Supporting Property Graphs in Apache Giraph

Renzo Angles*†, Federico Meza‡, Francisco Moya§
Departamento de Ciencias de la Computación, Universidad de Talca
Camino Los Niches Km. 1 s/n - Curicó - CHILE
*Centro de Investigación de la Web Semántica (http://ciws.cl/)
Email: †rangles@utalca.cl, ‡fmeza@utalca.cl, §fmoya09@alumnos.utalca.cl

***Resumen*—Apache Giraph is a powerful tool for processing very large graphs in distributed environments. One of its main features is a novel programming model that facilitates the design and execution of graph-oriented algorithms in distributed environments.**

Despite its benefits, using Giraph is a complex and challenging process. Moreover, the Giraph data model (based on labeled directed graphs) makes its use difficult in application domains where nodes and edges contain metadata.

This article presents an extension of the Giraph API to provide support for Property Graphs, that is, graphs whose nodes and edges could have properties. In particular, we define a formal method to transform a Property Graph into a Giraph graph, we describe an API for manipulating and querying Property Graphs in Giraph, and we present experimental results that show the applicability and efficiency of our extension.

***Index Terms*—Apache Giraph, Graph Processing Frameworks, Property Graphs.**

I. Introducción

Las bases de datos no relacionales han surgido como respuesta a las necesidades de gestión de datos en dominios de aplicación de gran relevancia como la *World Wide Web*, las redes sociales y la Bioinformática, entre otros. Los datos generados por muchos de estos dominios de aplicación pueden ser fácilmente modelados y analizados como grafos, caracterizados por su estructura irregular, su alta interconexión, y su gran tamaño. Por ejemplo, la *World Wide Web* contiene mas de 50 billones de páginas web sobre las cuales el algoritmo de *page rank* es ejecutado; Facebook tiene más de 1.65 billones de usuarios, cada uno con cientos de relaciones las cuales son exploradas por herramientas de *social media analysis*; el *Protein Data Bank* (PDB) registra un poco más de cien mil proteínas –de un posible universo de 10^{12}– cada una de las cuales contiene cientos de aminoácidos y átomos cuyas miles de relaciones son estudiadas por biólogos y bioinformáticos en busca de tratamientos para las enfermedades. Para cumplir con el desafío de procesar dichas cantidades de datos con estructura de grafo –o *graph data*– se han desarrollado diversos sistemas de procesamiento de datos.

La gestión de datos de grafos (*Graph Data Management* [1]) involucra la investigación y desarrollo de sistemas y tecnologías para el almacenamiento, procesamiento y análisis de grandes volúmenes de datos con estructura de grafo.

978-1-5090-1633-4/16/$31.00

Dentro de esta área, podemos identificar dos tipos de sistemas: las bases de datos orientadas a grafos (*graph databases*), y los sistemas de procesamiento para grafos (*graph processing frameworks*). Los primeros son sistemas desarrollados para el almacenamiento y consulta de grafos siguiendo las técnicas tradicionales de base de datos, es decir, son muy similares a las bases de datos relacionales. Los segundos son *frameworks* enfocados en el procesamiento y análisis de grandes grafos a través de múltiples computadores, con el objetivo de disminuir el tiempo de ejecución de los algoritmos involucrados. Estos tipos de sistemas representan dos enfoques distintos para la gestión de grafos, cada uno con sus propias ventajas y desventajas.

Los sistemas de procesamiento para grafos surgieron debido a la necesidad de procesar grandes y complejas colecciones de datos, las cuales son difíciles –por no decir imposibles– de almacenar y procesar en un solo computador. En consecuencia, estos sistemas se caracterizan por realizar procesamiento por lotes en memoria (*in-memory batch processing*) sobre una plataforma computacional de procesamiento paralelo y/o distribuido. Cabe mencionar que el uso de un sistema distribuido con mayores recursos computacionales –básicamente, procesador y memoria– abre la posibilidad de procesar muchos datos; sin embargo, este sistema puede entregar peor desempeño que un único computador cuando se ejecutan ciertos tipos de algoritmos [2].

Los sistemas de procesamiento para grafos se pueden dividir en dos grupos, dependiendo de su forma de implementación. Por un lado, encontramos los sistemas genéricos de procesamiento masivo que han sido adaptados para procesar grafos, la mayoría de ellos basados en el modelo de programación *MapReduce*. Dentro de este grupo encontramos a Hadoop [3], YARN [4], y Stratosphere [5]. Aunque estos sistemas mejoran el tiempo de computación, los usuarios necesitan traducir sus consultas de grafos al modelo de trabajo de *MapReduce*. De hecho, se ha comprobado que expresar algoritmos sencillos para grafos en dicho modelo resulta ser un gran desafío [6]. Adicionalmente, estos sistemas no pueden tomar ventaja de las características de los grafos, y la implementación de algoritmos iterativos frecuentemente resulta en complejas cadenas de trabajo y un movimiento excesivo de datos entre los componentes de la plataforma [7].

En un segundo grupo encontramos sistemas diseñados específicamente para trabajar con grafos. Estos sistemas comúnmente proveen una interfaz de programación (API) que

permite expresar algoritmos de análisis para grafos de una manera más fácil y directa. En este grupo encontramos a Pregel [8], Apache Giraph [9] y GraphLab [10], entre otros. Estos sistemas, usualmente llamados sistemas de procesamiento *offline* para análisis de grafos (*offline graph analytic systems*), se basan en un procesamiento iterativo por lotes sobre el total de los datos, hasta que el método de computación alcanza un punto fijo (*fixed-point*) o un criterio de término. De esta manera, estos sistemas están capacitados para ejecutar algoritmos para grafos que requieren una exploración total de los datos y el uso intensivo de los recursos computacionales [11], como por ejemplo el algoritmo de *PageRank*, consultas recursivas como la búsqueda de *shortest-paths*, y algoritmos de *clustering*.

Apache Giraph –en adelante simplemente *Giraph*– es un *framework* para el procesamiento de grafos de gran tamaño sobre una plataforma distribuida, con una alta tolerancia a fallas [9]. En términos prácticos, es una implementación libre de *Google Pregel*, la arquitectura de procesamiento de grafos desarrollada por Google [8]. Giraph se ejecuta como un trabajo de sólo mapping sobre el entorno distribuido Hadoop, ampliamente utilizado para el análisis de Big Data [3], [12]. Desde su lanzamiento en 2013, Giraph ha sido empleado en diversos dominios de aplicación [13], [14], [15], [2], [16] y ha sido considerado en diversas evaluaciones empíricas de los sistemas de procesamiento para grafos [7], [17], [18], [19].

Una buena prueba de la capacidad de Giraph para procesar grafos de gran tamaño es su uso en Facebook. Ching, et al., condujeron un estudio calculando el algoritmo *PageRank* sobre el grafo de Facebook, compuesto de 1.39 billones de usuarios, incluyendo más de un trillón de relaciones sociales entre ellos [2].

Giraph hereda de Pregel un modelo de programación centrado en los vértices (*vertex-centric programming model*), en el cual cada vértice puede intercambiar mensajes con otros vértices en una secuencia de iteraciones. En ningún momento el usuario debe lidiar con la complejidad de programación de un sistema de procesamiento paralelo y/o distribuido, incluida la distribución de los datos sobre el cluster o la forma en que se implementa la tolerancia a fallas.. De esta manera, el programador se concentra en los aspectos específicos del algoritmo, mientras que Giraph se encarga de ejecutar el algoritmo en paralelo y en una plataforma distribuida de manera transparente, obteniendo aplicaciones escalables, descentralizadas y masivamente paralelas [20].

Si bien el paradigma de programación de Giraph permite una implementación más sencilla de algoritmos sobre grafos, esto en comparación con los sistemas de programación paralela/distribuida tradicionales. Su uso no es tan sencillo como se argumenta. Sólo la instalación y configuración inicial de Giraph, considerando las bibliotecas involucradas y la plataforma distribuida, toma como mínimo 25 minutos, según nuestros experimentos con cuatro máquinas, y requiere de conocimientos intermedios de administración de sistemas. La ejecución de un algoritmo se puede resumir en tres pasos: carga de datos, ejecución del algoritmo, y procesamiento del resultado. La carga de datos implica transformar los datos originales a alguno de los formatos de texto soportados por Giraph, tarea que puede tomar bastante tiempo dependiendo de la estructura de los datos y el formato elegido. Como se indicó anteriormente, los algoritmos deben ser programados centrándose en los vértices. Esto puede parecer sencillo, sin embargo es un gran desafío adaptarse a este nuevo paradigma de programación ya que es muy distinto a la forma tradicional de diseñar algoritmos para grafos. Si bien las funciones básicas del API de Giraph son fáciles de entender, existen algunas funciones que no son intuitivas y para las cuales existe poca documentación. Finalmente, el resultado de ejecutar Giraph se traduce en un archivo plano que contiene una linea de texto para cada nodo/etiqueta en el grafo. Claramente, este tipo de resultado implica un procesamiento y análisis adicional.

Cabe destacar que el modelo de datos de Giraph se basa en multigrafos dirigidos y etiquetados, es decir, cada nodo o arista tiene una etiqueta o valor, las aristas son dirigidas, y pueden existir múltiples aristas entre un mismo par de nodos. Si bien este es el modelo básico empleado en la teoría de grafos, actualmente se emplean otros modelos que se adaptan mejor a la estructura de los datos reales. En particular, el modelo de datos denominado "grafo con propiedades" (*property graph data model*) está siendo usado en diversos dominios de aplicación reales [21]. La principal característica de un *property graph* es que tanto nodos como aristas pueden contener múltiples pares llave-valor (*key-value*), cada uno de ellos representando un atributo del nodo o arista. Giraph carece de soporte para este modelo de grafos.

Las contribuciones de este artículo son las siguientes:

- Mostramos y discutimos la complejidad de emplear Giraph, para lo cual proveemos una guía paso a paso de su instalación y uso (Sección II);
- Definimos formalmente el modelo de datos de Giraph, y presentamos un método para transformar cualquier *property graph* a un *Giraph graph* (Sección III).
- Estandarizamos el formato de codificación de datos de entrada para Giraph, además de desarrollar una herramienta que facilita el proceso de transformación de datos a dicho formato (Sección IV);
- Presentamos un interfaz de programación (API), la cual modifica y/o extiende el API original de Giraph con funciones que permiten manipular y consultar *property graphs* (Sección IV).
- Presentamos resultados experimentales del uso de la API propuesta, eligiendo una red social como caso de uso (Sección V).

II. *Apache Giraph*

Las aplicaciones en Giraph se ejecutan sobre un cluster Hadoop, soportado por una instancia del sistema de archivos distribuido HDFS. Los datos modelados en el dominio de la aplicación se almacenan en una serie de archivos en diversos formatos, como por ejemplo, texto plano o CSV.

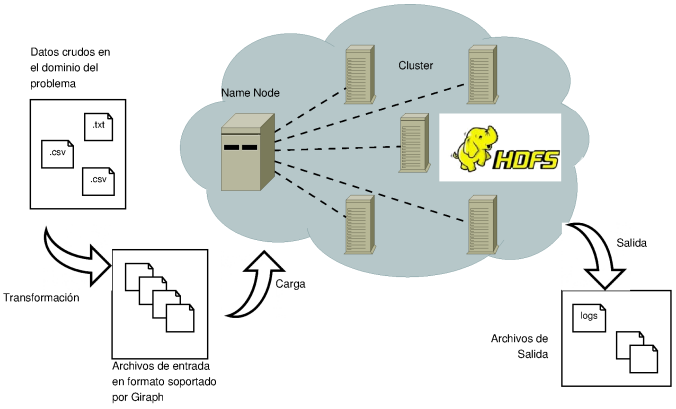

Figura 1. Workflow de Apache Giraph.

Posteriormente, los datos de entrada deben ser convertidos a uno de los formatos de entrada que soporta Giraph, en uno o múltiples archivos, los que deben ser cargados al sistema de archivos distribuido. Desde ahí, Giraph podrá construir el grafo involucrado como una serie de objetos `Vertex` y `Edge`. Como salida, la aplicación puede construir archivos en un formato de salida particular, y Giraph genera además bitácoras para efectuar un análisis posterior del comportamiento de la ejecución. La Figura 1 muestra este flujo de ejecución de forma gráfica.

II-A. Carga de datos

Si bien todas las computaciones que lleva a cabo Giraph giran alrededor de las clases de objetos `Vertex` y `Edge`, que modelan los componentes de un grafo, existe una gran cantidad de formatos en los cuales pueden almacenarse los datos que definen un grafo particular. Estos formatos pueden ser utilizados tanto para los datos de entrada como para las salidas de las aplicaciones. La implementación de un formato de entrada define la forma en que los datos serán leídos desde el sistema de archivos, y cómo serán convertidos a un conjunto de objetos `Vertex` y `Edge`.

Los formatos de entrada de Giraph son implementaciones de dos clases abstractas: `VertexInputFormat` y `EdgeInputFormat`, que agrupan las dos categorías de tipos de representación existentes. A saber, aquéllos centrados en la información de vértices y de aristas, respectivamente. También se proveen interfaces especializadas para soportar distintos sistemas de almacenamiento, por ejemplo, mediante archivos de texto plano o mediante tablas de *HBase*, ya sea en sistemas de archivos locales o distribuidos como el *Hadoop Distributed File System* (HDFS). Por ejemplo, `TextVertexInputFormat` es una extensión de `VertexInputFormat` diseñada para la entrada de grafos almacenados como archivos de texto en HDFS. Por su parte, la clase abstracta `AdjacencyListTextVertexInputFormat` puede manejar formatos de texto en los que el grafo se encuentra descrito como una listas de adyacencia [20].

```
0  0  1  1  3  3
1  0  0  1  2  2  3  1
2  0  1  2  4  4
3  0  0  3  1  1  4  4
4  0  3  4  2  4
```

Figura 2. Ejemplo de archivo de entrada para Giraph, usando el formato de entrada `LongDoubleDoubleAdjacencyListVertexInputFormat`.

Para recibir los datos de entrada para una aplicación, Giraph sigue el mismo enfoque que Hadoop, procesando todos los archivos que se encuentran en un subdirectorio especificado. Estos archivos son considerados como parte de la misma definición de grafo.

La Figura 2 muestra un archivo de entrada con un grafo simple, utilizando el formato `LongDoubleDoubleAdjacencyListVertexInputFormat` de Giraph. En este formato, cada línea del archivo corresponde a la información de un vértice del grafo. La linea comienza con el identificador del vértice seguido del valor del vértice. Como lo indica el nombre del formato, el identificador es de tipo `Long`, mientras que el valor es de tipo `Double`. A continuación, en cada línea aparece una lista de pares de valores (lista de adyacencia) en donde el primero corresponde al identificador del vértice destino de una arista saliente y del valor asociado con dicha arista. Según lo indica el nombre del formato, el valor de las aristas también de tipo `Double`.

II-B. Modelo de programación

Como se mencionó en la sección anterior, Giraph sigue un modelo de programación centrado en los vértices. La escritura de los programas obliga al programador a pensar en la forma en que lo haría un vértice del grafo. A este concepto se le llama *Think like a vertex* [20].

Los vértices cuentan con un identificador y con un valor que puede cambiar a lo largo de la ejecución. Además, cada vértice tiene asociada una lista de las aristas salientes, que definen su vecindario. Por su parte, las aristas cuentan con un valor que también podrá cambiar a lo largo de la ejecución, así como una referencia al identificador del vértice al que llega.

Durante la ejecución de los programas, cada vértice del grafo intercambia mensajes con sus vecinos al final de iteraciones similares a los *supersteps* del modelo BSP (*Bulk Synchronous Parallel*) [22]. En cada uno de estos *supersteps*, el programa ejecuta el método Java `compute()` sobre cada vértice del grafo.

En cada *superstep* el vértice puede acceder a su valor almacenado y a las aristas salientes, incluidos sus valores. Además, puede crear nuevas aristas o eliminar las existentes. Finalmente, un vértice puede detener su ejecución, situación que puede ser permanente a no ser que reciba nuevos mensajes en algún *superstep* posterior.

Los programadores cuentan con una API del lenguaje de programación Java cuyos métodos se agrupan en tres clases de objetos: `Vertex`, `Edge` y `BasicComputation`.

Los vértices del grafo cuentan con los siguientes métodos de la clase `Vertex`:

```
getId()
getValue()
setValue()
getEdges()
getNumEdges()
getEdgeValue(targetId)
setEdgeValue(targetId, value)
getAllEdgeValues(targetId)
voteToHalt
addEdge(edge)
removeEdges(targetId)
```

Las aristas cuentan con los siguientes métodos de la clase `Edge` para su manipulación:

```
getTargetVertexId()
getValue()
setValue(value)
```

Finalmente, la clase `BasicComputation` provee al programador de un conjunto de métodos para especificar el comportamiento de un vértice. El método `compute()` será ejecutado en cada *superstep* por cada vértice. El comportamiento que se programa para el vértice determina la forma en que se manejan los mensajes entrantes, la actualización del valor del vértice, la creación o eliminación de aristas salientes, y el envío de mensajes. A los métodos descritos anteriormente se agregan:

```
compute(vertex, message)
getSuperstep()
sendMessage(targetId, message)
sendMessagetoAllEdges(vertex, message)
```

Los mensajes enviados por un vértice son despachados al inicio del siguiente *superstep*. La computación se detiene cuando todos los vértices se han detenido y no hay mensajes en tránsito El *framework* oculta los detalles de la ejecución paralela, por lo que la implementación de `compute()` no requiere ningún tipo de *locking* o sincronización explícita.

La Figura 3 muestra el método `compute()` de un programa que hace uso de la API de Giraph para determinar el camino más corto entre un vértice del grafo –en este caso el primero– y todos los demás. Esto se conseguirá haciendo que cada vértice guarde en su valor la menor distancia desde el vértice origen hasta él.

En el primer *superstep* cada vértice inicializa su valor a un máximo. En cada iteración, un vértice recibe de sus vecinos la distancia que toma llegar a él a través de ellos, y almacena en el valor del vértice la menor de ellas. Una vez hecho esto, el vértice envía a sus vecinos la nueva distancia para llegar a ellos, sumando el valor almacenado en el vértice con el costo de la arista que lo une con cada uno de ellos. Estos mensajes serán despachados en el siguiente *superstep*. Luego de esto el vértice se bloquea (voteToHalt()) a la espera de nuevos mensajes o del término de la computación.

II-C. Instalación de Apache Giraph

La instalación de Giraph es un proceso complejo que involucra la instalación y configuración de diversas herramientas y un conocimiento intermedio de administración de sistemas. En resumen, contempla las siguientes etapas:

1. Instalación de Java SDK 7
2. Instalación de Hadoop
   - Creación de usuario y configuración de permisos
   - Instalación de Hadoop
   - Ajuste de archivos de configuración
   - Configuración de ssh
   - Configuración de computadores maestro y esclavos en Hadoop
   - Formateo del sistema de archivos distribuido (HDFS)
   - Inicialización de Hadoop
   - Prueba de la instalación
3. Instalación de Giraph
   - Obtención de distribución a partir de GitHub
   - Modificación de archivos de configuración
   - Copia de archivos al HDFS
   - Prueba de la instalación

Los autores han elaborado una guía detallada para la instalación y configuración de Giraph, así como para la ejecución de aplicaciones, la que se encuentra disponible en línea (http://dev.iccutal.cl).

III. MODELANDO *Property Graphs* EN GIRAPH

El modelo de grafo de Giraph contempla un identificador y un valor asociado con cada vértice, y sólo un valor para el caso de las aristas. Sin embargo, carece de tipos de nodos y aristas, y no permite la inclusión de propiedades arbitrarias asociadas con nodos y aristas. Esta restricción le resta generalidad al modelo e impide modelar una diversidad de datos reales. En esta sección se presenta una extensión del modelo de datos de Giraph para soportar *property graphs*.

Un *Property Graph* consiste en un grafo dirigido basado en propiedades expresadas como pares *key/value*. Tanto los vértices como las aristas pueden tener cualquier número de propiedades de este tipo. Además, puede haber distintos tipos de aristas, identificadas por una etiqueta o *label*, lo que permite modelar varios tipos de relaciones entre los vértices. A continuación presentamos una formalización del modelo de datos de Giraph, y un método formal para transformar un *property graph* en un *Giraph graph*.

III-A. Modelo de datos de Giraph

Informalmente, un Grafo Giraph (*Giraph Graph*) es básicamente un multigrafo dirigido y etiquetado, es decir, un grafo donde nodos y aristas tienen etiquetas, las aristas son dirigidas, y dos nodos pueden estar conectados por más de dos aristas. Adicionalmente, cada nodo tiene un identificador

```
public void compute(Vertex<Text, DoubleWritable, FloatWritable> vertex,
    Iterable<DoubleWritable> messages) throws IOException {

    if (getSuperstep() == 0) {
        vertex.setValue(new DoubleWritable(Double.MAX_VALUE));
    }
    double minDist = isSource(vertex) ? 0d : Double.MAX_VALUE;
    for (DoubleWritable message : messages) {
        minDist = Math.min(minDist, message.get());
    }

    if (minDist < vertex.getValue().get()) {
        vertex.setValue(new DoubleWritable(minDist));
        for (Edge<Text, FloatWritable> edge : vertex.getEdges()) {
            double distance = minDist + edge.getValue().get();
            sendMessage(edge.getTargetVertexId(), new DoubleWritable(distance));
        }
    }
    vertex.voteToHalt();
}
```

Figura 3. Código Java de un ejemplo de programa en Giraph.

el cual es empleado para referenciar al nodo. La etiqueta de un nodo/arista es empleada para almacenar un valor el cual puede cambiar durante la ejecución de Giraph; en particular, las etiquetas de aristas son empleadas para enviar mensajes entre los nodos del grafo durante la ejecución de Giraph. A continuación presentamos la definición formal de un grafo Giraph.

Asumamos un conjunto infinito de identificadores (para nodos), **I**, y un conjunto infinito de valores, **V**. Dado un conjunto X, usaremos $\mathrm{SET}(X)$ para denotar todos los subconjuntos finitos de X.

Definition III.1. *Un Giraph Graph (GG) es un par* $G_G = (N, E, I_N, \phi, \psi, \delta, \tau)$ *donde:*

- N *es un conjunto finito de nodos (también llamados vértices);*
- E *es un conjunto finito de aristas, cumpliendo que* E *no tiene elementos en común con* N*;*
- I_N *es un subconjunto finito de* ***I****, satisfaciendo que* $|I_N| = |N|$ *(se decir,* I_N *y* N *tienen el mismo número de elementos);*
- $\phi : N \to I_N$ *es una función total que asocia cada nodo de* N *con un único identificador de* I_N*;*
- $\psi : (N \cup E) \to \boldsymbol{V}$ *es una función total que asocia cada nodo/arista con un único valor en* ***V****;*
- $\delta : N \to \mathrm{SET}(E)$ *es una función total que asocia cada nodo* $n \in N$ *con un subconjunto de aristas* $\mathrm{SET}(E)$*, el cual contiene a los aristas salientes de* n*; y*
- $\tau : E \to I_N$ *es una función total que asocia cada arista* e *con un identificador de nodo, el cual corresponde al nodo "destino" (u objetivo) de* e*.*

Nótese que la definición anterior varía bastante respecto de la definición usual de un grafo dirigido y etiquetado, donde la relación entre nodos y aristas suele definirse a través de una función de adyacencia que vincula cada arista e con un par de nodos (n, n'). Por otra parte, el modelo de datos de Giraph hace uso explícito de identificadores para referenciar nodos y aristas, mientras que en los algoritmos de grafo tradicionales se hace uso de las etiquetas.

III-B. Modelo de datos para Property Graphs

Un grafo con propiedades (*Property Graph*), es una multigrafo dirigido y etiquetado que se caracteriza porque cada nodo o arista mantiene un conjunto (posiblemente vacío) de propiedades representadas como pares llave-valor. Desde el punto de vista de modelado de datos, un nodo representa una entidad, un arista representa un relación entre dos entidades, y una propiedad representa un característica específica de una entidad o una relación. A continuación presentamos una definición formal del modelo de datos para *Property Graphs*[1].

Asumamos un conjunto infinito de etiquetas (para nodos y aristas), **L**, un conjunto infinito de nombres de propiedades, **P**, y un conjunto infinito de literales (o valores), **V**.

Definition III.2. *Un property graph (PG) es una tupla* $G_P = (N, E, \rho, \lambda, \sigma)$ *donde:*

- N *es un conjunto finito de nodos (también llamados vértices);*
- E *es un conjunto finito de aristas, cumpliendo que* E *no tiene elementos en común con* N*;*
- $\rho : E \to (N \times N)$ *es una función total que asocia cada arista de* E *con un par de nodos en* N *(es decir,* ρ *es la usual función de incidencia en teoría de grafos);*
- $\lambda : (N \cup E) \to \boldsymbol{L}$ *es una función parcial que asocia un nodo/arista con un único valor en* L *(es decir,* λ *es una función de etiquetado para nodos y aristas);*
- $\sigma : (N \cup E) \times \boldsymbol{P} \to \boldsymbol{V}$ *es una función parcial que asocia nodos y aristas con propiedades, y asigna un valor a cada una de dichas propiedades.*

[1]Cabe mencionar que actualmente no existe una definición formal estándar del *Property Graphs Data Model*. La definición presentada en este artículo es bastante común en el contexto de las bases de datos para grafos.

Dados dos nodos $n_1, n_2 \in N$ y una arista $e \in E$, cumpliendo que $\rho(e) = (n_1, n_2)$, emplearemos $e = (n_1, n_2)$ como una representación corta de la arista e, donde n_1 y n_2 se denominan el nodo "origen" y el node "destino" del arista e respectivamente. No es obligatorio que todo nodo/arista tenga una etiqueta, es decir, el valor de $\lambda(o)$ puede ser indefinido (vacío) para un nodo/arista o. Por otra parte, dado un par $(o, p) \in (N \cup E) \times \mathbf{P}$ y la asignación $\sigma((o, p)) = v$, usaremos $(o, p) = v$ como una representación corta de una propiedad p donde o es el "dueño" de la propiedad, p es el "nombre" de la propiedad, y v es el "valor" de la propiedad.

III-C. Transformación Property Graph → Giraph Graph

Si comparamos un Giraph Graph (GG) y Property Graph (PG) podemos encontrar que ambos comparten la estructura básica de un grafo, es decir; ambos están compuestos de nodos y aristas, los cuales podrían ser etiquetados con algún valor; las aristas son dirigidas; y es posible tener varias aristas entre dos nodos. Por otro lado, también encontramos diferencias sustanciales:

- los nodos y aristas de un PG soportan propiedades, mientras que los nodos y aristas de un GG están restringidos a contener un único valor;
- cada nodo de un GG posee un identificador que se emplea para referenciarlo, mientras que un PG no emplea identificadores;
- la lista de aristas adyacentes a un nodo n se puede obtener directamente en un GG a través de la función $\delta(n)$, mientras que en un PG dicha lista deberá construirse empleando la función de adyacencia $\rho(e)$, esto para cada arista e; y
- para obtener los nodos origen y destino de una arista e, en un PG es suficiente emplear la función $\rho(e)$, mientras que en un GG, la función $\tau(e)$ retorna el nodo destino, y el nodo origen requiere una exploración de la lista de adyacencia de cada nodo n a través de la función $\delta(n)$.

De las diferencias mencionadas anteriormente, la más importante es la capacidad de un PG para que sus nodos y aristas contengan propiedades. Esta diferencia es fundamental para facilitar el modelado de diversos casos de uso donde los nodos y aristas contienen metadatos, en especial las aristas. Por ejemplo, si consideramos una red social como Facebook, donde las personas comentan o hacen "like" a una foto de un amigo, es deseable almacenar la fecha en la cual se registró el comentario o el like (de hecho Facebook lo hace). Nótese que en PG, un *like* se representaría como una arista etiquetada con el valor "like" o "likes", cuyo nodo origen sería el nodo correspondiente al amigo, el nodo destino correspondería a la foto, y la fecha del *like* se registraría como el valor de una propiedad *fecha*. En el caso de un GG, no es posible un representación sencilla de la información anterior ya que una arista solo pueden contener un único valor. Peor aún, hay que recordar que dicho valor es usualmente empleado por Giraph para transferir información entre nodos durante la etapa de procesamiento del grafo, por lo tanto dicho valor no será estable en tiempo de ejecución.

Afortunadamente, existe una manera relativamente sencilla de poder extender el modelo de datos de Giraph para soportar property graphs. La idea consiste en codificar las propiedades dentro del valor asociado a cada nodo o arista. Considerando que el valor de un/una nodo/arista puede contener un texto de cualquier tipo y tamaño, entonces dicho valor podría contener múltiples pares llave-valor empleando algún formato de codificación. En nuestro caso, proponemos el uso de JSON por su simplicidad y facilidad de procesamiento (*parsing*).

JSON (JavaScript Object Notation) es un formato de texto ligero que está siendo muy usado por desarrolladores como alternativa al verboso XML. Al igual que XML, JSON implementa un modelo de datos semi-estructurado, es decir, los datos se organizan siguiendo una estructura (jerárquica) con forma de árbol. En nuestro caso, solo emplearemos el fragmento de JSON que permite representar objetos como una colección no ordenada de pares <llave>:<valor>, cada uno de los cuales representar un par nombre-propiedad, valor-propiedad respectivamente. Por ejemplo, el siguiente texto codifica la información de un objeto persona: {"nombre":"Juan","apellido":"Perez","edad":"20"}.

Consideremos una función de transformación f la cual recibe como entrada un property graph $G_P = (N, E, \rho, \lambda, \sigma)$ y entrega como resultado un Giraph graph $G_G = (N', E', I_N, \phi, \psi, \delta, \tau,)$. Específicamente, la función $f(G_P) = G_G$ se define de la siguiente manera:

1. Por cada nodo $n \in N$ existirá un nodo n' en N' satisfaciendo que:
   - $\phi(n') = k$ y $k \in I_N$;
   - $\psi(n')$ = {"_label":"v_0","p_1":"v_1",...,"p_r":"v_r"} donde $v_0 = \lambda(n)$ y existe $\sigma((n, p_i)) = v_i$ para $1 \leq i \leq r$.
2. Por cada arista $e \in E$, existirá una arista e' en E' satisfaciendo que:
   - $\psi(e')$ = {"_label":"v_0","p_1":"v_1",...,"p_s":"v_s"} donde $v_0 = \lambda(e)$ y existe $\sigma((e, p_j)) = v_j$ para $1 \leq j \leq s$;
   - e' pertence a $\delta(n')$ si y solo si $\rho(e) = (n, n_i)$ tal que $n \in N$ corresponde a $n' \in N'$;
   - $\tau(e') = \phi(n')$ si y solo si $\rho(e) = (n_j, n)$ tal que $n \in N$ corresponde a $n' \in N'$.

El punto más importante de la transformación es la codificación de las propiedades de nodos y aristas del property graph G_P. Básicamente, el conjunto de propiedades de un nodo (o arista) perteneciente a G_P se codifican como un texto JSON el cual queda almacenado en la etiqueta del nodo correspondiente en G_G. Nótese que "label" es una propiedad especial la cual representa la etiqueta de un nodo/arista en el Giraph graph, pudiendo ser empleada de manera usual. Para el caso de las aristas, "label" podrá seguir usándose para el envío de mensajes entre nodos. Cabe destacar que la transformación es lineal respecto al número de nodos y aristas en el property graph, y que a nivel de procesamiento, el único costo adicional corresponderá a la codificación y decodificación de los textos en JSON.

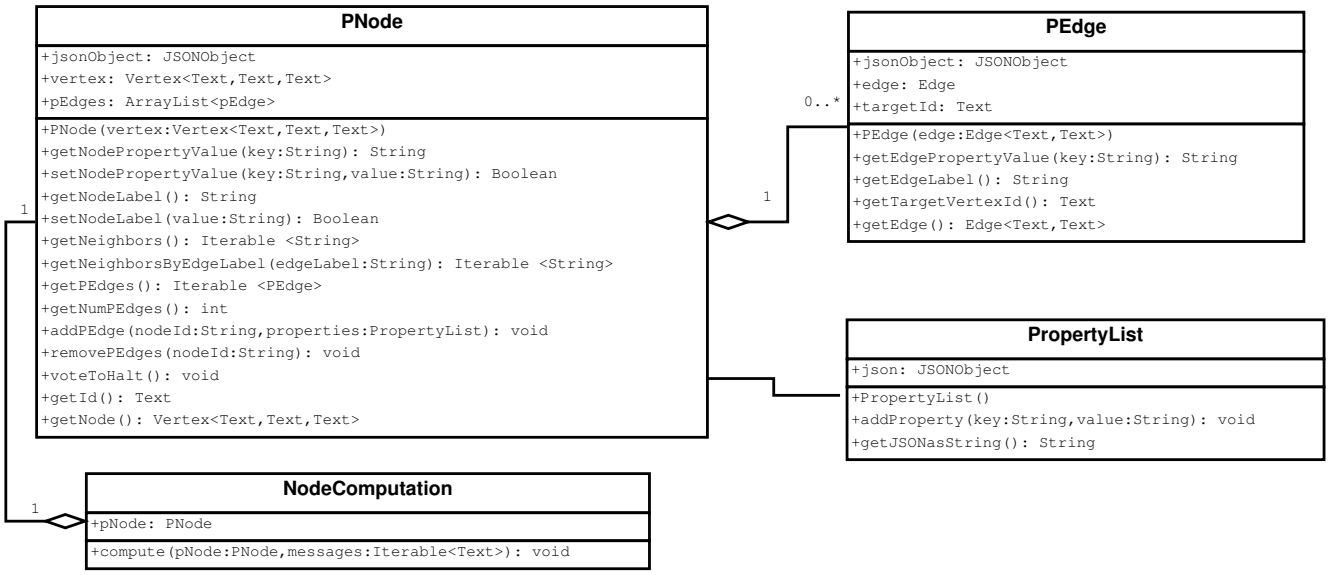

Figura 4. Principales clases de objetos, métodos públicos y relaciones que conforman la Interfaz de Programación.

IV. Interfaz de Programación

En esta sección describiremos una extensión de la API de Giraph para soportar *property graphs*, de acuerdo con la transformación definida en la Sección III. La API se encuentra disponible en línea (http://dev.iccutal.cl) para su descarga y consulta general.

La Figura 4 resume los métodos públicos que pone a disposición nuestra interfaz de programación para el modelo de *property graphs*. Los métodos se agrupan en 4 clases de objetos. Las clases `PNode` y `PEdge` modelan nodos y aristas, respectivamente, cada una de las cuales tiene un tipo (*label*) y puede además asociarse con un objeto de la clase `PropertyList`, que modela una lista de propiedades. La última clase, `NodeComputation`, encapsula el comportamiento de un nodo a lo largo de la secuencia de *supersteps* que constituyen la computación.

A continuación describimos en detalle los métodos de la API propuesta. Es necesario referirse al diagrama de clases de la Figura 4 por los parámetros y valores de retorno.

IV-A. Clase `PropertyList`

Ofrece métodos para la creación de listas de propiedades que serán asociadas a nodos o aristas del grafo.

- `PropertyList()`. Constructor que crea una lista de propiedades vacía.
- `addProperty()`. Agrega a la lista de propiedades una nueva propiedad con (`key`, `value`). Si la llave ya existe, modifica el valor asociado.
- `String getJSONasString()`. Retorna un String con la representación JSON de la lista de propiedades.

IV-B. Clase `PNode`

Pone a disposición del programador un conjunto de métodos públicos aplicables a un nodo del grafo.

- `getNodePropertyValue()`. Retorna el valor de la propiedad `key` en el nodo. Si la propiedad no existe genera una excepción y retorna `ERROR`.
- `setNodePropertyValue()`. Modifica el valor de la propiedad `key` en el nodo, con el valor indicado en `value`. Si la propiedad no existe la agrega. Retorna verdadero si la operación es exitosa, o falso en caso contrario.
- `getNodeLabel()`. Retorna el valor de la etiqueta del nodo, o `ERROR` en caso de excepción.
- `setNodeLabel()`. Modifica la etiqueda del nodo, con el valor indicado en `value`. Retorna verdadero si la operación es exitosa, o falso en caso contrario.
- `getNeighbors()`. Retorna una lista con los identificadores de los nodos vecinos al nodo.
- `getNeighborsByEdgeLabel()`. Retorna una lista con los identificadores de los nodos vecinos, considerando únicamente aristas cuya etiqueta coincida con la etiqueta indicada en `edgeLabel`.
- `getPEdges()`. Retorna una lista de objetos `PEdge`, que corresponde a la lista de aristas salientes del nodo.
- `getNumPEdges()`. Retorna la cantidad de aristas salientes del nodo.
- `addPEdge()`. Agrega una arista saliente al nodo, con la lista de propiedades `properties`, y cuyo destino es el nodo indicado por `nodeId`.
- `removePEdges()`. Elimina todas las aristas que salen del nodo, con destino al nodo indicado por `nodeId`.
- `voteToHalt()`. Saca de la computación al nodo, hasta que reciba nuevos mensajes.
- `getId()`. Retorna el identificador del nodo.

IV-C. Clase `PEdge`

Pone a disposición del programador un conjunto de métodos públicos aplicables a una arista del grafo.

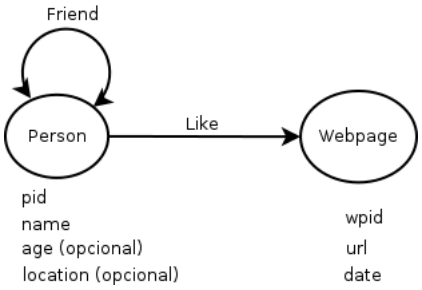

Figura 5. Esquema de datos de la Red Social empleada como caso de uso.

- `getEdgePropertyValue()`. Retorna el valor de la propiedad de la arista individualizada por la llave indicada por `key`. Si la propiedad no existe genera una excepción y retorna `ERROR`.
- `getEdgeLabel()`. Retorna la etiqueta de la arista.
- `getTargetVertexId()`. Retorna el identificador del nodo al que llega la arista.

IV-D. Clase `NodeComputation`

- `compute()`. Se ejecuta en cada *superstep* para cada objeto `PNode`, entregándole una lista iterable de los mensajes enviados en la iteración anterior.

V. Caso de uso & Experimentos

Para ilustrar la forma en que puede utilizarse la interfaz de programación propuesta, presentamos un ejemplo de aplicación en el dominio de las redes sociales. El esquema de datos de la red social elegida se muestra en la Figura 5. El esquema define dos tipos de nodos, *person* y *webpage*; y dos tipos de aristas, *friend* y *like*. Las personas se relacionan con otras personas a través de aristas no dirigidas de tipo *friend*, y se conectan a páginas web con aristas dirigidas de tipo *like*. De esta forma se modela las relaciones de amistad entre personas y de interés de una persona por una página web, respectivamente. Los atributos de una persona son el *pid* o identificador, el nombre, y dos campos opcionales, edad y ubicación. Una página web tiene como atributos su identificador o *wpid*, un URL y una fecha de creación.

El esquema de datos antes descrito proviene de un microbenchmark para bases de datos de grafo denominado GDBench [23]. Este microbenchmark incluye un generador de datos sintéticos llamado GDGenerator, el cual permite crear datos que siguen la estructura de la red social descrita anteriormente. La ejecución de GDGenerator es muy sencilla ya que se encuentra implementado en Java y puede ser descargado desde Github [24]. El método de generación de GDGenerator se basa en la información real publicada por aplicaciones de redes sociales como Facebook, e intenta simular sus características, como por ejemplo una distribución de tipo *power law* para las aristas que corresponden a relaciones de amistad y de interés.

V-A. Pruebas de carga de datos

Para los experimentos de carga de datos se generaron grafos de distintos tamaños, ademas de elegir a CSV como formato de salida para el generador. Por cada grafo generado, GDGenerator produce cuatro archivos CSV conteniendo la información de personas, páginas web, relaciones "friend" y relaciones "like" respectivamente. El formato de cada uno de los archivos CSV de salida se muestra en la Figura 6.

```
// Datos generados del tipo "person"
pid,name,age,location
1,MOHAMMED ROBESON,?,UTAH
2,GOLDA POSTEMA,?,AGNITA
3,PRINCE MONTIS,92,UTAH
...

// Datos generados del tipo "webpage"
wpid,url,creation
501,http://www.site.org/webpage501.html,?
502,http://www.site.org/webpage502.html,2003/8/1
503,http://www.site.org/webpage503.html,2009/1/20
...

// Datos generados del tipo "friend"
id,personId1,personId2
1,1,251
2,1,126
3,2,376
...

// Datos generados del tipo "like"
id,personId,webpageId
3118,1,751
3119,1,626
3120,2,876
...
```

Figura 6. Formato de los archivos de salida producidos por el generador de redes sociales empleado en los experimentos.

Como parte del workflow de Giraph, los datos originales tienen que transformarse a un formato de datos intermedio que pueda ser aceptado por Giraph, pero a la vez compatible con nuestra codificación de *property graphs*. Como parte de las pruebas, se experimentó con dos tipos de formato intermedio. El primer formato intermedio consiste de un único tipo de archivo centrado en vértices. En este caso, cada vértice está acompañado de su lista de adyacencia. Puesto que hay dos tipos de vértices, los datos fueron transformados a dos archivos con la misma estructura. El segundo formato intermedio define dos tipos de archivo, uno para los vértices y uno para las aristas, de forma que los datos se encuentran menos acoplados. Como hay dos tipos de vértice y dos tipos de arista, los datos fueron transformados a cuatro archivos, dos de cada tipo. La Figura 7 muestra unas líneas de ejemplo producto de esta última transformación.

La Figura 8 muestra una comparativa del tiempo necesario para la transformación de los datos de entrada, empleando los dos formatos intermedios descritos anteriormente. Para esto se consideraron cinco tamaños de grafo: 500 mil, 1 millón, 2 millones, 4 millones y 8 millones de nodos. Puede verse cómo el tiempo de transformación es notablemente superior para el caso de un único archivo (primer formato intermedio), pues el algoritmo debe combinar la información de vértices y aristas. Por ejemplo, cuando se procesa una arista no dirigida

```
// Ejemplo de datos para nodos de tipo "person"
1     {"_label":"person","name":"MOHAMMED ROBESON","age":"?","location":"UTAH"}
2     {"_label":"person","name":"GOLDA POSTEMA","age":"?","location":"AGNITA"}
3     {"_label":"person","name":"PRINCE MONTIS","age":"92","location":"UTAH"}
...

// Ejemplo de datos para nodos de tipo "webpage"
501   {"_label":"webpage", "url":"http://www.site.org/webpage501.html","creation":"?"}
502   {"_label":"webpage", "url":"http://www.site.org/webpage502.html","creation":"2003/8/1"}
503   {"_label":"webpage", "url":"http://www.site.org/webpage503.html","creation":"2009/1/20"}
...

// Ejemplo de datos para aristas de tipo "friend"
1     251     {"_label":"friend"}
251   1       {"_label":"friend"}
1     126     {"_label":"friend"}
126   1       {"_label":"friend"}
1     376     {"_label":"friend"}
376   1       {"_label":"friend"}
...

// Ejemplo de datos para aristas de tipo "like"
1     751     {"_label":"like"}
1     626     {"_label":"like"}
1     876     {"_label":"like"}
...
```

Figura 7. Formato de datos de entrada para el caso de uso de la red social. Observe el uso de JSON para codificar los atributos de los nodos de tipo *person* y *webpage*.

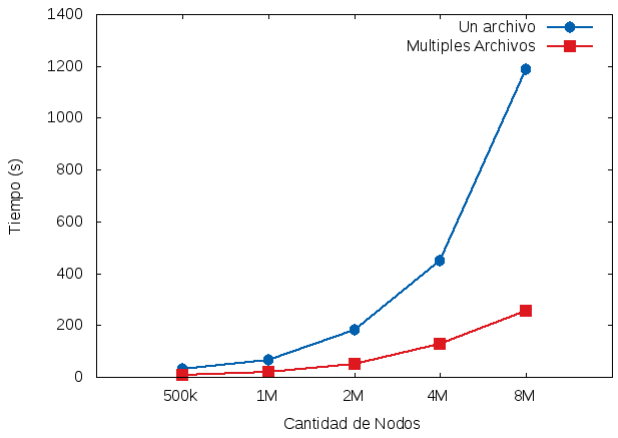

Figura 8. Tiempo de transformación de los datos desde CSV a uno y múltiples archivos de entrada, para la aplicación de la red social.

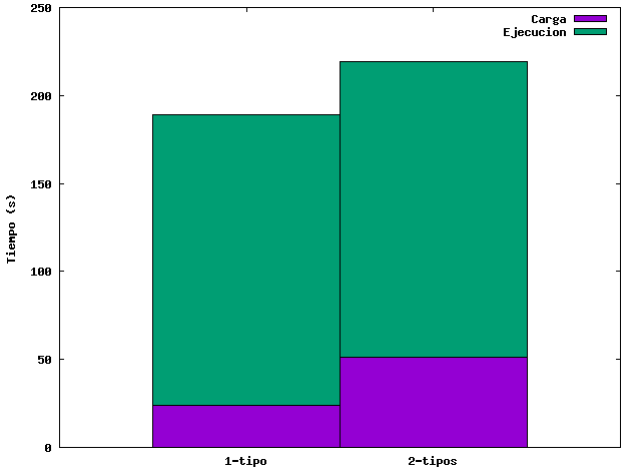

Figura 9. Tiempo de carga de los datos utilizando uno y dos tipos de archivos.

que une los vértices x e y, debe buscarse la línea del archivo que contiene la información de x y agregar ahí la arista $x \rightarrow y$, para luego tener que buscar la línea correspondiente a y para incluir ahí la arista inversa $y \rightarrow x$. En el segundo caso, cuando los datos están particionados, el procedimiento de transformación es más rápido. Además, no es necesario preocuparse por el orden en que se incluyen las aristas, pues Giraph se encarga de combinar los datos durante la carga.

Por otra parte, el tiempo de carga de los archivos intermedios en el sistema de archivos HDFS exhibe un comportamiento inverso. El costo cuando se utiliza un único tipo de archivo (primero formato intermedio) es menor que cuando se utilizan dos tipos (segundo formato intermedio). Esta situación se observa en la Figura 9. No obstante lo anterior, separar los datos de entrada en varios archivos puede ser ventajoso cuando la aplicación no necesita de todas las relaciones o todos los tipos de nodos. En estos casos, una carga parcial de los datos puede hacerse de manera mas sencilla y eficiente.

V-B. *Pruebas de ejecución de consultas*

Para evaluar el desempeño del API se eligió una consulta de prueba que requiere explorar todos los nodos del grafo. En términos generales, la consulta consiste en contar el número de "likes" que han sido emitidos por los amigos de cada nodo

```
public void compute(PNode pNode, Iterable<Text> messages) {
   // Filtra para actuar unicamente sobre nodos person
   if (pNode.getNodeLabel().equals("person")) {
      // En el superStep 0 envia mensaje a todos los vecinos de tipo friend, con la cantidad de likes
      if (getSuperstep()==0) {
         ArrayList<Text> friends = new ArrayList<Text>();
         int likes =0;
         // Para todas las aristas, verifica si es de tipo friend y la agrega al array;
         // si no, la relacion es de tipo like y aumenta el contador.
         for (PEdge pedge : pNode.getPEdges()) {
            if (pedge.getEdgeLabel().equals("friend"))
               friends.add(new Text(pedge.getTargetVertexId()));
            else
               likes++;
         }
         // Agrega a una propiedad del nodo la cantidad de likes que ha realizado a paginas
         pNode.setNodePropertyValue("myLikes", Integer.toString(likes));
         // Envia mensaje a los amigos, con el contador de likes
         sendMessageToMultipleEdges(friends.iterator(), new Text(Integer.toString(likes)));
      }
      else {
         // Suma los likes de todos los amigos y guarda el resultado en una propiedad
         int suma=0;
         for (Text message : messages) {
            suma += Integer.parseInt(message.toString());
         }
         pNode.setNodePropertyValue("friendsLikes", Integer.toString(suma));
      }
   }

   // El nodo se bloquea a esperar por nuevos mensajes
   pNode.voteToHalt();
}
```

Figura 10. Ejemplo de un programa en Giraph que usa la API para *property graphs*. El programa cuenta el número de "likes" que han sido emitidos por los amigos de cada nodo de tipo *person*.

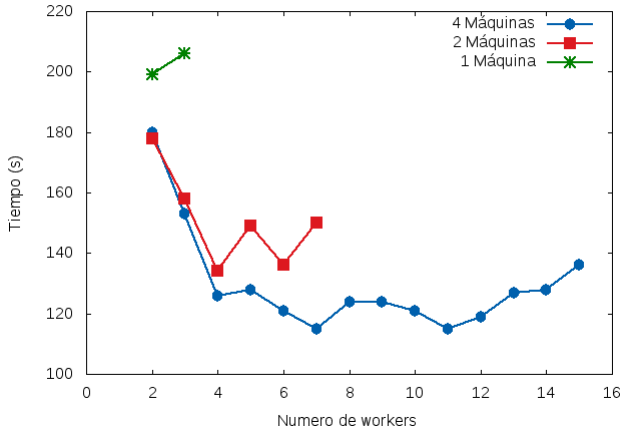

Figura 11. Tiempo de ejecución para la consulta de prueba considerando un grafo de un millón de nodos, pero variando la cantidad de computadores (máquinas) y de *workers*.

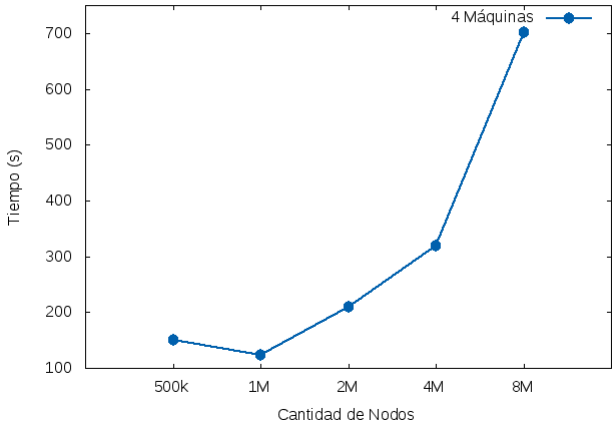

Figura 12. Tiempo de ejecución para la consulta de prueba considerando 7 *workers* sobre 4 computadores, pero variando la cantidad de nodos en el grafo (de quinientos mil a ocho millones de nodos).

de tipo *person*. El código Java que implementa la consulta anterior empleando el nuevo API se muestra en la Figura 10.

Como toda implementación en Giraph, la función `compute()` describe el comportamiento de un nodo a lo largo de una computación que será organizada iterativamente como una sucesión de *supersteps*. El método limita su ejecución únicamente a nodos de tipo `person`, de acuerdo al *label* que contenga. En el primer *superstep* cada nodo de tipo `person` construye un arreglo con los identificadores de sus vecinos, es decir, aquellos nodos para los que existe una arista

saliente de tipo `friend`. Además, cuenta las aristas salientes que son de tipo `like`. Este último valor es almacenado como una propiedad en el nodo, y es enviado a todos los amigos que se almacenaron en el arreglo. En el siguiente *superstep* los nodos reciben los contadores de *likes* de sus vecinos y los suman, almacenando este valor total en una propiedad del nodo.

La Figura 11 muestra los tiempos de ejecución de la consulta de prueba tomando como base de datos un grafo compuesto de un millón de nodos. El experimento se realizó variando el número de computadores y *workers* (procesos) de Giraph. El mejor desempeño se obtuvo con 4 computadores y con las configuraciones de 7 y 11 *workers*. La configuración con 2 computadores obtuvo un desempeño comparable, con 4 y 6 *workers*. Cabe resaltar que las configuraciones con 1 y 2 computadores no pudieron completar la ejecución para un número elevado de *workers*, esto por restricciones de memoria. Por esta razón, el gráfico sólo incluye datos para 2 y 3 *workers* en el caso de 1 computador, y hasta 7 *workers* para el caso de 2 computadores.

En la Figura 12 puede verse la variación del tiempo de ejecución para una configuración con 4 computadores y 7 procesos *workers*, variando la cantidad de nodos en el grafo (desde 500 mil hasta 8 millones de nodos). Puede verse cómo el tiempo de ejecución tiene una tendencia exponencial, como era de esperarse por la complejidad del algoritmo involucrado.

VI. Conclusiones y Trabajo Futuro

Giraph es un framework para el procesamiento de grafos de gran tamaño que exhibe varias ventajas, como la tolerancia a fallas, transparencia, escalabilidad y flexibilidad. No obstante, tiene también varias limitantes. En particular, los vértices y aristas no tienen tipos, y no es posible asociar datos complejos –como propiedades– a vértices y aristas, pues estos elementos tienen únicamente un valor asociado.

En este trabajo presentamos una formalización del modelo de datos de Giraph, junto con un método formal para transformar un Property Graph en un Giraph graph utilizando un formato de codificación de datos basado en JSON. A partir de esto, la API de Giraph fue extendida con métodos que permiten la manipulación y consulta de property graphs.

Para evaluar el API propuesto se eligió una red social como caso de estudio. El esquema de datos de la red social presenta características que fueron fácilmente modeladas a través del nuevo API, en particular la existencia de nodos con metadatos. Esta característica fue fácilmente soportada a través del modelo de datos basado en *property graphs*, ventaja que no presenta el modelo de grafo básico de Giraph.

Para medir la eficiencia del nuevo API, se realizaron experimentos de carga de datos y de ejecución de consultas. Se presentaron dos métodos de carga y se analizo su comportamiento, demostrando que la selección de cada uno de ellos depende de los datos de entrada originales. Para las pruebas de ejecución, se diseño una consulta de prueba que requería explorar toda la red social. Los resultados de ejecutar la consulta mostraron una aceleración aceptable y estable de la plataforma, lo que respalda la aplicabilidad del API propuesto.

Actualmente se está trabajando en emplear el nuevo API en la manipulación y consulta de datos de proteínas modeladas como property graphs. Esperamos que este caso de uso real aporte mayor evidencia respecto de la utilidad de nuestra extensión de Giraph, y nos permita hacer un estudio más profundo de su desempeño.

Agradecimientos

Renzo Angles es financiado por el Núcleo Milenio Centro de Investigación de la Web Semántica (Millennium Nucleus Center for Semantic Web Research) bajo convenio Nro. NC120004.

Referencias

[1] S. Sakr and E. Pardede, *Graph Data Management: Techniques and Applications*, 1st ed. Information Science Reference - Imprint of: IGI Publishing, 2011.

[2] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1804–1815, Aug. 2015.

[3] Apache Foundation, "Apache Hadoop," http://hadoop.apache.org, online; accessed: May 30th, 2016.

[4] "Apache Hadoop NextGen MapReduce (YARN)," http://hadoop.apache.org/docs/current/hadoop-yarn, online; accessed: May 30th, 2016.

[5] "Stratosphere - next generation big data analytics platform," http://stratosphere.eu, online; accessed: May 30th, 2016.

[6] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A Resilient Distributed Graph System on Spark," in *First International Workshop on Graph Data Management Experiences and Systems (GRADES)*. ACM, 2013, pp. 1–6.

[7] Y. Zhao, K. Yoshigoe, M. Xie, S. Zhou, R. Seker, and J. Bian, "Evaluation and analysis of distributed graph-parallel processing frameworks," *Journal of Cyber Security and Mobility*, vol. 3, no. 3, pp. 289–316, July 2014.

[8] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-scale Graph Processing," in *Proc. of the International Conference on Management of Data (SIGMOD)*. ACM, 2010, pp. 135–146.

[9] Apache Foundation, "Apache Giraph," http://giraph.apache.org, online; accessed: May 30th, 2016.

[10] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, 2012.

[11] A. Khan and S. Elnikety, "Systems for Big-Graphs," in *Proc. of the 40th International Conference on Very Large Data Bases (VLDB)*, 2014.

[12] T. White, *Hadoop: The Definitive Guide (4th. ed)*. O'Reilly, 2015.

[13] S. Yang, N. D. Spielman, J. C. Jackson, and B. S. Rubin, "Large-scale neural modeling in mapreduce and giraph," in *IEEE International Conference on Electro/Information Technology*, June 2014, pp. 556–561.

[14] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From think like a vertex to think like a graph," *Proc. VLDB Endow.*, vol. 7, no. 3, pp. 193–204, Nov. 2013.

[15] M. Nolé and C. Sartiani, "Processing regular path queries on giraph," in *Proceedings of the Workshops of the EDBT/ICDT*, ser. CEUR Workshop Proceedings, vol. 1133, 2014, pp. 37–40.

[16] H. F. Liu, C. T. Su, and A. C. Chu, "Fast quasi-biclique mining with giraph," in *Proc. of the IEEE International Congress on Big Data*, June 2013, pp. 347–354.

[17] B. Elser and A. Montresor, "An evaluation study of BigData frameworks for graph processing," in *Proceedings of the IEEE International Conference on Big Data*, 2013, pp. 60–67.

[18] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin, "An Experimental Comparison of Pregel-like Graph Processing Systems," *Proc. VLDB Endow.*, vol. 7, no. 12, pp. 1047–1058, Aug. 2014.

[19] Z. Cai, Z. J. Gao, S. Luo, L. L. Perez, Z. Vagena, and C. Jermaine, "A comparison of platforms for implementing and running very large scale machine learning algorithms," in *Proc. of the International Conference on Management of Data (SIGMOD)*. ACM, 2014, pp. 1371–1382.

[20] C. Martella, R. Shaposhnik, and D. Logothetis, *Practical Graph Analytics with Apache Giraph*. Apress, 2015.

[21] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*, 1st ed. O'Reilly Media, June 2013.

[22] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[23] R. Angles, A. Prat-Pérez, D. Dominguez-Sal, and J.-L. Larriba-Pey, "Benchmarking database systems for social network applications," in *First International Workshop on Graph Data Management Experiences and Systems (GRADES)*. ACM, 2013, pp. 15:1–15:7.

[24] R. Angles, "A Micro-benchmark for Benchmarking Graph Databases based on a Social Network Data Model ," https://github.com/renzoar/GDBench, online; accessed: May 30th, 2016.