mientras que la Sección V presenta trabajos relacionados. Por último la Sección VI presenta las conclusiones y trabajos a futuro.

II. Conceptos preliminares

En esta sección se introducen los principales conceptos de RDF y SPARQL, y una breve revisión de los motores de almacenamiento y consultas que los soportan, conocidos como RDF *Triplestores*.

II-A. RDF y SPARQL

RDF El Resource Description Framework (RDF) [9] es un modelo de datos para expresar afirmaciones sobre recursos identificados por un identificador universal (URI del inglés *Universal Resource Identifier*). Las afirmaciones son expresadas como triplas *sujeto - predicado - objeto*, donde el *sujeto* y el *predicado* siempre es un recurso y el *objeto* puede ser un recurso o un string. Los *nodos blancos* son utilizados para representar recursos anónimos o recursos sin URI, que típicamente tienen una función estructural como por ejemplo agrupar un conjunto de afirmaciones. Los valores en RDF son llamados *literales*, y sólo puede aparecer como objeto en las triplas. Un conjunto de triplas RDF puede verse como un grafo dirigido donde los *sujetos* y *objetos* son nodos, y los *predicados* son arcos. Existen varios formatos para la serialización de RDF, en este trabajo utilizamos Turtle [2].

A modo de ejemplo, a continuación presentamos triplas extraídas de la DBPedia que representan el lugar y fecha de nacimiento de Roger Federer, mientras que la Figura 1 muestra una representación gráfica de estas triplas.

```
PREFIX dbr:<http://dbpedia.org/resource/>
PREFIX dbo:<http://dbpedia.org/ontology/>

dbr:Roger_Federer dbo:birthPlace dbr:Basel .
dbr:Roger_Federer dbo:birthDate "1981-08-08" .
```

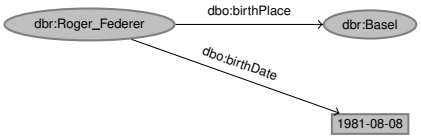

Figura 1: Representación gráfica de datos en RDF.

SPARQL es el lenguaje estándar propuesto por la W3C para consultas sobre RDF [6]. El mecanismo de evaluación de consultas de SPARQL se basa en correspondencia entre subgrafos. Las triplas RDF son interpretadas como vértices y aristas de grafos dirigidos, y el grafo de la consulta se busca en el grafo de datos, instanciando las variables presentes en el grafo de definición de la consulta. Los criterios de selección se expresan como un patrón de grafos en la cláusula `WHERE`, la cual consiste básicamente de un conjunto de patrones básicos (BGPs del inglés *basic graph patterns*) conectados por el operador '.' . Los patrones de grafo son similares a las triplas RDF, excepto que cada sujeto, predicado y objeto puede ser una variable. Las variables se indican anteponiendo el símbolo '?'. Un patrón de grafo básico concuerda con un subgrafo de datos RDF cuando los términos RDF (*RDF terms*) de dicho subgrafo pueden ser sustituidos por las variables y el resultado es un grafo RDF equivalente al subgrafo en cuestión. La consulta a continuación, aplicada al grafo RDF de la Figura 1, retorna el lugar de nacimiento de Roger Federer.

```
1  PREFIX dbr:<http://dbpedia.org/resource/>
2  PREFIX dbo:<http://dbpedia.org/ontology/>
3
4  SELECT ?place
5  WHERE
6  {
7    dbr:Roger_Federer dbo:birthPlace ?place.
8  }
```

II-B. Manejadores de datos en RDF

Existen dos enfoques típicos en el diseño de los sistemas manejadores de datos en RDF, también conocidos como *Triplestores*. Por un lado, los enfoques basados en el modelo relacionalque representan los datos RDF en forma tabular siguiendo diferentes técnicas, y hacen un uso extensivo de índices sobre las tablas para resolver las consultas. En algunos casos, las consultas SPARQL son traducidas a consultas SQL sobre esta representación. Por otro lado, existen enfoques basados en grafos, que buscan representar a los datos RDF como un grafo, y usualmente emplean técnicas basadas en homomorfismos para resolver las consultas SPARQL. Una revisión exhaustiva del estado del arte en estos sistemas puede encontrarse en [11].

En esta sección se presentan las características principales de dos herramientas que corresponden al enfoque relacional (Virtuoso y RDF-3X), y una que corresponde al enfoque orientado a grafos llamada gStore.

II-B1. Virtuoso: Virtuoso[1] es un *middleware* y motor de base de datos híbrido que combina la funcionalidad de un

[1]Virtuoso Open-Source Edition http://vos.openlinksw.com/owiki/wiki/VOS/

sistema de gestión de bases de datos relacionales (RDBMS), base de datos relacional de objetos (ORDBMS), base de datos virtual, RDF, XML, texto libre, servidor de aplicaciones web y servidor de archivos en un solo sistema. Este producto ha sido desarrollado por OpenLink Software, y en sus orígenes era un RDBMS orientado a transacciones de filas utilizando *clusters* de servidores, el cual fue sucesivamente reorientado como una base de grafos RDF que admite consultas SPARQL y soporta ciertos niveles de inferencia (*entailment regimes*). En sus versiones más recientes este producto soporta almacenamiento comprimido por columnas y ejecución vectorizada [4]. El almacenamiento por columnas generalmente implica un motor de ejecución vectorizada que realiza operaciones de consulta en un gran número de tuplas a la vez, ya que la latencia para el acceso a las tuplas es mayor que con un almacenamiento por filas. Asimismo, la ejecución vectorizada puede mejorar también el rendimiento del almacenamiento por filas. También se destaca que la eliminación de la sobrecarga de interpretación, así como las mejoras en la ubicación de la memoria caché, redundan en ventajas tanto en el almacenamiento columnar como por filas.

II-B2. RDF-3X: RDF-3X [10] es un motor SPARQL que sigue una arquitectura de estilo RISC, utilizando intensivamente mecanismos de indexación y mejoras en el procesamiento de consultas. RDF-3X explota la lógica de los sistemas de gestión de datos diseñados y personalizados para los dominios de aplicación específicos, buscando así superar el desempeño de los sistemas genéricos.

Esta propuesta se basa en los siguientes tres principios claves:

- Independencia del diseño físico de la carga de trabajo: para lograr esto se construyen índices sobre las seis permutaciones de los tres items que constituyen una tripla RDF y, adicionalmente se cuenta con índices sobre agrupaciones de estos items (por ejemplo, índice por sujeto-objeto que permite responder eficientemente a consultas que poseen patrones $< s, p, ?o >$). Dado que cada uno de estos índices se puede comprimir, el espacio para almacenar todos los índices es menor que el tamaño de los datos.
- El procesador de consultas es de estilo RISC, y se basa fuertemente en ejecutar *merge joins* sobre listas de índices ordenados. Esto es posible gracias a la indexación exhaustiva de la tabla de triplas. Asimismo, los planes de ejecución de consultas se optimizan y se reordenan operaciones de forma de aprovechar al máximo los índices para realizar las operaciones de *join*. El motor se encuentra altamente optimizado a nivel de código para realizar este tipo de consultas, pero es lo suficientemente versátil como para soportar las características de SPARQL, entre las que se destacan la eliminación de duplicados y el soporte a patrones de consultas que incluyen disyunción.
- El optimizador de consultas se centra en optimizar el orden de las operaciones de *join* en los planes de ejecución. Para esto se usa programación dinámica, con un modelo de costos basado en estadísticas específicas de RDF que incluyen contadores de predicado-secuencias frecuentes en trayectorias del grafo de datos ya que estos caminos son potenciales patrones.

Este enfoque presenta ciertas debilidades a la hora de insertar tuplas, dado que esto implica la actualización de los diferentes índices de la herramienta. Para mitigar esto las actualizaciones se recopilan en espacios de trabajo aislados, que cuentan con sus propios índices diferentes. Éstos son fusionados con los índices principales de la base de datos mediante procesos batch. Por último, cabe señalar que RDF-3X no proporciona transacciones ACID completas, pero incluye formas de control de la concurrencia y soporta el nivel *read-committed*.

II-B3. gStore: gStore [14] es un motor de gestión de datos RDF y procesamiento de consultas SPARQL basado en grafos. El almacenamiento se realiza como una tabla de lista de adyacencias, donde a cada entidad o vértice en el grafo, de acuerdo con las etiquetas de sus adyacentes, se le asigna una cadena de bits como la firma del vértice. Esta representación transforma al grafo RDF en un grafo de firma de datos G*. Sobre G* se construye un índice, llamado *VS*-tree* que es utilizado para la resolución de las consultas.

Para el procesamiento de una consulta Q se genera en tiempo de ejecución una representación de la misma en términos de firmas de vértices (llamada Q*) y luego se buscan las coincidencias de Q* sobre G*. Ese método permite responder consultas SPARQL exactas y con variables de una manera uniforme. Para acelerar la búsqueda dentro de G* se implementan reglas de poda y algoritmos de búsqueda que tratan de aumentar la eficiencia. Por último, esta propuesta presenta un algoritmo de mantenimiento para manejar actualizaciones en línea sobre repositorios RDF.

III. Propuesta

En esta sección se presenta la propuesta específica. Se comienza presentando la representación lógica de los datos en forma matricial en la Sección III-A. Luego, en la Sección III-B se reseña el subconjunto de consultas SPARQL que se abordan en este trabajo, asi como su implementación mediante operaciones de ALN sobre la representación matricial. Por último, en la Sección III-C, se discuten y evalúan posibles representaciones físicas de estas matrices.

III-A. Representación lógica de grafos RDF

Como ya se mencionó, existe una relación natural entre los grafos y su representación mediante matrices. Para representar un grafo RDF, la representación más directa consiste en una matriz de $S \times O$ siendo S la cantidad de Sujetos y O la cantidad de Objetos, donde cada entrada es una lista de los Predicados que conectan un determinado sujeto con un objeto (notar que puede haber más de un arco entre un sujeto y un objeto porque en realidad en RDF es posible representar hipergrafos). En esta representación se le asigna a cada sujeto y objeto un identificador que se corresponderá a una columna de la matriz. Una posible asignación para las triplas del Listado II-A es $dbr : Roger\_Federer = 1$,

$dbr : Basel = 2$ y $'1981-08-08' = 3$ (llamaremos *hashObj* a la tabla que almacena este mapeo). También se le asigna un identificador a cada predicado, por ejemplo: $dbo : birthPlace = 1$ y $dbo : birthDate = 2$ El Cuadro I muestra la representación lógica de las triplas de Listado II-A considerando estos mapeos.

	Roger_Federer	Basel	1981-08-08
Roger_Federer		1	
Basel			2

Cuadro I: Ejemplo de representación lógica

Para resolver sobre esta representación la consulta que devuelve el lugar de nacimiento de Roger Federer (presentada en la Sección II-A), es preciso realizar las siguientes operaciones:

- Como la consulta fija el sujeto, se sabe que el resultado está contenido en la fila correspondiente al sujeto. Esa fila se obtiene con la operación $v \times A$ siendo el vector $v = (1, 0)$ y A la matriz ya descripta.
- El resultado de la operación anterior será w = (_,1,_) dicho resultado tendrá que ser recorrido por entrada buscando la propiedad buscada.
  A continuación se presenta un pseudocódigo del procedimiento de búsqueda.

```
for i = 1 to O
while w[i].HasNext;
predicado = w[i].next;
if (predicado == 1) then
result.add(i)
fin while
fin for
```

El resultado es 2, buscando este índice en la tabla *hashObj* se obtiene que la ciudad de nacimiento de Federer es Basel.

Otra opción es una representación no tradicional, en donde se tiene una Matriz multidimensional o Tensor. En esta representación cada nivel está asociado a un predicado. Al igual que en la representación tradicional se le asigna a cada sujeto y objeto un identificador que se corresponderá a una columna de las matrices y, también se asigna un identificador a cada predicado pero este definirá un nivel de la matriz tridimensional. La existencia de una arista entre un sujeto y objeto será dada por el valor 1 en el nivel correspondiente al predicado que los conecta.

Notar que el enfoque tradicional permite resolver con menor cantidad de operaciones aquellos patrones de consulta que incluyen una variable en la posición del predicado, mientras que con el enfoque multidimensional es necesario una multiplicación de matriz por nivel de cada operando.

III-B. Consultas objetivo y su implementación en ALN

Dada la amplia expresividad del lenguaje de consultas SPARQL, en este trabajo se decidió abordar un subconjunto del mismo priorizando aquellas consultas que son más frecuentes en la práctica. Para esto se toma como base el trabajo de Arias et al. (2011) [1], donse se presenta un análisis empírico de consultas SPARQL. Este análisis considera registros de consultas realizadas sobre DBPedia[2] y SWDF[3] que forman parte del conjunto de datos ofrecido en el marco de la competencia USEWOD2011 Challenge [3]. Estos registros recopilan cerca de 3 millones de consultas, que de acuerdo a los autores son suficientemente heterogéneas e incluyen consultas generadas tanto por humanos como por computadoras. A continuación se resumen las principales conclusiones de dicho trabajo.

La categorización por tipo de consulta SPARQL (`SELECT`, `DESCRIBE`, `CONSTRUCT` y `ASK`) concluye que las consultas del tipo `SELECT` son la amplia mayoría en ambas fuentes (96.9 % y 99.7 % sobre DBPedia y SWDF respectivamente.).

El análisis sobre los funciones del lenguaje que son utilizadas arroja que las cláusulas de tipo `FILTER` son las más comunes, apareciendo en un 49 % de las consultas en ambas fuentes. El análisis de los operadores de comparación indica que el operador de igualdad es el más utilizado, apareciendo en 23 % de las consultas. Los autores concluyen también que el resto de las funciones de SPARQL no son demasiado utilizadas, a excepción de la cláusula de `UNION`, que está presente en aproximadamente el 10 % de las consultas.

Este estudio además incluye un análisis del tipo de patrones que aparecen en las consultas. Los siguientes valores son los tres más comunes:

- $< s, p, ?o >$ (sujeto y predicado constante, objeto variable): 66 % en DBPedia y 48 % en SWDF.
- $< s, ?p, ?o >$ (sujeto constante, predicado y objeto variables): 22 % en DBPedia y 0.52 % en SWDF.
- $<?s, p, o >$ (sujeto variable, predicado y objeto constantes): 7 % en DBPedia y 46 % en SWDF.

El segundo y tercer puesto presentan grandes diferencias según la fuente de datos considerada, y los autores suponen que esto se debe a la naturaleza semántica de estas bases de datos. Se destaca la utilidad de estos resultados a la hora de diseñar índices para acelerar las consultas (en estos casos, Sujeto-Predicado, Sujeto y Predicado-Objeto serían los más recomendables).

En cuanto a cómo estos patrones detectados se combinan entre ellos, el resultado más común es el de un único patrón por consulta (97 % en SWDF y 66 % en DBPedia). Para el caso de DBPedia se analizan las consultas con dos tuplas, las cuales implementan operaciones tipo JOIN con una variable común. En este caso se observó que el JOIN más común es el Sujeto-Sujeto (60 %), seguido luego por Sujeto-Objeto (34,5 %), y el tercer lugar por Objeto-Objeto (4,5 %).

Este trabajo analiza también la topología del grafo subyacente, en particular intenta identificar si corresponde a patrones tipo estrella o si incluye cadenas largas. Para esto se exploran los grafos resultados de las consultas, en particular, el camino

[2]https://wiki.dbpedia.org/

[3]http://data.semanticweb.org

más largo que estos presentan, encontrando que el 98 % de ellos tienen largo máximo de 1. Por otro lado, se reporta que en SWDF la gran mayoría de los grafos resultados son de tipo nodo central con una hoja. En el caso de DBPedia, si bien este último tipo de grafo es el más usual (66 % de ocurrencias), es seguido por grafos con un nodo central y 3 hojas con el 27 % de las ocurrencias. Los otros tipos de grafos encontrados ocurren con una frecuencia menor al %10, y la mayor parte de ellos también es de tipo estrella. Los autores concluyen que si bien depende de la base de datos, aproximadamente un 33 % de los grafos resultados son de tipo estrella.

III-B1. Consultas objetivo: Se decidió concentrarse en las seis consultas más comunes, de acuerdo con los resultados de [1]. Se consideran entonces consultas con uno y dos parámetros, y se incluye una consulta de tipo *join* con el objetivo de observar su comportamiento en la representación matricial utilizando GPU. A continuación, se presenta el conjunto de consultas objetivo del presente trabajo y su implementación mediante operaciones de ALN.

Para la especificación de operaciones en ALN se utilizan las siguientes definiciones:

- M matriz que modela grafo.
- $v(s) = v(0,..,0,1,0,...,0)$ donde la posición del 1 corresponde al índice que le corresponde al sujeto s en la matriz M (análogo para los objetos).
- n cantidad de objetos en el grafo.
- $objects$ vector que contiene todos los elementos que son objeto en alguna tripla.
- $subjects$ vector que contiene todos los elementos que son sujeto en alguna tripla.
- $predicates$ vector que contiene todos los elementos que son predicado en alguna tripla.

Consulta 1 Sujeto y predicado constantes

```
SELECT ?o
WHERE { s p ?o }
```

La Consulta 1 implementa el patrón más común de consulta reseñado en [1]. Su implementación sobre una representación matricial del grafo implica seleccionar la fila correspondiente al sujeto especificado, para luego buscar en la misma el predicado en cuestión. Su traducción a ALN consiste en multiplicar por el vector que tiene 1 en la fila correspondiente al sujeto en cuestión y luego filtrar. Las celdas que queden con al menos un predicado indican la columna de el o los objetos que deben ser devueltos. A continuación se presenta un pseudocódigo.

Pseudocódigo 1 Consulta 1 en ALN

Require: M es la matriz que representa al grafo, s y p son el sujeto y predicado que se buscan
Ensure: o es un conjunto de objetos que satisfacen el patrón
1: $w \leftarrow v(s) \times M$
2: **for all** $i \in 1..n$ **do**
3: **if** $p \in w[i]$ **then**
4: $o \leftarrow o \cup objects[i]$
5: **end if**
6: **end for**

Consulta 2 Sujeto y predicado constantes mas JOIN

```
SELECT ?o2
WHERE { s p1 ?o1.
        ?o1 p2 ?o2
      }
```

En la Consulta 2 se busca probar que tan eficiente puede ser el prototipo para componer más operaciones. Se agrega al patrón de la Consulta 1 una segunda tripla que exige que los valores a devolver están a distancia 2 del sujeto, considerando sólo los arcos p1 y p2. Su implementación en ALN consiste en realizar lo mismo que para la consulta 1, y una vez obtenidos los objetos correspondientes a la variable ?o1 estos son ingresados como parámetro de entrada nuevamente. A continuación se presenta un pseudocódigo.

Pseudocódigo 2 Consulta 2 en ALN

Require: M es la matriz que representa al grafo, s, $p1$ y $p2$ son el sujeto y los dos predicados parámetros
Ensure: o es un conjunto de objetos que satisfacen ambos patrones
1: $w \leftarrow v(s) \times M$
2: **for all** $i \in 1..n$ **do**
3: **if** $p \in w[i]$ **then**
4: $\mu \leftarrow v(i) \times M$
5: **for all** $j \in 1..n$ **do**
6: **if** $p \in \mu[j]$ **then**
7: $o \leftarrow o \cup objects[j]$
8: **end if**
9: **end for**
10: **end if**
11: **end for**

Consulta 3 Sujeto y objeto constantes

```
SELECT ?p
WHERE { s ?p o }
```

Las Consultas 3 y 4 consideran patrones que no son muy frecuentes, pero se agregan para completar los tres casos en que se fijan dos elementos. El caso de la Consulta 3 es interesante además porque la variable es la tercer dimensión de nuestra matriz. Esto implica que primero se debe fijar una fila y una columna, para ello se multiplica la matriz por un vector seleccionando el sujeto. El vector que se obtiene es multiplicado por otro vector para seleccionar el objeto, lo que produce una única celda que contiene el/los predicados que cumplen la consulta. A continuación se presenta un pseudocódigo.

Pseudocódigo 3 Consulta 3 en ALN

Require: M es la matriz que representa al grafo, s, y o son el sujeto y objeto parámetros
Ensure: p es un conjunto de predicados que satisfacen el patrón
1: $w \leftarrow v(s) \times M$
2: $i \leftarrow w \times v(o)$
3: **for all** $p_i \in properties[i]$ **do**
4: $p \leftarrow p \cup p_i$
5: **end for**

Consulta 4 Predicado y objeto constantes

```
SELECT ?s
WHERE { ?s p o }
```

Esta consulta es análoga a la Consulta 1, pero en lugar de multiplicar para seleccionar un sujeto, se multiplica para

seleccionar un objeto. Luego de eso, se validan los predicados de igual manera que se hace en la Consulta 1. A continuación se presenta un pseudocódigo.

Pseudocódigo 4 Consulta 4 en ALN

Require: M es la matriz que representa al grafo, p y o son el predicado y objeto que se buscan
Ensure: s es un conjunto de sujetos que satisfacen el patrón
1: $w \leftarrow v(o) \times M$
2: **for all** $i \in 1..n$ **do**
3: **if** $p \in w[i]$ **then**
4: $s \leftarrow s \cup subjects[i]$
5: **end if**
6: **end for**

A continuación se presentan dos consultas de dos variables.

Consulta 5 Sujeto constante

```
1  SELECT ?p ?o
2  WHERE { s ?p ?o }
```

A la hora de analizar la Consulta 5 de manera algebraica basta pensar que lo que se require son todas las triplas que tienen un sujeto fijo, por lo tanto se debe multiplicar la matriz por un vector que seleccione el sujeto provisto. El vector resultante contendrá todas las parejas (predicado, objeto) que se deben retornar. A continuación se presenta un pseudocódigo.

Pseudocódigo 5 Consulta5 en ALN

Require: M es la matriz que representa al grafo, s es el sujeto que se busca
Ensure: w representa parejas (predicado, objeto)
1: $w \leftarrow v(s) \times M$

Consulta 6 Objeto constante

```
1  SELECT ?s ?p
2  WHERE { s? ?p o }
```

Por último la Consulta 6 busca fijar el objeto. La resolución de la misma es análoga a la Consulta 5, donde en este caso se multiplica un vector que selecciona el objeto por la matriz, obteniendo un vector que nuevamente tiene todas las parejas (sujeto, predicado) que deben ser devueltas. A continuación se presenta un pseudocódigo.

Pseudocódigo 6 Consulta 6 en ALN

Require: M es la matriz que representa al grafo, o es el objeto que se busca
Ensure: w representa parejas (sujeto, predicado)
1: $w \leftarrow M \times v(o)$

III-C. Representación física de grafos RDF

Típicamente, los conjuntos de datos en RDF llevados a las representaciones antes descriptas, se traducen en matrices dispersas y de grandes dimensiones. En este contexto, mapear a las matrices de forma directa (matrices completas) no es una buena opción, ya que se desperdicia memoria en datos que no son relevantes (valores nulos) y las operaciones implican grandes volúmenes de trabajo.

Se consideran entonces tres alternativas de formatos de almacenamiento dispersos, en concreto: *Compressed Storage Row* (CSR), *Compressed Diagonal Storage* (CDS) e *Hybrid* (HYB) [5].

Formato CSR: En este formato se tienen tres vectores. Uno almacena los valores de las entradas no nulas (vector-datos, donde las entradas nulas son las de valor 0) es de tamaño N, siendo N la cantidad de elementos no nulos de la matriz. Otro vector (vector-columnas) que guarda los índices de las columnas también de tamaño N y el tercero (vector-índices) almacena los índices de valores válidos por filas dejando el primer valor en 0. Este último vector es de tamaño $k + 1$, siendo k la cantidad de filas de la matriz.

Formato CDS: En este formato se tiene una matriz compuesta por las diagonales válidas en donde la primer entrada de cada fila representa el índice de la diagonal a la que corresponde esa fila.

Formato HYB: Como su nombre lo indica esta estrategia de almacenamiento es una mezcla de dos formatos, por un lado el diagonal presentado en la sección anterior y, por otro, el denominado simple o elemental. En este último se utilizan tres vectores: uno contienen los valores no nulos, otro las columnas de cada valor no nulo y un tercero que indica la fila de cada valor no nulo.

En el formato CDS si una diagonal tiene un solo valor válido, por ejemplo la diagonal -2, es necesario guardar toda la diagonal, mientras que en el formato HYB las diagonales con pocos valores válidos son almacenadas en el formato elemental, ahorrando espacio. En el formato HYB es necesario definir un criterio que determine qué diagonal almacenar con formato CDS y cuál con formato elemental. Por ejemplo, se puede utilizar como criterio que la diagonal tiene que tener más de la mitad de sus entradas válidas para que se almacenen en formato diagonal.

Evaluación experimental de la representación física: para elegir una de las representaciones físicas propuestas se mide y compara el desempeño de las mismas en tres operaciones fundamentales: cargar la matriz desde un archivo, multiplicar la matriz por un vector y multiplicar un vector por la matriz. La plataforma de hardware utilizada para esta etapa se describe en el Apartado IV-A. Las Tablas II, III y IV resumen los tiempos de ejecución para las operaciones antes mencionadas. Los tiempos reportados son el promedio de cinco ejecuciones independientes.

Entradas válidas	CSR	CDS	HYB
6	0,000072	0,000077	0,000086
35	0,000117	0,000126	0,000128
88627	0,426136	0,644117	0,447438
412148	0,512749	0,640551	0,601455
1891669	2,386610	-	2,799282

Cuadro II: Tiempos de ejecución (en seg.) cargar matriz.

Como se observa en la Tabla II con una cantidad de entradas muy pequeña los tiempos son muy similares, pero a medida que la cantidad de entradas aumenta se empieza a notar una diferencia en favor de los formatos CSR e HYB. Con una

cantidad de entradas del orden de 1,9 millones, el formato CSR obtiene una performance un poco mejor que HYB (el tiempo del formato CDS no figura dado que excede los recursos de memoria de la plataforma utilizada).

Entradas válidas	CSR	CDS	HYB
6	0,073590	0,074808	0,079638
35	0,085577	0,088232	0,081770
88627	0,083897	0,408101	0,095506
412148	0,080070	0,324645	0,096726
1891669	0,101785	-	0,149929

Cuadro III: Tiempos de ejecución (en seg.) de la operación $Matriz \times Vector$.

En la Tabla III se puede observar que con 6 o 35 entradas los tiempos son casi idénticos pero a partir de 88627 entradas el formato CDS demora 5 veces más mientras que CSR no ve afectado su tiempo e HYB aumenta sólo en forma marginal. La diferencias entre CSR e HYB son despreciables salvo en la última entrada de la tabla donde HYB demora $50\,\%$ más aproximadamente.

Entradas validas	CSR	CDS	HYB
6	0,081819	0,076892	0,081486
35	0,079974	0,080255	0,075778
88627	0,085115	0,365529	0,093926
412148	0,119318	0,336381	0,104435
1891669	0,095897	-	0,142981

Cuadro IV: Tiempos de ejecución (en seg.) de la operación $Vector \times Matriz$.

En la Tabla IV se repite la misma situación que en los estudios anteriores. El formato CSR resulta ser el de mejor desempeño, seguido por el formato HYB. Una vez observados estos resultados también se puede hacer la presunción de que el buen desempeño del formato HYB se debe a que muchos de sus elementos se almacenaron en formato elemental, ya que si se hubieran encontrado en el formato CDS los tiempos serían más similares a los obtenidos para esa estructura.

Dados el análisis efectuado y los resultados obtenidos se utiliza la representación CSR para llevar a cabo el prototipo.

IV. Evaluación experimental

En esta sección se resume la evaluación experimental realizada para validar nuestra propuesta Primero se detallan las plataformas de hardware y los casos de estudios utilizados y, luego, se presentan los resultados experimentales y su análisis.

IV-A. Hardware utilizado

Para generar los datos se utilizó un hardware con las siguientes especificaciones:

- SO: Ubuntu 16.0
- Procesador: Intel® Core™ i7-6700 CPU @ 3.40GHz
- RAM: 64 GB DDR4

Para ejecutar las consultas de prueba se utilizó el siguiente hardware:

- SO: Windows 10
- Procesador: Intel® Core™ i7-8550U CPU @ 1.80GHz
- RAM: 12 GB DDR4
- Tarjeta grafica: NVIDIA® GeForce® MX150 2GB VRAM

IV-B. Casos de prueba

A modo de caso de prueba se ejecutaron las seis consultas presentadas en la Sección III-B1. Los datos utilizados fueron generados con las herramientas provistas por el *Berlin SPARQL Benchmark* (BSBM)[4]. Esta herramienta genera datos sintéticos sobre Productos, Vendedores, Personas y Opiniones, permitiendo indicar el tamaño del conjunto a generar en función de la cantidad de Productos. Para las pruebas se generaron los siguientes conjuntos:

1. 100.000 Productos, 32 millones triplas aprox.
2. 150.000 Productos, 49 millones triplas aprox.
3. 200.000 Productos, 65 millones triplas aprox.
4. 250.000 Productos, 98 millones triplas aprox.

Para cada una de las consultas, se sortearon los parámetros a utilizar y se ejecutaron las consultas sobre RDF-3X, Virtuoso y el prototipo propuesto. Específicamente, se tomaron nodos aleatorios en cada conjunto de datos BSBM, de manera de observar si existen diferencias según el índice generado por el traductor de URIs a enteros. Dado el conjunto de datos de prueba se puede observar que hay operaciones en las cuales predominan los resultados con sub-grafos grandes mientras en otras se obtienen sub-grafos de dos nodos y una arista.

IV-C. Resultados experimentales

Las Tablas V-X resumen los resultados experimentales alcanzados para las seis consultas planteadas anteriormente utilizando los cuatro conjuntos de datos. En las tablas se incluye la siguiente información:

- $\#Prods$: Cantidad de productos que contiene el conjunto de datos.
- Ej: Ordinal de la ejecución (1 a 4).
- $\#Res$: Número de triplas resultado de la consulta específica.
- $TProt$: Tiempo en segundos del prototipo para resolver la consulta.
- $TRDF3X$: Tiempo en segundos de RDF-3X para resolver la consulta.
- $TVirt$: Tiempo en segundos de Virtuoso 7 para resolver la consulta.
- $TProdSImpr$: Tiempo en segundos del prototipo para resolver la consulta, sin contar el tiempo de impresión en pantalla del resultado.

De los resultados experimentales obtenidos se puede observar que en las consultas donde los resultados son de dimensión acotada, decena o centena de triplas, el prototipo no aporta

[4]Berlin SPARQL Benchmark http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/

#Prods	Ej	#Res	TProt	TRDF3X	TVirt	TProtSImpr
100.000	1	1	0,673	0,105	0,016	0,653
	2	19	0,733	0,141	0,016	0,668
	3	1	0,673	0,109	0,016	0,653
	4	1	0,667	0,094	0,016	0,657
150.000	1	1	0,985	0,129	0,032	0,967
	2	1	0,996	0,140	0,015	0,964
	3	1	0,992	0,252	0,063	0,962
	4	1	0,998	0,271	0,016	0,963
200.000	1	22	1,320	0,189	0,582	1,273
	2	19	1,306	0,180	0,109	1,270
	3	1	1,312	0,183	0,891	1,281
	4	1	1,309	0,190	0,563	1,271
250.000	1	1	1,635	0,223	1,281	1,564
	2	1	1,627	0,323	0,047	1,572
	3	1	1,615	0,235	0,500	1,562
	4	1	1,634	0,234	0,312	1,560

Cuadro V: Tiempos de ejecución (en seg.) de la Consulta 1.

#Prods	Ej	#Res	TProt	TRDF3X	TVirt	TProtSImpr
100.000	1	1	1,450	0,940	0,015	1,299
	2	1	2,047	0,125	0,016	1,299
	3	1	1,357	0,109	0,015	1,316
	4	1	1,423	0,109	0,016	1,316
150.000	1	21	1,984	0,244	0,328	1,922
	2	17	2,02	0,149	0,031	1,922
	3	1	2,023	0,472	0,500	1,949
	4	24	2,013	0,453	0,079	1,921
200.000	1	19	2,611	1,861	0,703	2,563
	2	13	2,605	1,718	0,156	2,531
	3	14	2,634	0,280	0,062	2,542
	4	2	32,152	0,500	2,535	31,187
250.000	1	1	3,272	0,315	0,187	3,153
	2	1	3,530	0,250	0,317	3,122
	3	1	3,601	0,535	0,640	3,124
	4	1	3,906	0,231	0,266	3,176

Cuadro VI: Tiempos de ejecución (en seg.) de la Consulta 2.

#Prods	Ej	#Res	TProt	TRDF3X	TVirt	TProtSImpr
100,000	1	1	0,467	0,109	0,016	0,429
	2	2	0,430	0,125	0,016	0,434
	3	3	0,440	0,094	0,016	0,430
	4	4	0,440	0,109	0,016	0,427
150.000	1	3	0,647	0,451	0,047	0,645
	2	1	0,628	0,423	0,047	0,625
	3	2	0,681	0,476	0,047	0,635
	4	1	0,873	0,481	0,093	0,633
200.000	1	3	1,231	0,277	0,172	0,833
	2	1	1,235	0,185	0,078	0,837
	3	2	1,068	0,452	0,078	0,831
	4	2	0,836	0,183	0,125	0,826
250.000	1	1	1,040	0,260	0,203	1,026
	2	2	1,021	0,250	0,185	1,019
	3	3	1,092	0,220	0,156	1,020
	4	4	1,034	0,236	0,188	1,030

Cuadro VII: Tiempos de ejecución (en seg.) de la Consulta 3.

#Prods	Ej	#Res	TProt	TRDF3X	TVirt	TProtSImpr
100.000	1	2.429	0,747	0,266	0,141	0,629
	2	2.649	0,730	0,293	NA?	0,633
	3	508.117	16,307	32,818	34,000	0,633
	4	1	0,670	0,108	0,015	0,626
150.000	1	1.500.000	40,559	103,691	117,375	0,988
	2	12.224	1,342	0,971	0,938	0,962
	3	87.268	4,279	6,573	9,047	0,926
	4	4.038	3,723	Error	1,156	0,929
200.000	1	6.970	4,900	8,571	16,890	1,229
	2	2.183	1,331	0,350	0,562	1,226
	3	100	1,269	0,181	0,250	1,236
	4	60.843	3,439	4,270	8,172	1,223
250.000	1	1917	1,892	2,818	1,218	1,539
	2	2.500.000	89.208	194,418	184,172	1,985
	3	51.518	1,567	5,917	9,766	1,533
	4	1	1,576	0,195	0,313	1,545

Cuadro VIII: Tiempos de ejecución (en seg.) de la Consulta 4.

ningún beneficio. De hecho, en las Tablas V, VI y VII los tiempos de ejecución del prototipo son superiores a los de RDF-3X y Virtuoso para todos los conjuntos de datos. Si bien en estos casos los resultados en desempeño del prototipo son inferiores a los de las herramientas de referencia, una característica positiva de los resultados es que a medida que crecen los conjuntos de datos, el tiempo de ejecución de RDF-3X y Virtuoso en general crece en mayor medida que los tiempos de ejecución del prototipo propuesto.

Por otro lado, cuando las consultas arrojan resultados de gran dimensión, situación que se encuentra principalmente en las Consultas 4 y 6, el prototipo propuesto permite reducir los tiempos de ejecución en varios casos. Así se puede constatar para el conjunto de datos 2 (150K productos) en la Ejecución 1 de la Consulta 4 y en la Ejecución 3 de la Consulta 6, donde el prototipo ofrece interesantes reducciones en los tiempos de ejecución cuando se lo compara con las herramientas de referencia.

Otra particularidad que se desprende de los resultados obtenidos es que, en los casos de consultas con resultados de grandes dimensiones el tiempo de ejecución de la propuesta está completamente dominada por el tiempo de despliegue en pantalla. Notar que los tiempos de ejecución, sin incluir los tiempos de despliegue (ver columna $TProtSImpr$) son completamente acotados. Esta situación está alineada con lo esperable por dos razones. Primero, en la propuesta no se ha trabajado en la optimización de las tareas de despliegue a pantalla. Segundo, los tiempos de transferencia entre la memoria de CPU y GPU pueden ser un cuello de botella en el tipo de hardware utilizado.

Considerando los resultados obtenidos y lo expuesto en los párrafos anteriores, parece razonable utilizar las técnicas planteadas en la propuestas en contextos donde sea común las consultas con grandes volúmenes de datos como resultado, o se encadenen varias consultas donde los resultados intermedios de una consulta son la entrada de otra y/o se tenga acceso a hardware masivamente paralelo con conexiones entre la memoria de la CPU y la de la GPU sea de alta velocidad (por ejemplo NVLink). Notar que las dos primeras consideraciones son contextos comunes en el área de trabajo Base de Datos de Grafos, mientras que la tercera es una opción de hardware accesible para un centro de cómputo, incluso acotado.

Por último, es importante destacar que la evaluación experimental se realizó sobre una GPU de bajo porte. En especial, una tarjeta optimizada para computadores portátiles que ofrece un pico de desempeño de 1,127 GFlops y un ancho de banda de memoria de 48.0 GB/s. Estas especificaciones están en el entorno de 20 veces menos potente que una GPU de alto desempeño, por ejemplo una GPU NVIDIA V100.

V. Trabajo relacionado

Diferentes autores han avanzando en el uso de técnicas de ALN para resolver consultas en Bases de Datos de grafos.

#Prods	Ej	#Res	TProt	TRDF3X	TVirt	TProtSImpr
100.000	1	9	0.397	0,115	0,016	0,359
	2	39	0.400	0,125	0,015	0,365
	3	35	0.397	0,118	0,015	0,361
	4	6	0.393	0,125	0,016	0,356
150.000	1	10	0,581	1,772	0,265	0,534
	2	12	0,593	1,712	0,140	0,538
	3	6	0,585	0,142	0,031	0,540
	4	5	0,584	0,144	0,047	0,543
200.000	1	33	1,062	0,185	0,281	0,729
	2	9	0,772	0,174	0,375	0,708
	3	9	0,764	0,190	0,282	0,717
	4	11	0,797	0,186	0,156	0,708
250.000	1	40	0,958	0,224	0,266	0,898
	2	39	0,988	0,223	0,203	0,895
	3	34	0,981	0,257	0,234	0,895
	4	6	0,958	0,232	0,141	0,893

Cuadro IX: Tiempos de ejecución (en seg.) de Consulta 5.

#Prods	Ej	#Res	TProt	TRDF3X	TVirt	TProtSImpr
100.000	1	10	0,373	0,114	0,015	0,344
	2	355	0,413	0,138	0,047	0,347
	3	1	0,383	0,112	0,015	0,337
	4	2066	0,610	0,412	0,375	0,346
150.000	1	1.500.000	159,787	118,831	172,641	0,510
	2	11.182	1,447	0,967	1,422	0,507
	3	859.166	94,693	65,481	108,313	0,514
	4	76.848	6,911	7,301	9,469	0,510
200.000	1	63.745	6,455	5,049	20,844	0,691
	2	42	0,761	0,181	0,062	0,689
	3	1,672	0,988	0,403	0,422	0,677
	4	4.000.000	385,350	312,986	472,016	0,674
250.000	1	1.917	1,207	0,542	0,485	0,842
	2	2.500.000	233,428	235,820	276,828	0,839
	3	51,518	7,080	5,020	21,187	0,841
	4	1	0,890	0,193	0,094	0,845

Cuadro X: Tiempos de ejecución (en seg.) de Consulta 6

Un aspecto importante y diferencial de estos esfuerzos es la representación que se utiliza para el almacenamiento de los datos. Notar que utilizar diferentes estructuras de datos para representar el grafo implica la necesidad de distintas operaciones del álgebra para computar las consultas. Algunos ejemplos destacados de esta situación son los trabajos de F. Jamour et al. [7] y de X. Zhanget al. [13].

En el primer caso, los autores trabajan con grafos que admiten un único predicado por relación. Bajo esta hipótesis el grafo se puede representar con una única matriz dispersa cuadrada, donde las filas y columnas están dadas por la unión de los objetos y los sujetos. Un valors k distinto de cero en la posición (i, j) de la matriz sigifica que existe un predicado entre el sujeto i y el objeto j con valor k. Obviamente, se necesita además un mapeo entre las distintas entidades (por un lado la unión de sujetos y objetos y por otro los predicados) y valores enteros.

Por otro lado, el trabajo de Zhanget et al. plantea utilizar un cubo RDF. Los autores especifican el cubo como una colección de matrices dispersas, donde cada matriz está asociada a un predicado. En otras palabras, la dimensión principal del cubo es la cantidad de predicados presentes en el grafo. Mientras que las otras dos dimensiones depende de cada matriz. Notar que al utilizar un formato de almacenamiento disperso tipo Elemental [5] esto no genera mayores restricciones. En realidad, se utiliza una variante del formato elemental donde se almacena solamente los valores de fila y columna, es decir se omite el valor que está implicito por el identificador de la matriz.

VI. Conclusiones y trabajo futuro

En el trabajo se explora el uso de operaciones de álgebra lineal numérica para acelerar el tiempo de ejecución de consultas sobre bases de datos de grafos. Además, al expresar las consultas en términos de operaciones de ALN se puede explotar en forma directa el poder de cómputo de plataformas masivamente paralelas, en especial los procesadores gráficos (GPUs), y valerse del importante volumen de trabajo sobre la aceleración de operaciones de ALN en GPUs desarrollados por la comunidad.

El prototipo desarrollado está fuertemente basado en cubrir seis consultas que abarcan los requisitos más comunes en el contexto de RDF y SPARQL. En cuanto a las estructuras de datos utilizadas, se define el uso de dos funciones de hash que mapean, en un caso, los objetos y sujetos a los enteros y, en el otro, los predicados a los enteros. Además, se utiliza una matriz dispersa para representar el grafo en RDF, cuyas dimensiones son la cardinalidad del mapeo de objetos y sujetos y donde cada posición no nula en la matriz es una lista de identificadores de predicados.

Los resultados experimentales obtenidos, al utilizar una GPU de bajo porte, muestran que existe una ventaja al utilizar una GPU para los casos de consultas con resultados con muchas triplas. Esta situación está alineada con la teoría, ya que al usar operaciones de ALN para realizar las consultas no cambia la dimensión del resultado. En otras palabras, no es necesario realizar las recorridas sobre las triplas resultado (a menos de la etapa de despliegue en pantalla). Adicionalmente, las GPUs permiten alcanzar su máximo desempeño cuando el volumen de datos a manejar es importante.

El prototipo desarrollado permitió corroborar ciertas hipótesis en cuanto a desempeño. Es decir, que ciertas consultas se podrían hacer en base a operaciones de ALN, y estas aceleradas al utilizar una GPU para su cómputo. Sin embargo, muchas características del prototipo propuesto pueden ser extendidas o mejoradas. Algunas de estas mejoras posibles se enumeran a continuación:

- Incorporar el prototipo a un motor de base de datos abierto. Esta opción permite, de manera sencilla, acelerar ciertas operaciones de un motor genérico (al utilizar la propuesta) sin la necesidad de desarrollar por completo el motor.
- Utilización de otras técnicas de optimización. El prototipo no utiliza ningún tipo de técnica para acelerar la ejecución, fuera de explotar el poder de cómputo de la GPU para hacer las operaciones necesarias para resolver consultas. Existen varias técnicas típicas para acelerar este tipo de cómputos, entre otros pre cargar la matriz que representa el grafo o utilizar diccionarios tipo cache.
- Extender las consultas. Si bien el análisis realizado muestra que con las consultas cubiertas se alcanza gran parte de los requerimientos en bases de datos RDF, parece

interesante avanzar hacia una propuesta que abarque por completo la especificación de SPARQL 1.1.

- Otra línea de investigación que puede ser aprovechada son las operaciones de tipo *bitwise* en GPU, muchos de los resultados intermedios de los algoritmos que resuelven consultas son 0 o 1 esto podría aumentar la eficiencia de las operaciones necesarias.
- Por último, corresponde realizar evaluaciones experimentales del prototipo realizado en GPUs de alta gama.

REFERENCIAS

[1] Mario Arias, Javier D. Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. An empirical study of real-world SPARQL queries. *CoRR*, abs/1103.5043, 2011.

[2] Dave Beckett and Tim Berners-Lee. Turtle - Terse RDF Triple Language, 2011.

[3] B. Berendt, L. Hollink, V. Hollink, M. Luczak-Rösch, K. H. Möller, and D. Vallet, editors. *WWW20111st International Workshop on usage analysis and the Web of Data, Hyderabad, India, March 29, 2011*, 20th International World Wide WebConference (WWW2011), 2011.

[4] Orri Erling. Virtuoso, a Hybrid RDBMS/Graph Column Store. Technical report, OpenLink Software, 2012.

[5] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[6] Steve Harris and Andy Seaborne. SPARQL 1.1 query language, 2013.

[7] Fuad Jamour, Ibrahim Abdelaziz, and Panos Kalnis. A demonstration of magiq: Matrix algebra approach for solving rdf graph queries. *Proc. VLDB Endow.*, 11(12):1978–1981, August 2018.

[8] D. Kirk and W. Hwu. *Programming Massively Parallel Processors, Second Edition: A Hands-on Approach*. Morgan Kaufmann, 2012.

[9] G. Klyne, J. J. Carroll, and B. McBride. Resource description framework (RDF): Concepts and abstract syntax, 2004.

[10] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19:91–113, February 2010.

[11] M. Tamer Özsu. A survey of rdf data management systems. *Frontiers of Computer Science*, 10(3):418–432, Jun 2016.

[12] W3C. Resource description framework (RDF), 2014.

[13] Xiaowang Zhang, Mingyue Zhang, Peng Peng, Jiaming Song, Zhiyong Feng, and Lei Zou. A scalable sparse matrix-based join for sparql query processing. In Guoliang Li, Jun Yang, Joao Gama, Juggapong Natwichai, and Yongxin Tong, editors, *Database Systems for Advanced Applications*, pages 510–514, Cham, 2019. Springer International Publishing.

[14] Lei Zou, Jinghui Mo, Lei Chen, M. Tamer Özsu, and Dongyan Zhao. gstore: Answering SPARQL queries via subgraph matching. *PVLDB*, 4(8):482–493, 2011.

Referencias bibliográficas

[1] G. Klyne, J. J. Carroll, and B. McBride. Resource description framework (RDF): Concepts and abstract syntax, 2004. URL http://www.w3.org/TR/rdf-concepts/.

[2] Dave Beckett and Tim Berners-Lee. Turtle - Terse RDF Triple Language, 2011. URL http://www.w3.org/TeamSubmission/turtle/.

[3] W3 Org. Sparql query language for rdf, Febrero, 2018. URL https://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[4] Wikipedia contributors. Foreign key, Febrero, 2019. URL https://en.wikipedia.org/wiki/Foreign_key.

[5] Mauricio Bello Claudia Pinilla and Cristian Peña. Bases de datos orientadas a grafos. http://revistas.udistrital.edu.co/ojs/index.php/tia/article/viewFile/8769/pdf, Julio, 2017.

[6] Wikipedia contributors. Nosql, Mayo, 2019. URL https://en.wikipedia.org/wiki/NoSQL.

[7] Wikipedia contributors. Nosql document oriented databases, Abril, 2019. URL https://en.wikipedia.org/wiki/Document-oriented_database.

[8] Wikipedia contributors. Sql, Abril, 2019. URL https://en.wikipedia.org/wiki/SQL.

[9] Wikipedia contributors. Sparql, Febrero, 2019. URL https://en.wikipedia.org/wiki/SPARQL.

[10] Wikipedia contributors. Gpgpu, Abril, 2019. URL https://es.wikipedia.org/wiki/GPGPU.

[11] M. Tamer Özsu. A survey of rdf data management systems. Frontiers of Computer Science, 10(3):418–432, Jun 2016. ISSN 2095-2236. doi: 10.1007/s11704-016-5554-y. URL https://doi.org/10.1007/s11704-016-5554-y.

[12] W3 Org. Virtuosouniversalserver openlink virtuoso, Julio, 2018. URL https://www.w3.org/wiki/VirtuosoUniversalServer.

[13] Orri Erling. Virtuoso, a hybrid rdbms/graph column store. IEEE Data Eng. Bull., 35(1):3–8, 2012. URL http://sites.computer.org/debull/A12mar/vicol.pdf.

[14] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. VLDB J., 19(1):91–113, 2010. doi: 10.1007/s00778-009-0165-y. URL https://doi.org/10.1007/s00778-009-0165-y.

[15] Lei Zou, M. Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang, and Dongyan Zhao. gstore: a graph-based SPARQL query engine. VLDB J., 23 (4):565–590, 2014. doi: 10.1007/s00778-013-0337-7. URL https://doi.org/10.1007/s00778-013-0337-7.

[16] Roberto De Virgilio. A linear algebra technique for (de)centralized processing of SPARQL queries. In Conceptual Modeling - 31st International Conference ER 2012, Florence, Italy, October 15-18, 2012. Proceedings, pages 463–476, 2012. doi: 10.1007/978-3-642-34002-4_36. URL https://doi.org/10.1007/978-3-642-34002-4_36.

[17] Steve Borgatti. Graphs and matrices. URL http://www.analytictech.com/networks/graphs.htm.

[18] Mario Arias, Javier D. Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. An empirical study of real-world SPARQL queries. CoRR, abs/1103.5043, 2011. URL http://arxiv.org/abs/1103.5043.

[19] The suitesparse matrix collection, Noviembre, 2019. URL https://sparse.tamu.edu/.

[20] Christian Bizer, Andreas Schultz. The berlin sparql benchmark, Noviembre, 2019. URL https://www.semanticscholar.org/paper/The-Berlin-SPARQL-Benchmark-Bizer-Schultz/0efcd1d38ad020da7c01613b7818eb123cb34121.

[21] Wikipedia contributors. Nvlink, Abril, 2019. URL https://en.wikipedia.org/wiki/NVLink.

[22] Videocardz. Nvidia mx150, 2019. URL https://videocardz.net/nvidia-geforce-mx150/.

[23] NVIDIA. Nvidia tesla v100, 2019. URL https://www.nvidia.com/en-us/data-center/tesla-v100/.