# Resumen del artículo sobre Pregel

En primer lugar, conviene recordar que en un grafo, a los nodos, o “entidades”, se les llama vértices (“vertex”), y a las líneas que unen a los vértices, aristas (“edges”).

Así, este artículo presenta un modelo de procesamiento orientado a grafos masivos con miles de millones de vértices, en los que los métodos tradicionales son ineficientes. En concreto, el modelo de Pregel propone el paradigma de “piensa como un vértice” (“thinking like a vertex”), en el que los programas se presentan como una serie de iteraciones en las que un vértice recibe mensajes enviados en la iteración previa, modifica su estado propio, y envía mensajes para la próxima iteración.

La idea fundamental es que en el modelo de Pregel, a pesar de tener un grafo interconectado, cada vértice funciona de manera independiente, gracias al paso de mensajes, lo que permite paralelizar el procesamiento y hacerlo muy eficiente.

## Introducción

Los modelos actuales de procesado presentan muchos problemas a la hora de tratar con los grafos masivos que tenemos en la actualidad (webs de Internet, redes sociales, etc.), por ejemplo, poca localidad en el acceso a memoria, poco procesado por vértice, o cambios en el grado de paralelismo durante la ejecución. Así, las opciones actuales son limitadas:

* Montar una infraestructura distribuida propia, lo que supone un gran esfuerzo y poca reusabilidad.
* Utilizar un paradigma de procesado distribuido inadecuado para el caso particular de los grafos, como Map-Reduce.
* Utilizar alguna librería especializada en grafos, pero sin computación distribuida, como BGL.
* Ídem a la anterior, pero con computación distribuida, como CGMgraph, que presentan, entre otros, problemas de tolerancia a fallos.

Pregel pretende ser una plataforma con una API que funcione para grafos masivos, con computación distribuida, afrontando los problemas típicos de este paradigma.

A alto nivel: el fundamento consiste en lo expresado anteriormente. Existen una serie de iteraciones, llamadas “súper-pasos” (“supersteps”), durante las cuales el framework, con paralelización, invoca a una función definida por el usuario que define el comportamiento del vértice V. Además, modifica su estado, lee mensajes enviados en el súper-paso S-1, y envía mensajes a otros vértices que se recibirán en S+1. Normalmente se envían a los vértices directamente conectados entre sí, aunque técnicamente estos pueden ser arbitrarios.

Así, el usuario se centra en acciones locales, procesando los ítems de forma independiente, y luego el sistema compone las acciones para un conjunto de datos masivos, de manera similar a Map-Reduce. Además, como el orden de ejecución en S no es relevante, y los mensajes siempre van de S a S+1, funciona bien en el contexto de computación distribuida. Gracias al sincronismo del modelo se evitan problemas de la computación asíncrona, como las condiciones de carrera. De hecho, con la capacidad de paralelización suficiente, equilibrando la carga de vértices entre las distintas máquinas, el rendimiento debería ser similar al de un sistema asíncrono.

## Modelo de computación

Cada **vértice** se identifica unívocamente mediante un String, y está asociado un valor modificable definido por el usuario. Las **aristas** están dirigidos (son unidireccionales, van de un vértice origen a un vértice destino), y consiste en dos elementos: un valor modificable definido por el usuario, y un identificador del vértice destino. Generalmente, el procesado consiste en la llegada de un input, una serie de súper-pasos separados por puntos de sincronización globales hasta que acaba el algoritmo, y un output.

La función definida por el usuario, que se aplica en a los vértices en paralelo, expresa la lógica del algoritmo utilizado. Así, cada vértice puede modificar su estado, o el de las aristas salientes, recibir mensajes que fueron enviados en S-1, enviar a su vez mensajes que se recibirán en S+1, e incluso modificar la topología del grafo. De este modo, se dice que las aristas no son ciudadanos de primera clase en el modelo de Pregel, ya que no tienen computación asociada.

Los vértices alternan entre dos estados, Activo en Inactivo, que alternan según la siguiente máquina de estados:

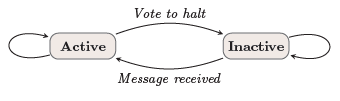


Ilustración 1. Máquina de estados de los vértices.

Básicamente, el algoritmo finaliza cuando todos los vértices votan la parada, y en el súper-paso 0, todos los vértices empiezan en activo. Para pararse, el propio vértice lo vota de forma explícita, no es automático. Cuando un vértice está inactivo, no hace nada, y no vuelve a activarse hasta que le llega un mensaje de otro vértice (se activa de forma externa). Mientras está parado, el framework no lo ejecutará. Se considera que el algoritmo completo acaba cuando todos los vértices están inactivos, y no hay mensajes en tránsito.

La salida del programa es el conjunto de valores explícitamente sacados por los vértices. A menudo es un grafo isomorfo con respecto a la entrada, aunque puede no ser así, ya que se contempla, la adición y eliminación de vértices durante la computación.

Hay que tener en cuenta que es un modelo de paso de mensajes puro, sin lecturas remotas ni nada similar, ya que es suficientemente expresivo para que no sea necesario, y más eficiente. Una máquina mantendrá a sus vértices con sus aristas salientes, y solo usará la red para enviar mensajes.

Todo esto lo hace mucho más eficiente frente a Map-Reduce, que, al ser funcional, requiere pasar el estado completo del grafo de una fase a la siguiente, lo que implica mayor comunicación, y serialización, en definitiva, mayor complejidad e ineficiencia.

## La API en C++

Básicamente, existe una clase *Vertex*, que tiene tres valores: vértices, aristas, y mensajes. Luego el desarrollador la implementa para su caso particular. Hay un método *Compute()* en el que el usuario implementa su función, que es la que se aplicará a cada vértice. Como la visibilidad de los cambios queda restringida a al vértice modificado, no aparecen condiciones de carrera.

### Paso de mensajes

Los mensajes consisten en un valor, el nombre del vértice destino. Un vértice puede enviar un número de mensajes cualquiera en S, y estarán disponibles cuando se llame a *Compute()* en S+1. Se garantiza la entrega de los mensajes, y que no van a estar duplicados, pero no el orden.

Los vértices pueden conocer los identificadores del resto bien gracias a los mensajes, bien de forma implícita. Si un vértice destino no existe, ha de existir un manejador definido por el usuario que, por ejemplo, cree el vértice.

### Combinadores

Cuando se manda un mensaje a un vértice que, casualmente, no se encuentra en la misma máquina, es necesaria una comunicación, con el consumo de tiempo asociado. Así, por ejemplo, si la idea original es enviar un montón de valores, pero en realidad lo único que importa es la suma de todos ellos, el usuario puede definir un combinador que calcule dicha suma, y la envíe, de manera que varios mensajes se “combinan” en uno solo.

### Agregadores

Los agregadores son un mecanismo de comunicación global y monitorización. Básicamente, cada vértice provee de un valor a un agregador en S, el sistema combina esos valores mediante reducción (mínimo, máximo, o suma), y envía el resultado a todos en S+1.

Estos entes son útiles para generar estadísticas, pero también para coordinación global (por ejemplo, algo de *Compute()* no se ejecuta hasta que un agregador *and* determina que todos los vértices cumplen cierta condición).

### Mutación de la topología

La función *Compute()* de un usuario también puede solicita la adición o eliminación tanto de vértices como de aristas. Sin embargo, esto genera problemas, ya que dos vértices distintos pueden llevar a cabo solicitudes conflictivas en S, como la creación de un vértice con diferente valor inicial. Esto se resuelve con ordenación parcial y manejadores.

Así, teniendo en cuenta que la mutación solicitada en S pasa a ser efectiva en S+1 (como los mensajes), primero se llevan a cabo las eliminaciones, y dentro de ellas, primero las aristas, y después los vértices. Después, se ejecutan las adiciones, y dentro de ellas, primero los vértices y luego las aristas. Esta ordenación parcial garantiza resultados deterministas en la mayoría de casos. Cualquier otro conflicto, como el del ejemplo, ha de resolverlo el usuario implementando el manejador que él decida, es su problema, no el del modelo de Pregel, en aras de su simplificación.

Hay que tener en cuenta que el mecanismo de coordinación global es perezoso, ya que las mutaciones globales no requieren coordinación hasta el punto en el que se aplican. Además, se permiten mutaciones locales, es decir, un vértice puede decidir eliminar sus propios aristas salientes, o, incluso, borrarse a sí mismo, sin que se lo ordene nadie a través de un mensaje.

### Input y output

El formato de los grafos es muy variado, por ejemplo, ficheros de texto, o vértices almacenados en una base de datos relacional. Sin embargo, Pregel desacopla su interpretación de la computación en sí del grafo, es problema del usuario.

## Implementación

Pregel se diseñó para la arquitectura de Google, con miles de PCs organizados en racks con anchos de banda masivos, interconectados, pero distribuidos geográficamente.

### Arquitectura básica

Pregel divide los gráficos en particiones, cada una de las cuales consiste en un conjunto de vértices con sus aristas salientes. En función de los identificadores de vértices, estos se asocian a una partición u otra. La función de particionamiento se basa en *hash*.

La ejecución de un programa Pregel consiste en las siguientes fases:

1. Muchas copias del programa del usuario empiezan a ejecutarse en la granja de máquinas. Una de ellas es la copia maestra, a la que no se le asigna ninguna porción del grafo, sino que se ocupa de la coordinación de los trabajadores, que utilizan el servicio de nombrado del sistema de gestión del clúster para descubrir la localización del maestro, y enviarle sus mensajes de registro.
2. El maestro determina cuántas particiones habrá en el grafo, y asigna una o más a cada trabajador (el número exacto lo puede definir el usuario). Si los trabajadores tienen más de una partición, mejora el rendimiento, gracias al paralelismo y el equilibrado de la carga. Cada trabajador, que conoce el conjunto de asignaciones de todos los demás, es responsable de mantener su trozo del grafo, ejecutando *Compute()*, y del paso de mensajes con el resto de trabajadores.
3. El maestro asigna una parte de los datos de entrada del usuario a cada trabajador. La entrada se trata como un conjunto de registros, cada uno de los cuales contiene un número arbitrario de vértices y aristas. La división de estos datos es ortogonal a las particiones del grafo, y normalmente se basa en los límites de los ficheros. Si un trabajador carga un vértice que pertenece a su propia sección del grafo, las estructuras de datos se actualizan inmediatamente. Si no, se encola un mensaje para el trabajador que posea el vértice. Una vez se ha cargado toda la entrada, todos los vértices se marcan como Activos.
4. El maestro ordena a cada trabajador que ejecute un súper-paso. Cada trabajador itera sobre sus vértices activos, usando una hebra para cada partición. El trabajador invoca a *Compute()* para cada vértice activo, entregando los mensajes que se enviaron en el súper-paso previo. El envío de mensajes es asíncrono, para solapar la comunicación y la computación, pero todos los mensajes se entregan antes de que acabe el súper-paso. Cuando el trabajador acaba, le dice al maestro cuantos vértices estarán activos en el siguiente super-paso. Todo esto se repite mientras haya vértices activos, o mensajes en tránsito.
5. Cuando acaba la computación, el maestro ordena a los trabajadores que guarden su porción del grafo.

### Tolerancia a fallos

Pregel define un mecanismo de puntos de control: al comienzo del súper-paso, el maestro ordena a los trabajadores que guarden el estado de sus particiones en almacenamiento persistente, incluyendo los valores de los vértices, lo valores de los aristas, y los mensajes entrantes.

Los fallos se detectan mediante mensajes de ping que el maestro envía a los trabajadores. Si un trabajador no recibe un ping tras cierto tiempo, finaliza su procesado. Si el maestro no recibe un pong, marca al trabajador como fallido.

Si falla uno o más trabajadores, el estado actual de las particiones se pierde. El maestro ha de reasignar las particiones a los trabajadores disponibles, y estos cargarán el estado del punto de control disponible más reciente, que puede ser bastante más antiguo que S, lo que implica que la recuperación de la partición ha de repetir varios súper-pasos.

En el futuro, se pretende implementar la recuperación confinada, en la que los trabajadores generan logs que pueden facilitar el proceso de recuperación, aunque este mecanismo es exclusivo para algoritmos deterministas.

### Implementación de trabajadores

Un trabajador se encarga de mantener el estado de su porción del grafo en memoria. Este puede ser un mapa que asocie un ID de vértice a su estado, que consiste en su valor actual, una lista de aristas salientes, una cola con mensajes entrantes, y un flag que indique si está activo.

Cuando un trabajador procesa sus vértices en S, simultáneamente, en otra hebra, recibe mensajes de otros trabajadores ejecutando el mismo súper-paso. De este modo, hay dos copias para el flag y la cola de mensajes, una para S, y otra para S+1.

### Implementación del maestro

El maestro se ocupa de la coordinación de las actividades de los trabajadores, a los que asigna identificadores únicos cuando se registran. El maestro posee una lista con los trabajadores vivos, que incluye sus identificadores, su información de contacto, y la porción de grafo asignada. Es notable que el tamaño de la información mantenida por el maestro depende de la cantidad de particiones, no de los vértices ni las aristas.

La mayoría de operaciones del maestro acaban en barreras: el maestro envía peticiones a los trabajadores que han de ejecutar una operación, y espera a que a que todos le respondan.

El maestro también posee estadísticas, como el tamaño del grafo, su estado, histogramas, etc., que pueden ser monitorizadas por el usuario, ya que el maestro se ejecuta como un servidor HTTP.

### Agregadores

Los trabajadores poseen instancias de los agregadores, y las alimenta mediante valores combinados.

## Aplicaciones

El modelo de computación de Pregel puede aplicarse para diferentes casos de uso que apliquen a diferentes algoritmos, por ejemplo, la clasificación de páginas en Google, encontrar los caminos más cortos en una red, semi-clustering, etc.

# Resumen del artículo de Raphtory

Un grafo temporal captura el desarrollo de las relaciones de los datos a lo largo del tiempo, lo que acompaña de forma natural a arquitecturas de streaming. Así, el artículo describe un modelo de grafo temporal que puede actualizarse a través de eventos de streaming. En concreto, Raphtory es un sistema de gestión distribuido para grafos temporales, que mantiene en memoria la historia completa del grafo, lo que permite insertar eventos en streaming directamente al modelo, sin necesidad de un orden centralizado. Provee de una API que permite ejecutar análisis sobre la versión mas reciente del grafo y análisis sobre la historia completa, ejecutadas en paralelo con la ingestión de datos.

## Introducción

Los grafos siempre han supuesto un problema destacado de rendimiento, pues suelen requerir el procesado de grandes cantidades de datos, y las relaciones entre ellos. Sin embargo, aunque se han propuesto diferentes modelos para mitigar esta inconveniencia, como Pregel, si hablamos de gráficos temporales, no hay tantas soluciones donde elegir. Raphtory pretende ser un modelo que provea de escalabilidad a estos grafos temporales.

En parte, sigue el paradigma de “piensa como un vértice”, ya que guarda el historial de cambios de cada vértice y arista, en lugar del grafo completo. De manera similar a Pregel, el grafo se divide en un conjunto de particiones, que gestiona el histórico de los vértices y aristas asignados, e inserta actualizaciones en el orden cronológico correcto, sin un mecanismo central de sincronización.

El historial del grafo se gestiona comprimiendo y desplazando los cambios más antiguos, que se almacenan en almacenamiento a largo plazo hasta que son requeridos para análisis, de manera opaca.

## Modelo de grafos temporales

A la hora de tratar con grafos temporales, es necesario presentar un modelo formal.

### Modelo propuesto

Un grafo estático es una dupla de la forma <V, E>, donde V es el conjunto de vértices, y E, el de aristas, expresados a su vez como una dupla <vi, vj>, que es direccional. No puede haber dos aristas iguales, pero sí pueden formar un bucle: (<vi, vj>). Se llama entidades del grafo, Y, a V ∪ E. Finalmente tenemos las claves (K), que se asocian a una propiedad del grafo p.

Si el grafo es temporal, lo que tenemos es G(t) = <V(t), E(t)>. La variable t varía entre t0, que es el instante en el que se crea el grafo, y tn, que es el instante en el que se produjo el cambio más reciente. G(tn) es el “grafo en vivo”. Así, en un instante arbitrario G(t), tenemos el grafo con todos los cambios aplicados hasta el instante t. De este modo, un grafo temporal engloba todos los G(t) desde t0 hasta tn.

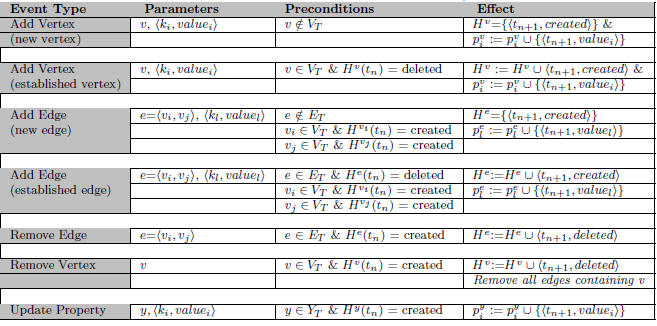
Además, también tenemos la historia de una entidad, Hy, que engloba todos los eventos que afectan a esa entidad, por ejemplo: (ti, creado) 🡪 la entidad y se creó en el instante ti. Esto permite la ordenación temporal de eventos.

Por último, ahora las propiedades en un instante i contienen todos los valores asociados a la entidad hasta llegar a ese instante: piy = {<tj, valuej>, <tk, valuek>, …, <ti, valuei>}.

### Semántica de las actualizaciones

En un grafo temporal, las actualizaciones son un stream de eventos, de la forma: {<t1, a1>, …, <tn, an>}, donde las acciones a son las de la Tabla 1, y t son los instantes en los que van ocurriendo. En general, como es lógico, las acciones se clasifican en tres categorías: adición de entidades, eliminación de entidades, y actualización de entidades. También hay que tener en cuenta que el stream no está acotado, van llegando más acciones, por ejemplo, en tn+1.

Tabla . Posibles acciones en un grafo temporal



#### Adición de entidades

Cuando se añade un vértice, primero se comprueba si ya existe en el conjunto V. Si no es el caso, se añade, y se le asocia una historia. Si sí es el caso, se comprueba si había sido borrado (así es, un vértice borrado sigue perteneciendo a V, este contiene todos los vértices que han existido en el grafo temporal, y lo mismo pasa con E, y, en general, con Y), y se añade un evento nuevo a su historia indicando la creación, a no ser que su estado actual sea “creado”, en cuyo caso la actualización se descarta.

Para las aristas pasa lo mismo, teniendo en cuenta que, si no existe alguno de los dos vértices que pretende unir, se descarta la actualización.

#### Eliminación de entidades

Para eliminar una entidad, esta ha de estar presente en el grafo temporal. Y cuando se borra, no se elimina ninguna información, sino que se añade un evento a su historia con el estado “borrado”. En el caso particular de un vértice, también hay que borrar todas las aristas que salen o entran en él.

#### Propiedades de las entidades y actualizaciones

Las propiedades de las entidades se van estableciendo y actualizando al crear vértices y aristas, así como al actualizarlos. En general, las propiedades se actualizan como piy := piy ∪ <tn+1, valori>, lo que permite actualizar varias propiedades juntas. Hay que tener en cuenta que las entidades pueden tanto cambiar los valores de propiedades ya establecidas, como ganar nuevas propiedades. En cuanto a la lógica de las actualizaciones, las propiedades cambian si la entidad existe, no pueden variar mientras la entidad haya sido borrada, etc.

## Retos de la distribución

Es obvio que, para el procesado de grafos temporales de cierto tamaño, es necesaria la escalabilidad.

### Distribución del modelo y mantenimiento

No es suficiente con la escalabilidad vertical, también hay que aplicar la escalabilidad horizontal. Si tenemos varias máquinas, es necesario dividir el grafo, lo cual, en grafos temporales, supone un reto adicional, ya que no solo hay que pensar en la localidad espacial de los vértices del grafo, sino también en la localidad temporal de los eventos.

Además, para minimizar las inconsistencias, las máquinas del sistema distribuido han de estar continuamente sincronizando su estado. La ordenación temporal de eventos también es relevante. Por ejemplo, si llega la creación de una arista antes de la creación del vértice, la primera se descartaría. Finalmente, también hay que pensar en la gestión de memoria. ¿Qué parte de la historia ha de mantenerse constantemente, y qué parte puede desplazarse a almacenamiento persistente para ser analizado más tarde?

### Análisis

El modelo de procesado ha de habilitar un análisis continuo del grafo, ya que, si no, es como si los datos se estancaran a la hora de entrar. Tampoco se puede ir demasiado rápido, para evitar inconsistencias.

Además, nos puede interesar analizar el grafo en un instante distinto al más reciente, lo que implica una recuperación de datos que idealmente ha de ser eficiente, y opaca para el usuario. También idílicamente, si el usuario cambia el código del algoritmo de análisis aplicado al grafo, este no debería tardar demasiado tiempo en volver a estar disponible.

Ya sea un análisis continuo, o sobre una instantánea, siempre se trabaja con una “Visión Aplanada” (*Flattened Vision*), es decir, el grafo temporal en un instante de tiempo concreto, como un grafo estático.

## Raphtory

Raphtory pretende ser el sistema que aborde todos estos retos, manteniendo un grafo temporal en un conjunto de particiones, e ingiriendo y procesando actualizaciones paralelas en tiempo cuasi-real.

Raphtory se basa en el modelo de actores, típico, por ejemplo, de Akka, en el que estos actores son la unidad básica de computación, una especie de hebras, pero que no comparten ningún tipo de estado común, sino que todas sus interacciones se basan en el paso de mensajes.

La arquitectura general de Raphtory es la que vemos en la siguiente figura:

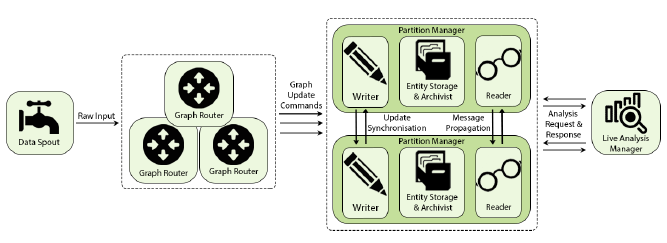


Ilustración . Arquitectura general de Raphtory.

Los actores que componen el núcleo de Raphtory son los *Graph Routers* (GR), y los *Graph Partition Manager* (GPM). Básicamente, los GR se asocian a un stream de entrada, y convierten los datos en crudo en uno de los tipos de actualizaciones descritas previamente. A continuación, los GPM se ocupan de la entidad afectada, pues son ellos los que poseen una partición del grafo. Conforme llegan las actualizaciones del conjunto de GR, los GPM las van insertando en las historias de las entidades afectadas, en el orden cronológico correcto. Esto elimina la necesidad de la sincronización centralizada, ya que los comandos se ejecutan en cualquier orden de llegada, pero siempre generan la misma historia (por ejemplo, aunque un borrado me llegue antes que una adición, gracias a las “t”, se insertan en orden cronológico en la historia). Los GPM también poseen unos umbrales de uso de memoria, de manera que, una vez se alcanzan, toda la historia hasta ese punto se transfiere a almacenamiento offline, lo cual también sirve como mecanismo de back-up.

Los *Live Analysis Manager* (LAM) permiten ir analizando el grafo, pues difunden peticiones a los GPM, que ejecutan el algoritmo correspondiente en súper-pasos del tipo *Bulk Synchronous Parallel*, y devuelven los resultados. Puede haber varios LAM trabajando en paralelo.

### Graph Router

Los GR son los puntos que ingieren los datos y eventos en crudo, y los transforman en operaciones de actualización de los grafos. Las operaciones extraídas se reenvían al GPM que almacena la partición que contiene la entidad afectada. Estos comandos reenviados poseen una marca temporal, que es la que permite a los GPM ordenarlos cronlógicamente. Como ya se ha comentado antes, este mecanismo de marcas evita la sincronización temporal, y proporciona bastante libertad al GR, que puede leer los eventos en crudo en un orden arbitrario, pues se ordenan más tarde.

Las marcas temporales deberían obtenerse de algún campo propio de los campos en crudo, pero si este no existiera, utilizan el reloj interno con PTP para generarlas, lo que restringe un poco más el orden temporal, pues hay que seguir el orden de ingestión. Los GR trabajan de forma totalmente independiente, con lo que pueden añadirse o eliminarse a discreción.

Puede haber varios GR conectados al mismo DS, de manera que se pueden generar diferentes grafos partiendo de un mismo conjunto de datos, o al revés, un GR con varios DS para conformar un solo grafo con varios conjuntos de datos.

Los mensajes entre los GR y los GPM llevan marcas de agua, cuándo se llevó a cabo la actualización más reciente, y garantizar que todo está sincronizado antes de empezar un análisis.

#### Ventanas temporales y declive de propiedades

Los GR no tienen estado, es decir, envían los comandos, y se desentienden de ellos. Sin embargo, tienen algo parecido, las ventanas temporales, esto es, un período de tiempo durante el cual todos los vértices y aristas del grafo en vivo han de haber sido creados o actualizados. Cualquier entidad fuera de la ventana se considera fuera del grafo, y un evento de borrado se añade a su historia enviando el comando al GPM correspondiente. Esto es muy útil para el análisis, por ejemplo, permitiendo eliminar a usuarios inactivos del grafo de una red social.

Para obtener esta funcionalidad, los GR registran todas las entidades que extraen de los datos en crudo, y la última vez que fueron vistas. Como los GR son independientes, es posible que uno de ellos no vea una entidad en una ventana, y mande eliminarla, mientras que otro sí la vea. Estos conflictos son resueltos por los GPM, que ignorarán el comando de borrado al recibir eventos de la entidad procedentes de otro GR.

El declive de propiedades es similar a la ventana, pero en lugar de ordenar una eliminación, simplemente se disminuye el valor de cierta propiedad.

#### Modelado de fuentes de datos

Las fuentes se modelan como *Data Spouts* (DS). Una fuente de eventos puede variar desde una base de datos, hasta un sistema de encolado de mensajes. Así, los DS realizan la conexión inicial, consumiendo duplas y eventos, y distribuyéndolos a los GR de forma estandarizada en un *Round-Robin*. Los DS también son completamente independientes, pudiendo trabajar en paralelo.

### Graph Partition Manager

Los GPM almacenan una partición del grafo completo, con sus vértices y aristas, cada uno de los cuales tiene sus propiedades. Además, se ocupan de mantener las historias actualizadas, completar peticiones de análisis, y llevar a cabo back-ups incrementales para las entidades. Para ello, delegan en sus Escritores (gestionan las actualizaciones sobre el estado del grafo, insertándolas en la historia de las entidades afectadas), Lectores (gestionan las peticiones de análisis, ejecutando el algoritmo provisto, y devolviendo los resultados), y los Archivistas (persisten los datos nuevos en almacenamiento permanente, y archivan las entidades más antiguas para aliviar las restricciones de memoria).

En general, el guardado de las entidades se basa en funciones de *hash*.

#### Almacenamiento de entidades con Singleton

El modelo de almacenamiento sigue el patrón *singleton*, de manera que si lectores, escritores, y archivistas, han de acceder a las mismas entidades, el entrelazado de operaciones es seguro. Las entidades dentro del *singleton* son objetos con metadatos, historia estructural, y un mapa con las propiedades asociadas. La historia se mantiene con un árbol rojo-negro, al igual que las propiedades con la clave correspondiente.

En Raphtory, los vértices están completamente almacenados en una partición, y las aristas se gestionan con el GPM guardando su vértice origen. Si el destino también está en la misma partición, se dice que la arista es local. En caso contrario, es *split*. En este caso, el GPM almacena una copia fantasma de la arista, guardando el vértice destino.

#### Escritor

Se ocupan de todas las actualizaciones, que proceden tanto de las operaciones extraídas por los GR, como de los mensajes de sincronización de otros GPM. Dichas actualizaciones se guardan en una cola ordenada para que el *Singleton* de almacenamiento de entidades lo procese, generando los metadatos necesarios. Las operaciones ocurren de este modo:

* **Adición de entidades**: como hemos dicho antes, al añadir un vértice, si no existía, se instancia, comenzando su historia y estableciendo su mapa de prioridades. Si sí existía, se inserta un evento de creación en su historia, por si se hubiera enviando un comando de eliminación previamente. Las aristas funcionan de forma similar, pero se generan unos vértices plantilla para que la arista no quede suelta. Si la arista es *split*, el GPM propaga los comandos al GPM con el vértice destino, solicitando la cración de copias fantasma.
* **Eliminación de entidades**. Funcionan del mismo modo que la adición, añadiendo un evento de eliminación a la historia, en cualquier caso, y generando las plantillas cuando sea necesario. Hay que tener en cuenta también que, cuando se elimina un vértice, también hay que suprimir sus aristas asociadas, insertando un evento de borrado en sus historias. Esto puede generar condiciones de carrera. Para solucionarlo, es obligatorio que los eventos de eliminación de los vértices adyacentes se inserten también en las historias de las aristas cuando estas se crean. De este modo, aunque una arista se salte la eliminación de un vértice, la información sigue estando presente.

#### Archivista

Los archivistas reducen la carga de memoria en el sistema, en las máquinas que implementan los GPM, ejecutando comprobaciones en las entidades almacenadas para comprimir la historia antigua y sacarla a un almacenamiento secundario. Como la historia es “maleable”, se puede consumir mucha memoria de forma ineficiente. Por ejemplo, tenemos la siguiente historia: H = {<t5, creado>, <t4, creado>, <t2, creado>, <t1, creado>}. Si llega un comando de borrado en t3, varios de los eventos pasan a ser irrelevantes. Así, el archivista, a la hora de guardar esta historia, una vez se asegura de que, al transcurrir cierta ventana temporal, que no es estática, ya no va a llegar ningún evento relevante, la comprime como: H = {<t4, creado>, <t3, borrado>, <t1, creado>}.

El tamaño de la ventana se fija como un porcentaje que pasa entre el evento más antiguo en memoria, y el más nuevo, normalmente del 90 %. Así, si el intervalo son 10 minutos, se pueden comprimir 9.

Como hemos dicho, la historia “antigua” se guarda en almacenamiento secundario persistente, gestionado mediante Cassandra.

A la hora de almacenar historia, una vez ha decidido cuánto “tiempo” se va a comprimir, el archivista elimina cierto intervalo temporal de eventos, en base a otro porcentaje, normalmente el 10 %, llamado profundidad temporal.

#### Lector

Son el motor de procesado de Raphtory, ejecutando las funciones definidas por el usuario en las entidades dentro de su partición. No tienen estado, y trabajan en base a los analizadores que envían los LAM.

El análisis sigue un modelo basado en vértices, parecido a Pregel: cuando se recibe un analizador, se completa un súper-paso, y se informa al LAM, devolviendo unos resultados parciales.

Para acceder a las entidades del grafo, el LAM debe utilizar un *Graph Retrieval Proxy* (GRP), que se ocupa de enmascarar la complejidad de acceder a entidades ya archivadas, y garantiza que solo el escritor puede modificar el estado del grafo. Al ejecutar un análisis sobre el grafo en vido, el GRP devuelve el estado más reciente de los vértices, y cualquier mensaje recibido en el súper-paso previo. Esto se itera hasta que se completa el algoritmo.

Cuando se prefiere analizar un grafo en un instante de tiempo t, se recupera de la memora, primaria o secundaria, el estado del grafo. En concreto, se clona el mapa de vértices/aristas de las particiones, y se iteran, filtrando las entidades creadas después de t, o borradas en t.

Como el archivista de cada uno de los GPM puede tener su propia profundidad temporal (no todas las máquinas han de ser iguales), cada lector ha de generar el snapshot de la partición del grafo de su GPM, aunque el proceso es opaco al usuario.

#### Live Analysis Manager

El LAM permite al usuario interactuar con el grafo, crear instantáneas, y supervisar los análisis. Cada LAM es responsable de la ejecución de un solo Analizador, registrando en qué síepr-paso se encuentra el algoritmo, qué funciones ha de ejecutar el Lector, y cuáles son las condiciones de terminación, en tres fases: preparación, análisis, y finalización.

En la preparación, se difunden peticiones a los lectores, para que cumplan cualquier pre-requisito. Cuando el Analizador recibe respuesta de todos los lectores, pasa al análisis, difundiendo la ejecución del primer súper-paso, que consistirá en una función definida por el usuario para ejecutarse en cada vértice. Cuando todos los lectores notifiquen al LAM, este agrega los resultados, y decide si se ejecuta otro súper-paso, o si el análisis ha finalizado, en cuyo caso se entra en la fase de finalización, en la que se eliminan todas las instantáneas.

Además de este análisis basado en ciclos, el LAM también soporta análisis perpetuo, sin fase de finalización, aunque en este caso los resultados han de considerarse aproximados, ya que ocurre en paralelo con mutaciones del grafo.

### API

Raphtory proporciona una API con una serie de objetos y funciones que emulan el modelo descrito, y que han de ser sobrescritas por el usuario para implementar la funcionalidad deseada.

# Funcionamiento interno

La arquitectura de Raphtory se basa en el modelo de actores, ya que todos los componentes son actores en un framework de Akka, que proporciona las bases para implementar el comportamiento de los componentes, y gestiona el paso de mensajes.

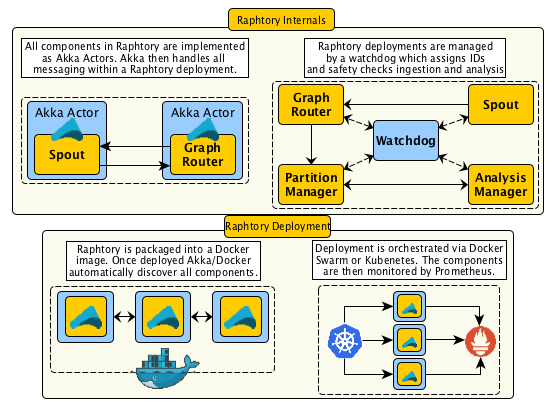


Ilustración . Entresijos de Raphtory.

Como vemos, se añade un actor vigilante (*Watchdog*), que asigna UUIDs a los GR y GPMs.

Actualmente, a la hora de desplegar Raphtory, hay dos posbilidades:

* Despliegue en local como un proyecto de código con Java/Scala.
* Despliegue en forma de contenedores Docker gestionado mediante Docker Swarm.

Así, el objetivo del TFM es que haya una tercera opción, que sea un despliegue como servicio en la nube. Es decir, una especie de API a la que le puedas indicar las características de lo que quieres de Raphtory, y automáticamente despliegue en Google Cloud una serie de instancias que implementen todo el sistema de Raphtory.

# Análisis del tutorial

Con Raphtory correctamente instalado (ver el .txt con los pasos que hay que ir siguiendo), el objetivo de este tutorial es estudiar las diferentes clases de Scala que implementan los componentes del clúster. En concreto, nos vamos a centrar en el ejemplo de El Señor de los Anillos.

## Spout

La primera clase a analizar es LOTRSpout.scala, que, como indica el nombre, es el Spout, la “fuente” que se encarga de adaptar los datos de entrada.

Esta es una clase muy breve que comienza declarando unas variables con las rutas y nombres donde se encuentran los datos que hay que inyectar en el grafo. A continuación, tenemos tres métodos, que implementan todas las funcionalidades necesarias del Spout:

* **setUpDataSource**: simplemente se encarga de extraer las líneas del fichero de datos (cada línea es una tupla de tres valores, del tipo (Personaje 1 (String), Personaje 2 (String), línea en la que aparecen juntos (int)), e insertarla en una cola.
* **generateData**: lo único que hace es desencolar las líneas comentadas anteriormente, y enviarlas al Graph Builder cuando la cola se ha vaciado.
* **closeDataSource**: sirve para cerrar ficheros en caso de que fuera necesario, no como en este caso, que no está implementada.

En realidad, el procesado de este elemento es bastante ligero, simplemente coge una fuente de externa de datos, y la convierte en un *stream* de registros para el siguiente componente, el Graph Builder.

## Graph Builder

Esta clase de Scala (*LOTRGraphBuilder.scala*) se ocupa de convertir cada una de las líneas anteriores en actualizaciones para el grafo, es decir, implementa las funcionalidades de un GR. Esto lo lleva a cabo mediante una única función llamada **parseTuple**.

Dicha función tiene dos partes. En primer lugar, la línea correspondiente se convierte en una tupla de tres elementos, eliminando cualquier espacio que tuviera el CSV. A continuación, se define lo que va a ser el nodo origen (primer personaje), se le asigna un ID, de tipo *long*, se repite lo mismo para el nodo destino (segundo personaje), y se crea un timestamp para indicar cuando ocurre el evento (en este caso, el tercer elemento de la tupla, esto es, la línea del libro en el que interactúan los dos personajes).

En la segunda parte, con estos elementos definidos, se generan las actualizaciones para el grafo, en forma de adición de vértices (función *addVertex*), y de adición de aristas (*addEdge*).

No vamos a entrar en qué es lo que hacen exactamente estas dos funciones, ya que, a nivel conceptual, ya lo hemos explicado anteriormente, y a nivel de código, implicaría entrar en los entresijos de Raphtory, que no es el objetivo de este análisis. Así, *addVertex* toma cuatro argumentos: el timestamp que hemos comentado, el ID (bien el de origen para el vértice origen, bien el de destino si fuera el vértice destino), una propiedad que se crea al vuelo, con el nombre del personaje (que es inmutable, obviamente), y el tipo, en este caso “Personaje”. La otra función, *addEdge*, es similar también, con cuatro argumentos, salvo que aquí no hay propiedades, sino ID de destino y de origen, y el tipo es “Coocurrencia de personaje”.

Como conclusión, vemos que el GB convierte cada ítem del stream de registros en una actualización de grafo.

## Raphtory Graph

Esta clase (*Runner.scala*), a pesar de ser el “main” que ejecutamos para obtener todas las funcionalidades de Raphtory, es extremadamente sencilla. Básicamente, lo único que se hace es crear una fuente, llamando a la clase que crea el Spout (*LOTRSpout.scala*); crear un constructor, llamando a la clase que crea el Graph Builder (*LOTRGraphBuilder.scala*); y crear el grafo como tal, mediante la línea *RaphtoryGraph[String](source,builder)*.

Con esto ya tenemos nuestro grafo, y, lo único que quedaría, es analizarlo, hecho donde reside la verdadera potencia de Raphtory.

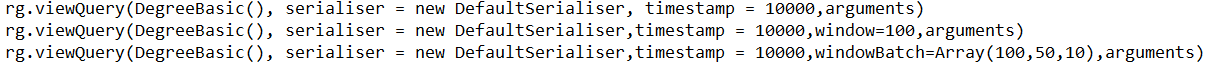
## Análisis de un grafo

Antes de entrar en el ejemplo particular de los seis grados, vamos a estudiar cómo se analiza un grafo.

Para hacer esto, básicamente lo que hay que hacer es ejecutar una consulta (*query*), con una serie de parámetros, fundamentalmente el algoritmo que quieres aplicar, y una ventana temporal donde quieras hacerlo (aunque en este ejemplo particular, los timestamps son líneas del libro, es más conveniente imaginarlo en general como instantes temporales). Así, existen dos tipos de consultas:

### View Query

Estas consultas te analizan un grafo en un instante de tiempo (*timestamp*); de un instante de tiempo (*timestamp*) hacia atrás en una ventana de duración determinada (*window*); o de un instante de tiempo (*timestamp*) hacia atrás en varias ventanas de duración determinada (*windowBatch*); esto se corresponde con cada da uno de los siguientes tres ejemplos:



El Serializador es lo que se usa para mostrar los resultados. Los argumentos pueden ser cualquier cosa de interés para el algoritmo de análisis, en este caso, un array vacío, porque no son necesarios. Recuerda que las ventanas implican que solo se van a incluir entidades actualizadas desde el instante elegido (más información sobre esto se puede leer en las secciones anteriores).

### Range Query

La idea es muy similar a la anterior, con las mismas variantes, pero aquí, en vez de coger un timestamp, se selecciona un instante inicial (*start*), un instante final (*end*), y un paso (*increment*), de manera que el algoritmo de análisis, en vez de ejecutarse una sola vez, como en el caso anterior, ocurre veces.

### Algoritmos de análisis

Esta entidad es algo más bien ajeno a Raphtory, ya que el usuario tendrá que ocuparse de definir su propio análisis para que Raphtory lo ejecute. Aun así, vamos a ver el ejemplo de la teoría de los seis grados, y algunos que ya vienen incluidos en Raphtory para no tener que reinventar la rueda.

#### Seis Grados de Gandalf

Este análisis pretende obtener la red de seis grados de separación del personaje Gandalf.Tiene tres partes:

1. **Paso preliminar (Pre-step)**: implementado mediante la función *setup*. Lo que vemos es que coge todo el grafo en el estado actual, con *view*, y, para cada vértice, determina cuál es su separación con Gandalf, con un parámetro que el usuario pasa al invocar el análisis, y se lo comunica a todos los nodos vecinos (*messageAllNeighbours*).
2. **Núcleo del algoritmo (bulk)**: esta parte está implementada en la función *analyse*. Si recordamos el modelo de Pregel, esto es lo que se ejecuta en cada uno de los súper-pasos. Básicamente, para cada vértice que ha recibido mensajes enviados en S-1 (*getMessagedVertices*). Con ello, saca la separación que le están mandando, y si es mayor que la que él tenía (su estado actual (*getState*)), actualiza el estado con la nueva separación (*setState*), y avisa a sus vecinos con mensajes (*messageAllNeighbours*). Esto se ejecuta en sucesivos súper-pasos hasta que ya no hay mensajes, lo que indicará que el algoritmo ha convergido, y todos los nodos saben cuán lejos están de la fuente.
3. **Generar los resultados**: una vez ha convergido el algoritmo, hay que obtener los resultados, y procesarlos si fuera necesario. En primer lugar, se llama a *returnResults*, que, con código particular de Scala (parecido al Map-Reduce que veíamos en MongoDB), devuelve solo los vértices con estados actualizados, y que alcanzan a Gandalf en menos de cierto número de pasos. También agrupa los resultados por el grado de separación (*map* y *groupBy*), y devuelve el tamaño de cada uno de estos grupos. Recuerda que como todo esto ejecuta sobre computación distribuida, cada trabajador devolverá los resultados de la partición concreta que él almacene. Así, es necesaria una última función, *extractResults*, que agrupe los resultados de las distintas particiones, y, si fuera necesario, ejecute más procesado (en este caso, obtiene el tamaño de toda la red, y el número de personajes que interactúan directamente con Gandalf). El resultado se devuelve como un Map[String, Any] (por eso hace falta un serializador).

## Funciones básicas de Raphtory

El objetivo de esta sección es el de analizar el código en Scala de algunas de las funciones incluidas en el framework de Raphtory.

### addVertex

El código de esta función se encuentra en src/main/scala/com.raphtory/core/model/entities/EntityStorage, al igual que otras funciones similares. Es una función sencilla que, mediante “Pattern matching” y el modelo de actores, ejecuta exactamente lo que comentábamos sobre la adición de vértices en el análisis del artículo.

### DegreeBasics

En primer lugar, hay que saber que el grado de un vértice es la cantidad de aristas entrantes en él. Así, esta clase tiene una serie de funciones que implementan todo lo necesario para el análisis, de forma similar a como hemos visto en el ejemplo (hay un setup, un returnResults, extractResults, número de pasos). Lo que hace es, para cada vértice, saca la cantidad de aristas entrantes (getIncEdges), salientes (getOutEdges), y el ID, y lo guarda todo (la información de todos los vétices) en un valor “degree”. Después, se centra solo en el tercer argumento, o sea, en la cantidad de aristas entrantes, y extrae los 20 vértices con mayor número de aristas entrantes (los 20 vértices de mayor grado). También obtiene el número total de vértices, aristas entrantes, y aristas salientes (lo hace en returnResults(), y en un solo paso (supongo que será algún tipo de análisis estático, sin intercambio de información entre vértices, con lo cual la convergencia del algoritmo es instantánea).

En el extractResults(), saca el número total de vértices, y de aristas entrantes (como componentes de los resultados anteriores), y los divide, para obtener el grado.

### ConnectedComponents

En el setup, lo que hace es recorrer los vértices y, para cada uno de ellos, fijar su estado (“cclabel”) con su ID, y enviar un mensaje a todos sus vecinos con su ID.

En el analyse, lo que hace es recorrer todos los vértices que han recibido un mensaje y, para cada uno de ellos, extrae el menor de los IDs (recuerda que lo que se está incluyendo en el mensaje es el ID), y, en el caso de que este sea menor que su estado propio (el ID que tiene en este momento), lo modifica fijando el nuevo valor inferior, y mensajea a todos sus vecinos con su nuevo ID. Si no es el caos, vota su parada,.

En el returnResults, coge todos los vértices, extrae su estado (aplicando la función getState con un map), y luego en un extractDa