INSTITUTO TECNOLÓGICO DE BUENOS AIRES – ITBA

ESCUELA DE INGENIERÍA Y GESTIÓN

TEMPORAL INDEX

Optimizaciones para el Cálculo de Caminos Continuos en Grafos Temporales

AUTOR/ES: Ribas, Ignacio (Leg. Nº 59442)

DOCENTE/S TITULAR/ES O TUTOR/ES: Soliani, Valeria Inés

TRABAJO FINAL PRESENTADO PARA LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN INFORMÁTICA

Lugar: Ciudad Autónoma de Buenos Aires, Argentina
Fecha: 16/12/2021

Optimizaciones para el Cálculo de Caminos Continuos en Grafos Temporales

Ignacio Ribas

Instituto Tecnológico de Buenos Aires, Buenos Aires, Argentina

Abstract. La adopción de bases de datos de grafos es cada vez mayor para diversas aplicaciones. Un concepto no muy extendido pero con mucho potencial, en especial en el ámbito de las redes sociales, es el de las bases de datos de grafos temporales, es decir, aquellas en las cuáles se almacena un historial de los nodos y las relaciones. En el presente trabajo se estudian algunas alternativas para la optimización de consultas por caminos continuos en bases de datos de grafos temporales. Estas optimizaciones involucran no sólo el uso de un índice estructural en el grafo cuya subestructura es el mismo camino continuo, sino también estrategias sin índice que aprovechan los algoritmos de cálculo de caminos built-in de Neo4j, el motor de base de datos en el que se desarrolla el sistema. También se presenta una extensión del lenguaje TGQL, permitiendo realizar operaciones sobre aristas que consideran sus consecuentes actualizaciones a los índices creados, así como operaciones propias para la creación de índices y la consulta a estos antes de realizar una consulta de cálculo de caminos continuos.

Keywords: Temporal Graph Databases · Neo4j · Temporal Index · Cypher Query Language.

Table of Contents

1 Introducción y Motivación

En la actualidad, la obtención de analíticos a partir de datos almacenados ha pasado de tratarse de una estrategia de negocios novedosa a una actividad cada vez más necesaria para garantizar el desarrollo y crecimiento de las organizaciones. Con grandes volúmenes de datos interconectados, las bases de datos relacionales, a pesar de ser performantes para las operaciones más básicas, pueden no ser el mejor paradigma a la hora de obtener relaciones intrínsecas. Desde hace varios años, las bases de datos de grafos, es decir, aquellas que representan datos a partir de nodos y aristas, comenzaron a adoptarse para resolver este tipo de problemas. Estas últimas, además de no necesitar una adhesión a un esquema predeterminado, son especialmente rápidas e intuitivas en el manejo de la profundidad de las relaciones.

A pesar de su rápida adopción, las bases de datos de grafos en sí no consideran las variaciones de las relaciones o de la misma existencia de los nodos en el tiempo. Esto, sin embargo, es una característica inherente en muchos escenarios de la realidad, sobre todo en las redes sociales, que están a la vanguardia de la migración al modelo de grafos. A modo ilustrativo, imaginemos el caso de un usuario que comienza a seguir a un influencer a partir de una publicación de este último. El usuario, pasado cierto tiempo, comienza a perder interés en las publicaciones del influencer, y decide dejar de seguirlo. A la hora de, por ejemplo, generar recomendaciones, es trivial la necesidad de considerar la validez de la relación en el período observado. Haciendo esto puede ganarse no sólo insight relacionado a las cuentas que dicho usuario podría seguir, sino también a qué tipos de contenidos son o no de su interés.

Las bases de datos de grafos temporales (también llamadas "grafos temporales" en el presente trabajo), son un concepto emergente en el ámbito académico, con aplicaciones dentro del dominio de OLAP. En línea con esto, los insights cuya obtención se busca simplificar a través de este tipo de grafos se reducen a caminos determinados en cierta forma por la validez de sus nodos y aristas en el tiempo. Un tipo de caminos de particular interés es aquel en el cual todos los nodos y aristas son válidos al mismo tiempo, en un cierto período. Estos se denominan caminos continuos, y son el fundamento del presente proyecto. Se explicarán en detalle más adelante.

En [5], se presenta un modelo de grafos temporales sobre los cuales pueden realizarse consultas de caminos continuos. En cuanto a la duración de las mismas, sin embargo, es inevitable el aumento exponencial de sus tiempos de ejecución en los casos en los que al aumentar la longitud de los caminos, aumenta también la cantidad de estos. Ante esta problemática, se propuso investigar la posibilidad de disminuir ese aumento de complejidad, trasladando esta a una estructura ajena al grafo (dentro de la misma base de datos).

Así es que se propone la posibilidad de crear un índice que permita guardar los caminos continuos para evitar la complejidad de calcularlos como en [5]. Permitiendo configurar la longitud máxima y el período de validez de los caminos

a indexar, sería posible guardar parte de los caminos para el posible caso en el que la cantidad de estos sea excesiva.

A partir de la presente investigación se responden interrogantes como:

- ¿Hasta qué punto es viable guardar caminos de cierta longitud en un grafo?
- ¿Es beneficioso combinar caminos de cierta longitud para calcular caminos de longitud mayor?
- ¿Pueden utilizarse las funciones built-in de Neo4j para el cálculo de caminos en pos de mejorar la performance?
- ¿Hay alguna forma de optimizar las consultas sin utilizar una estructura externa a un grafo?

2 Estado del Arte

2.1 Grafos

El grafo es esencial en las ciencias de la computación. Además de ser una entidad matemática subyacente en la representación de una gran cantidad de problemas de este área, también es implementable como una estructura de datos útil en la resolución de estos.

En particular, una aplicación de grafos con popularidad creciente son las bases de datos de grafos. Diversos son los casos de uso, incluyendo CRMs, sistemas de recomendaciones para distintos servicios de streaming y detección de fraudes [8].

2.1.1 Grafos de Propiedades (Property Graphs)

Una de las principales características de las bases de datos de grafos utilizados hoy en día es la flexibilidad en el modelado de la información. Para garantizar esta misma, en la mayoría de las implementaciones, se recurre al modelo lógico denominado Grafo de Propiedades, el cual consiste básicamente en nodos y aristas que pueden tener tipo y, como el nombre lo indica, propiedades o atributos con valores cambiantes.

A modo de ejemplo, supongamos que queremos modelar a un grupo de personas que viven en ciudades y pueden formar amistades entre sí. Para este fin, se podría definir un grafo con dos tipos de nodos y dos tipos de relaciones. Por un lado, se tendrían los nodos de tipo "Person" (o simplemente nodos Person), con los atributos "firstName", "lastName", "age", etc., y los de tipo "City", con atributos "name", "coordinates", etc. Por otro lado, las relaciones de tipo "LivesIn" (o relaciones LivesIn), que van de nodos Person a nodos City, y las de tipo "Friend", que van de nodos Persona otros nodos Person. Cabe aclarar que estas relaciones también podrían tener atributos.

Este concepto es simple pero fundamental al entendimiento del presente trabajo.

2.1.2 Neo4j

De entre los motores de bases de datos de grafos existentes, uno de los más usados al día de hoy es Neo4j [1]. Neo4j permite modelar grafos de propiedades con tipos y atributos tanto para nodos como para relaciones, sin un esquema estricto, y cuenta con un lenguaje de consultas propio, Cypher, que guarda sólo cierta relación con GQL, el estándar propuesto para lenguajes de consulta de bases de datos de grafos. Además, cuenta con una interfaz de usuario intuitiva que permite realizar consultas y visualizar sus resultados (una característica deseable en la mayoría de las aplicaciones de grafos), y también permite extender la funcionalidad de la plataforma por medio de funciones definidas por el usuario (implementadas en forma de plugins en Java). Sumado a esto, Neo4j cuenta con un equipo de desarrollo que la mantiene actualizada y añade y optimiza funcionalidades probadas con rigurosidad académica, como se puede ver en [9].

Es importante destacar que en Neo4j, para indicar el tipo de un nodo o arista, se utiliza el símbolo ':' antes del tipo. Por ejemplo, en el caso de una red social con nodos Person y aristas Friend, para encontrar los amigos de una persona de nombre "John Doe", se puede utilizar la consulta en Cypher de la Figura 1.

```
MATCH (n:Person) - [:Friend] -> (n2:Person)
WHERE n.firstName = "John" and n.lastName = "Doe"
RETURN n2
```

Fig. 1. Ejemplo de consulta en Cypher

En Neo4j, los tipos se llaman rótulos, o *labels*, y es posible que un nodo o una arista tenga más de uno. Para identificar tipos compuestos, basta con concatenar los nombres de los rótulos. Por ejemplo, la siguiente consulta (Figura 2) permite encontrar personas que también son empleados.

```
MATCH (n:Person:Employee)
RETURN n
```

Fig. 2. Rótulos compuestos en Cypher

A lo largo del presente informe, se utilizará intercambiablemnte la nomenclatura de Cypher con la utilizada en la Sección 2.1.1.

Finalmente, entre otras alternativas para motores de bases de datos de grafos existen JanusGraph, junto con el lenguaje de consulta Gremlin, y orientDB, con el mismo lenguaje de consulta, los cuales son utilizados, por ejemplo en [3].

2.2 Grafos Temporales

Puesto que los grafos permiten modelar problemas del mundo real, un caso de uso que resulta interesante (y útil) modelar es la variación de las entidades a lo largo del tiempo. Existe una moderada cantidad de bibliografía en la que este es el tema subyacente.

Por un lado, se cuenta con el desarrollo de una especificación elemental de grafos temporales sobre Neo4j, además de un lenguaje de consulta para grafos temporales. En [4], se establece una forma de representación de objetos y relaciones con un período de validez a partir de una base de datos de grafos con atributos, donde se aprovechan estos para guardar especificaciones temporales. Para llevar las funcionalidades de tal tipo de grafos a aplicaciones de alto nivel, también se crea un lenguaje de consulta asociado, TEG-QL. Esto permitiría estandarizar una forma de obtener analíticos sobre entidades relacionadas en el tiempo, como, por ejemplo, personas entre sí y con lugares.

Sumado a esto, existe trabajo relacionado con recorridos sobre grafos temporales. En [10], se propone una manera de hacer recorridos breadth-first search (BFS) y depth-first search (DFS) en grafos temporales. También se presentan casos de uso reales en los que un uso de grafos temporales junto con estos algoritmos de búsqueda son una alternativa superadora a los grafos no temporales, y al análisis de grafos por snapshots. [3] propone un sistema para la evaluación de recorridos en grafos temporales que tiene justamente como objetivo contener la explosión de escala de un grafo a lo largo del tiempo, y [2] propone un modelo centrado en el tiempo en vez de en los vértices que permite mejorar al de [3] en cuanto a búsqueda de caminos con múltiples fuentes.

2.3 Índices en Grafos

En el campo de índices en grafos, en la bibliografía existente se proponen y desarrollan ciertos tipos de índices y los métodos para generarlos. En [17], se proponen tres tipos de índices diferentes: Índices basados en caminos: efectivamente, la subestructura del grafo general a ser guardada en el índice es algún tipo de camino (el camino más corto entre dos nodos, por ejemplo) Índices basados en subgrafos: se utiliza data mining para obtener subgrafos frecuentes en el grafo a indexar. Éstos deberían acelerar la reconstrucción.

Índices construidos a partir de métodos espectrales: sin ir más en detalle, se basan en teoría espectral de grafos, donde los atributos del grafo son representados por vectores Además, en [16] se hace la distinción entre técnicas de indexado de grafos basadas y no basadas en data-mining. Básicamente, las primeras indexan todo el grafo y tienen costos elevados en memoria y para su creación, mientras que las últimas, como su nombre lo indica, usan data-mining para extraer características (features) del grafo para generar un índice invertido. Con esto, cuando se tiene una consulta, primero se encuentran las características correspondientes a q y luego se usa el índice invertido para obtener el grafo resultado a partir de estas características. Para cada uno de estos tipos se ilustra

con ejemplos de índices, los cuales pueden estar más o menos limitados al caso de uso particular al que sirven.

Finalmente, se concluye que la creación de índices con cualquiera de los métodos es costosa, que deben abarcarse sólo pocas características de los grafos a indexar para evitar explosiones en el tamaño.

Siguiendo con la idea de índices basados en caminos, [11] propone SPath, que consiste en tomar los caminos más cortos en la vecindad de un nodo como unidades de indexado, lo que mejora la capacidad de poda y la escalabilidad de la construcción de índices. Para la ejecución de consultas, se las descompone en una serie de caminos más cortos, los cuales se filtran según una medida de selectividad y se unen para formar la respuesta a la consulta. Éste trabajo también presenta pruebas con conjuntos de datos reales y generados.

En otros artículos se prueban estructuras menos estándar para el almacenamiento de índices, como lo son los índices por descomposición en árbol ([18]). Para generar estos índices, se descompone el grafo en árboles cuyos nodos -llamados bolsas- incluyen a subgrafos del (grafo) original. Estas bolsas no necesariamente son disjuntas. Se precisa que los factores determinantes del tiempo de respuesta a cada consulta son la altura del árbol generado y la cantidad de nodos en cada bolsa. En particular, esta configuración tendría buena performance en problemas consistentes en encontrar caminos más cortos o k vecinos más cercanos (KNN).

Para aplicaciones en las cuales la información completa no está al alcance, se pueden devolver grafos aproximados como respuestas a consultas. En este caso, [20] propone crear lo que llama índices conscientes de la estructura (SA-Index), que en la práctica son índices basados en el particionamiento del grafo según una medida de homogeneidad, basada en la similitud de los atributos de los nodos. En base a esto, por cada consulta, se computan los "mejores" caminos del grafo por medio del índice SA, a partir de los cuales se computa el grafo que mejor se corresponda con la consulta realizada.

Finalmente, [14] propone un intento de indexar grafos en base a patrones formados en estos (por ejemplo, triángulos). Se propone una estructura de árbol para almacenar metadatos relacionados a cada índice.

2.4 Índices Temporales

Respecto a índices para grafos temporales, existe una limitada cantidad de bibliografía de interés. Para empezar, [13] propone utilizar una estructura similar a un árbol B para la indexación y consecuente optimización de tareas de acceso a datos de una base de datos versionada. Lo nuevo del enfoque es que se aumenta la estructura de un árbol B con distintos criterios de división de nodos, ya no sólo por el valor de las claves insertadas, sino también por la combinación de esta con el tiempo en el cual una cierta entrada es válida.

Se hace énfasis en la división de una base de datos versionada en una parte "actual", en la cual todos los elementos son válidos (los últimos), y en una parte histórica, que contiene las versiones anteriores. La primera debe mantenerse de

un tamaño lo mínimo posible y soportar muchas operaciones de lectura y escritura. La segunda debe soportar muchas lecturas y pocas escrituras (de hecho, las escrituras consisten en agregar los datos que dejan de ser actuales "al final", y se realizan hasta una sola vez por cada entrada de la base de datos).

A partir de [13], sin embargo, no se llega a un enfoque universal de índices para bases de datos temporales. Por otro lado, [7] propone mejoras sobre otra estructura derivada de un árbol B, esta vez no para datos multiversión, sino para bases de datos temporales propiamente dichas, en las cuales a cada entrada le corresponde un período de validez [tinicio, tfin].

Para la construcción de la estructura sobre la que se trabaja, presentada en [6], se toma un conjunto de instantes temporales válidos obtenidos a partir de los intervalos presentes en entradas de la base de datos temporal, y se los usa de manera similar a claves en un árbol B convencional. La única salvedad es que los nodos hoja, además de contar con valores de instantes temporales tn, también cuentan con un puntero a entradas válidas entre tn y el valor de tiempo (discreto) anterior al próximo valor presente en el índice. Estos conjuntos de entradas se llaman buckets. Como una variación para mejorar la performance de las consultas, en vez de limitarse a conjuntos estáticos de entradas válidas, se toma un enfoque incremental o decremental para las entradas en buckets de un mismo nodo hoja, es decir, se indica qué entradas pasaron a o dejaron de ser válidas entre un instante y otro, en lugar de sólo conservar las válidas en cada instante.

A partir de esta especificación, se plantean tres variaciones que permiten optimizar el tiempo en las consultas. A grandes rasgos, la primera consiste en separar lo incremental de lo decremental en distintos buckets, requiriendo más espacio pero presentando mejores resultados en general. En los otros dos enfoques, se trata de evitar la duplicación de datos, ocupando menos espacio y con mejor rendimiento en general.

2.5 Índices en Grafos Temporales

El tema de índices para grafos temporales es un tema poco estudiado en el ámbito académico. Dicho esto, existe trabajo relacionado a índices y/o caminos en grafos de Resource Description Frameworks (RDFs), agregando características temporales. En esta línea, [12] trata el tema del guardado de caminos en un índice para mejorar la performance de queries en un grafo RDF.

La noción de temporalidad para RDFs es agregada en [19], donde se propone una indexación para esta aplicación en particular, con un índice para caminos de prefijos y sufijos de elementos a partir de un grafo RDF inducido. Lo importante de estos índices es la concentración en la subestructura del camino y la consideración de la noción de temporalidad. También cabe destacar que, además de creación, se manejan casos de inserción y remoción de elementos en la base de datos. Los beneficios que aporta esta alternativa pueden verse, sin embargo, sólo para los caminos más frecuentes, en los que se centra el índice propuesto.

[15] presenta una idea de indexado para grafos temporales propiamente dichos, para la aplicación en documentos XML. Agregando atributos temporales, se llega a una alternativa al versionado que mejora la performance para bases de datos con actualizaciones frecuentes que afectan muchas versiones de una misma entidad. Esta alternativa se materializa en un índice cuya unidad principal es el mismo camino continuo.

Finalmente, nuestro trabajo se basa en [5], donde se propone un lenguaje de consulta para grafos temporales basado en GQL (TGQL), se tipifican distintos caminos en este tipo de grafos, tales como caminos continuos, continuos a pares, y consecutivos, y se propone una idea para índices basados en estos tres tipos de caminos. En el mismo se presentan algoritmos de actualización del índice para casos de inserción o modificación de aristas en el grafo.

2.6 Bases para el Presente Trabajo

Como se mencionó en la sección anterior, el presente trabajo es una ampliación de un proyecto anterior [5], en el cual se implementa un modelo de base de datos de grafos temporales en Neo4j, junto con un lenguaje de consulta para grafos temporales -TGQL- y un analizador sintáctico que traduce las consultas de dicho lenguaje a Cypher. Además, este proyecto cuenta con una interfaz gráfica que permite ejecutar consultas de TGQL, traduciéndolas y ejecutándolas contra una base de Neo4j en tiempo real.

2.6.1 Modelo de Datos

Para poder introducir la dimensión temporal en una base de datos de grafos, fue necesario en el proyecto base [5] definir un modelo de datos por medio del cual se pudiera determinar la validez de los nodos, las relaciones y los atributos de estos. Esto pudo solucionarse en Neo4j explotando los beneficios de los grafos de propiedades, a partir de la definición de un esquema introduciendo distintos tipos de nodos y aristas.

Para los nodos, se tienen tres tipos, Object, Attribute y Value, que se explican a continuación:

- **Nodos Object:** estos nodos representan cualquier tipo de entidades, como, por ejemplo, personas o ciudades.
  Tienen los siguientes atributos:
  - **title:** hace referencia a la entidad que representa. Siguiendo con el ejemplo anterior, para representar a una persona el valor de este atributo podría ser 'Persona'.
  - **interval:** es el período de tiempo en el cual se considera que esta entidad existió en la circunstancia que se está modelando. En el mismo ejemplo, podría hacer referencia al período de vida de una persona, o al período en el cual dicha persona formó parte de una red social.

- **Nodos Attribute:** con estos nodos se representan los atributos que tiene o tuvo una entidad a lo largo del tiempo. Son, junto con los nodos Value, lo que permite añadir validez en el tiempo a los mismos atributos de un grafo de propiedades. Los nodos Object están relacionados a los nodos Attribute a partir de aristas de tipo Edge, explicadas más adelante. Es pertinente aclarar que dos nodos Object con un mismo atributo tienen cada uno su nodo Attribute.
  Tienen los siguientes atributos:
  – **title:** en este caso se refiere al nombre del mismo atributo. En el ejemplo anterior, las personas podrían tener cada una nodos Attribute donde la propiedad title tome el valor 'Nombre'.
  – **interval:** esta propiedad sirve para determinar en qué períodos una entidad tuvo un atributo. La utilidad de esto puede verse en redes sociales, donde, por ejemplo, un usuario puede tener una foto de perfil por cierto tiempo hasta que decide eliminarla.

- **Nodos Value:** definen los valores que tiene o tuvo un atributo de una entidad a lo largo del tiempo. Los nodos Attribute están relacionados a este tipo de nodos a partir de aristas Edge.
  Sus atributos son los siguientes:
  – **value:** es el valor propio del atributo. Por ejemplo, volviendo al ejemplo de las redes sociales, una persona (nodo Object con title = 'Persona') tiene como nombre de usuario (nodo Attribute con title = 'Username') el valor 'student' (nodo Object con value = 'student')
  – **interval:** intuitivamente, hace referencia a los intervalos en los cuales el atributo tomó el valor especificado en value.

Las aristas pueden clasificarse en dos grupos, las regulares y las de tipo Edge.

- **Aristas regulares:** el tipo de estas aristas está definido por la relación que representan. Por ejemplo, en una red social, las amistades podrían estar representadas por una arista de tipo amigoDe. Estas aristas tienen como única propiedad el campo intervalo, que, naturalmente, indica el período de validez de la relación que representan.
- **Aristas Edge:** su única función es relacionar nodos Object con nodos Attribute y nodos Attribute con nodos Value. No tienen propiedades en este esquema.

Puede observarse un ejemplo simple en la Figura 3. En esta, entre otras cosas, puede verse modelada una persona nacida en 1980 cuyo nombre fue Anna hasta 1990, y luego cambió a Alice. Alice es, desde 2020, amiga de otra persona, que a su vez fue amiga de Bob desde 1979 hasta 1991 (inclusive). En la actualidad, estas tres personas viven en Buenos Aires.

Cabe aclarar que el campo interval en cualquier tipo de nodo o arista mencionado anteriormente consiste en un conjunto de intervalos temporales disjuntos. Asimismo, cada uno de estos intervalos es cerrado.

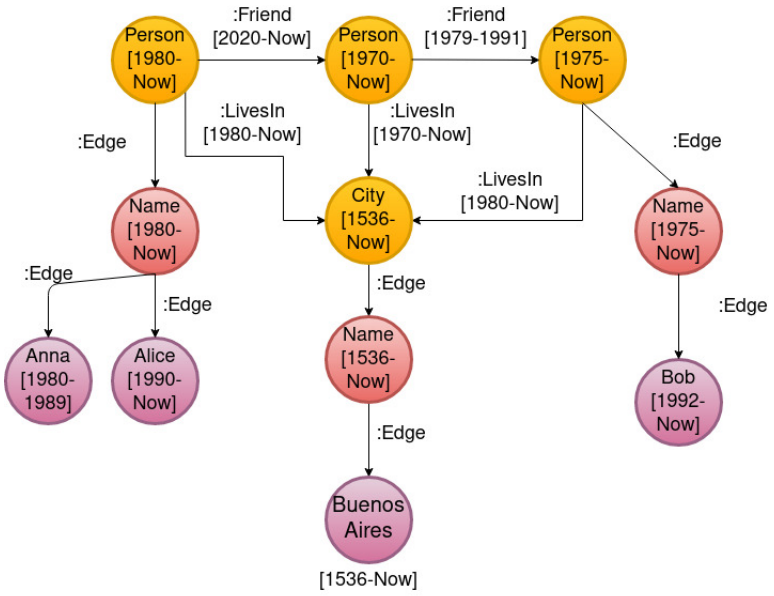

Fig. 3. Un grafo temporal de una red social. En naranja los nodos Object, en rojo los nodos Attribute y en violeta los nodos Value. Se omiten los nombres de los campos title, interval y value.

Además, cada nodo cuenta con un atributo id que permite referenciarlos independientemente de la base de datos en la cual se encuentre el dataset. Esto es así porque, utilizando los ids nativos de Neo4j (cada nodo es asignado un valor de esta propiedad automáticamente), estos pueden varían cuando se hacen backups y restores de bases de datos, lo cual es molesto para llevar a cabo pruebas en bases de datos que cuentan con el modelo previamente explicado.

2.6.2 Caminos Continuos

El trabajo anterior [5] contiene funciones y trata un espectro más amplio de características de grafos temporales que sólo caminos continuos, como, por ejemplo, caminos continuos a pares, o distintos tipos de caminos consecutivos según el caso (camino más corto, camino de partida más temprana, camino más rápido, entre otros). En este trabajo nos enfocaremos en caminos continuos únicamente.

Dado el modelo de datos introducido en el apartado anterior, los caminos continuos pueden redefinirse como caminos en un grafo temporal tales que la intersección de las propiedades interval de todos sus nodos Object y aristas regulares sea no vacía. Esta misma intersección es además el intervalo de validez del camino continuo.

Junto con esta definición, se retoma el pseudocódigo del algoritmo que permite calcular caminos continuos en un grafo temporal, presentado en [5], en Algoritmo 1.

En el resto del informe se asume que los grafos son consistentes, de forma que el intervalo de validez de un nodo Object siempre incluye al intervalo de validez de cualquiera de sus aristas salientes y entrantes, y de los nodos Attribute y Value con los cuales está relacionado por medio de aristas Edge. Siguiendo esto, para que un camino sea continuo, sólo basta con que la intersección de los intervalos de sus aristas sea no vacía.

Por ejemplo, en la Figura 4, el camino de A a B es continuo, ya que la intersección de los intervalos de sus aristas es [2010-2013, 2015-2015]. Sin embargo, para el camino de C a D, esta intersección es vacía, por lo que no es continuo.

Teniendo esto en cuenta, se presenta un panorama general del algoritmo coTemporalPaths:

1. Se parte de un nodo inicial, llamémoslo s. Se tiene una cola de nodos frontera, F. Esta implementación recuerda a la de una búsqueda BFS. De hecho, salvo por las condiciones de corte y el hecho de que los nodos frontera pueden repetirse, coTemporalPaths es similar a dicho algoritmo. Cada nodo tiene una referencia a su nodo previo. En este caso, s es el único que no tiene tal referencia. Sea u un nodo en F, siguiendo las referencias de nodos previos, se tiene un camino entre s y u, llamémoslo u.P.
   Además, se empieza con una lista de solucioens, S, vacía.
2. En cada iteración, se desencola un nodo u de F. Para cada vecino v de u, si la intersección entre el intervalo de validez de v, el intervalo de validez de u.P y el intervalo de búsqueda es no nula, se encola v en F. Se fija u como el nodo previo de v. Además, si v.P tiene una longitud en el rango buscado, se lo agrega a la lista de soluciones.
   Como no es necesario que los vecinos de u sean sólo nodos no visitados, es posible que los nodos frontera se repitan, pero tendrán una referencia a un nodo previo diferente en cada caso.
3. Para cada nodo n en S, se calcula n.P y se devuelve ese camino continuo.

También se introduce la notación de caminos como tuplas. Por ejemplo, un camino que pasa por los nodos A, B y C en ese orden, se puede expresar como (A, B, C). Asimismo, se introduce la notación de camino con su intervalo de validez como un par <camino, intervalo>, por ejemplo <(A, B, C), ['2015-2021']>.

2.6.3 Implementación del Sistema

Entendiendo el modelo de datos utilizado, así como la lógica detrás del cálculo de caminos continuos, se procede a explicar los componentes del sistema a grandes rasgos.

2.6.3.1 Arquitectura

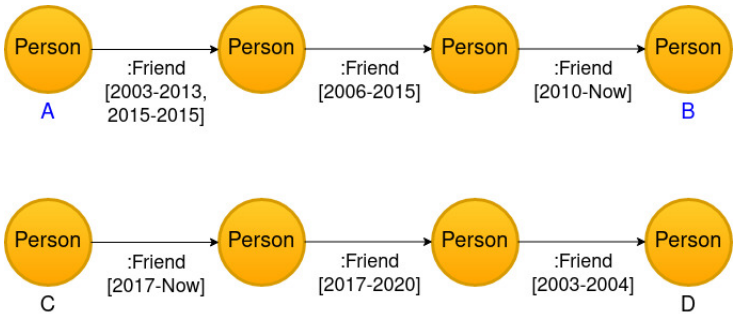

Fig. 4. Dos caminos. El camino de A a B es continuo, con intervalo de validez [2010-2013, 2015-2015]. El de C a D no es continuo ya que la intersección de los intervalos de sus aristas es nula.

.

Algorithm 1 Computes Coexisting Paths (Continuous and pairwise continuous paths)

Input: A graph G, a source node x, the minimum path length L_{min}, the maximum path length L_{max}, a function f depending on the type of path requested, and a destination node y (optional).

Output: A list of coexisting paths S.

Initialize a queue of paths Q and a list of solutions S.
$Q.enqueue([(x, [-inf, +inf], 0)])$
while not Q.isEmpty **do**
  $current = Q.dequeue()$
  $z, interval, length = current.last()$
  for $(z, otherInterval, dest) \in \mathcal{G}.edgesFrom(z)$ **do**
    if not current.containsNode(dest) and $interval \cap otherInterval \neq \emptyset$ **then**
      $newTuple = (dest, f(interval, otherInterval), length + 1)$
      $copy = current.copy()$
      $copy.insert(new_tuple)$
      if $L_{min} <= length + 1 <= L_{max}$ and (y not exists or $dest == y$) **then**
        $S.insert(copy)$
      end if
      if $length + 1 < L_{max}$ **then**
        $Q.enqueue(copy)$
      end if
    end if
  end for
end while

La infraestructura consta de dos módulos funcionales. Por un lado, se cuenta con una base de datos en Neo4j (versión 3.5.17[1]), junto con un plugin donde se definen las extensiones a Neo4j que contienen los algoritmos necesarios para la obtención de los distintos tipos de caminos (incluyendo la *procedure* coTemporalPaths).

Por otro lado, se tiene una aplicación con una interfaz gráfica simple desarrollada en Javalin, la cual permite hacer consultas en TGQL y mostrar sus resultados. También cuenta con la definición de la sintaxis del lenguaje implementado, junto con su analizador sintáctico, generado en antlr4. Por detrás, esta interfaz traduce las consultas de TGQL a Cypher y las ejecuta conectándose al módulo de base de datos.

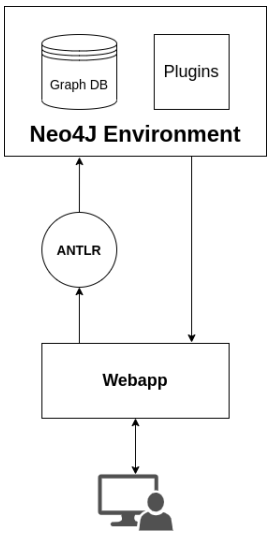

Fig. 5. Arquitectura del sistema y flujo de información. [5]

En la práctica, el flujo de evaluación de una consulta, ilustrado en la Figura 5 es el siguiente:

1. El usuario introduce, por medio de la interfaz gráfica de una webapp, una consulta en el lenguaje TGQL.
2. La consulta es traducida a su vez al lenguaje de consulta de Neo4j, llamado Cypher. La traducción es llevada a cabo por un parser generado en antlr4.
   – Cypher, que está basado en Java, permite extender su funcionalidad a partir del uso de funciones definidas por el usuario, o *procedures*. Estas mismas pueden ser llamadas directamente dentro de una consulta Cypher.
3. En el DBMS de Neo4j, el código Cypher se traduce a código de Java y se ejecuta.

[1] También se utilizó la versión 3.5.14 durante el desarrollo.

4. La respuesta de la consulta Cypher se devuelve a la webapp, donde se le da un formato y se muestra al usuario.

En cuanto al código fuente, se trata de un proyecto en Maven que consta de dos módulos: por un lado, la aplicación web desarrollada en Javalin - llamado Cliente TDBG, y, por otro lado, el plugin que permite integrar las *procedures* relacionadas al cálculo de caminos con Neo4j.

2.6.3.2 Cliente TDBG

En este módulo se implementa una interfaz que permite utilizar TGQL sobre una base de datos de Neo4j. En estos términos, puede considerarse de más "alto nivel" que el módulo de Procedures. El cliente TDBG funciona convirtiendo consultas escritas en TGQL a consultas escritas en Cypher, es decir, por cada consulta escrita en TGQL, existe una única consulta en Cypher, sin necesidad de ejecutar más de una consulta. Esto es posible gracias a la implementación de los ya mencionados procedimientos, que permiten encapsular lógica de consulta y filtrado en una línea de Cypher, facilitando la modularización del proyecto y la integración de extensiones.

Esto último es importante, ya que existe una cláusula de TGQL que en sí implica la realización de consultas anidadas, WHEN. Esta cláusula permite ejecutar consultas analíticas de gran significado, ya que permite realizar un mapeo entre estados de relaciones e intervalos de tiempo. Intuitivamente, permite filtrar los resultados obtenidos según si pertenecen a el o los intervalos de tiempo en los cuales se da la relación mencionada después de WHEN. Por ejemplo, "devolver los amigos de Mary cuando ella vivía en Londres".

También es de importancia que puedan realizarse dos consultas en una ya que se necesita hacer algo similar con el índice: si la respuesta a la consulta se puede obtener del mismo índice, entonces ejecutar una consulta que obtenga los datos del índice, y, si no, ejecutar una consulta que obtenga los datos sin usar el índice. Esta condicionalidad de ejecutar cierta consulta si otra consulta devolvió cierto valor (que requiere dos consultas en los casos mejor y peor), también se puede conseguir en Neo4j.

Para la traducción de consultas, se utiliza un analizador sintáctico, y, para definir el analizador sintáctico, primero es necesario definir la sintaxis del lenguaje. Es así que TGQL se encuentra definido en un archivo de antlr4. Esta biblioteca de Java permite la creación de parsers a partir de una especificación de un lenguaje. Básicamente, antlr4 genera una regla de análisis de entrada y otra se salida por cada token (se asume que el lector está familiarizado con teoría de lenguajes y compiladores), las cuales por defecto están vacías (son métodos de Java que pueden ser sobrescritos si se quiere aplicar cierta lógica a los tokens que estén por parsearse o que hayan sido parseados). A partir de estas reglas se obtiene una estructura que, si bien no es formalmente un árbol de sintaxis abstracto (AST), cumple con una de sus funciones, que es poder obtener la información suficiente sobre la entrada en TGQL para poder obtener el parseo correcto en Cypher.

Finalmente, se obtiene el output, en texto, que constituye la consulta deseada para utilizar en la base de datos de Neo4j. Una vez obtenida la consulta en Cypher, se procede, desde el mismo cliente, a ejecutarla (se establece una conexión con Neo4j al iniciar el cliente), obteniendo los resultados y mostrándolos en pantalla, como una tabla donde las columnas son las variables que se explicitan en la cláusula RETURN (también se asume que el lector cuenta con un cierto conocimiento sobre Neo4j y Cypher).

Sabiendo cómo funciona el analizador sintáctico, es necesario mencionar la función de TGQL que mayor importancia tiene para el presente proyecto, cPath. Esta función recibe, en notación de TGQL, una relación entre dos nodos, y, opcionalmente, un intervalo de tiempo según el cual se quiere filtrar a los caminos obtenidos (inducidos por la relación dada) por su validez. Para ilustrar lo mencionado, un ejemplo de uso de cPath en TGQL se puede ver en la Figura 6.

```
SELECT p.path[0].attributes.Name as from,
p.path[3].attributes.Name as to,
p.interval as interval
MATCH (n:Person),(n2:Person),
p = cPath((n)-[:Friend*3]->(n2), '2010-03-07', '2014-03-08')
```

Fig. 6. Ejemplo de uso de cPath en TGQL

En esta consulta se calculan los caminos continuos de longitud 3 en el intervalo entre el 3/7/2010 y 8/3/2014. Para cada camino, se obtienen los nombres de la primera y de la última persona del camino.

2.6.3.3 Procedures

Como ya se mencionó, en Neo4j, una *procedure* es una función definida por el usuario que puede ser integrada al sistema para ser ejecutada en la máquina virtual nativa (de Neo4j). Esto permite la implementación de cualquier tipo de lógica de función y su ejecución dentro del motor de base de datos -siempre y cuando sea expresable en Java-, lo que brinda una considerable flexibilidad a Neo4j, así como expresividad a Cypher, permitiendo también reducir el overhead de la ejecución de consultas desde un cliente conectado.

En el contexto del proyecto anterior [5], nos centramos en la lógica de los procedimientos que permiten la obtención de caminos continuos, es decir, no se entra en detalles de la implementación, salvo donde sea necesario.

Teniendo esto en cuenta, se procede a describir el procedimiento utilizado para la obtención de caminos continuos, y al cual se mapean los llamados a cPath en TGQL. Antes que nada, veamos en la Figura 7 cómo se traduce la consulta TGQL de la Figura 6 a Cypher [2]:

El parser en el cliente TDBG permite traducir las llamadas a cPath, obteniendo nodos de origen y destino, así como el nombre de la relación sobre la cual

[2] Antes de los cambios realizados en el presente trabajo

```
MATCH (n:Object {title: 'Person'}),(n2:Object {title: 'Person'})
CALL coexisting.coTemporalPaths(
    n, n2, 3, 3, {
        edgesLabel:'Friend',
        between:'2010-03-07-2014-03-08',
        direction:'outgoing'
    })
YIELD path as internal_p0, interval as internal_i1
WITH {path: internal_p0, interval: internal_i1} as p
RETURN p.path[0].attributes.Name as from,
       p.path[3].attributes.Name as to,
       p.interval as `interval`
```

Fig. 7. Traducción de la consulta de la Figura 6 en Cypher

se quiere buscar los caminos y su longitud o rango de longitudes. También se tienen en cuenta las fechas de inicio y fin del intervalo de validez deseado. Como resultado se obtiene una llamada a la *procedure* coexisting.coTemopralPaths (coTemporalPaths para abreviar), que recibe, en este orden, nodo de inicio, nodo de fin, longitud mínima, longitud máxima, y una estructura clave-valor (dict/hash) en la cual se especifican el nombre de la relación ("edgesLabel") y el intervalo ("between"). En resumen, cada llamada a cPath en TGQL se corresponde con una llamada a coexisting.coTemporalPaths en Cypher. Esta última, a su vez, es la implementación del pseudocódigo en el Algoritmo 1.

Independientemente del método utilizado, una vez ejecutado el algoritmo que calcula los caminos deseados, se los serializa a JSON y se los devuelve como respuesta.

3 Extensión y Alcance del Proyecto

La ampliación al proyecto anterior [5] consiste en el diseño e implementación de un índice que tiene como estructura de indexación el camino. Más específicamente, se tratan caminos continuos, según la definición dada más arriba. En el proceso de implementación, sin embargo, también se realizan mejoras sobre el trabajo base que no estaban previstas como objetivos del presente proyecto, pero que resultan favorables en pos de conseguirlos.

El proyecto busca alcanzar los siguientes objetivos:

- Diseñar un índice, añadiendo *procedures* de Neo4j para su creación.
- Permitir realizar consultas de caminos sobre el índice, llegando a mejorar la performance de coTemporalPaths
- Agregar reglas del lenguaje a TGQL para permitir:
  - Creación de índices
  - Consultar el índice en una consulta de cálculo de caminos
  - Creación, eliminación y modificación de aristas con su correspondiente actualización del índice

– Realizar pruebas sobre bases de datos sintéticas (realistas) y sobre un conjunto de datos real. Se le prestará más atención a las primeras, dado que presentan una mayor variedad de casos a analizar.

4 Solución

Con el fin de optimizar las consultas de cálculo de caminos continuos en términos de tiempos de ejecución, se desarrollaron varias soluciones, muchas de las cuales explotan la idea de un índice donde la subestructura almacenada es efectivamente el mismo camino continuo, o parte de este.

4.1 Optimización de coTemporalPaths - quickCoPath

Una de las primeras optimizaciones que pueden hacerse sobre el proyecto anterior [5] aplica directamente a la función que permite obtener los caminos continuos entre dos nodos, coTemporalPaths. Como se explica en la sección anterior, esta función utiliza una *procedure* de Neo4j, coTemopralPaths, que realiza una búsqueda BFS con criterios de filtrado de nodos frontera específicos para garantizar la continuidad de los caminos que se obtienen del subgrafo resultante. De esta manera, se pueden calcular correctamente los caminos continuos entre el nodo fuente y el destino, especificados como parámetros de la función.

Esta función, sin embargo, no explota las funciones built-in de Neo4j para el cálculo de caminos. Inicialmente el planteo fue si haciendo esto último no podría superarse la performance de coTemporalPaths de alguna manera.

La alternativa propuesta fue la siguiente: en vez de utilizar BFS con condiciones de filtrado de nodos frontera para garantizar continuidad de los caminos, lo que podría hacerse es buscar todos los caminos utilizando el poder expresivo del lenguaje de consulta nativo (Cypher), y luego podar los caminos que no sean continuos.

Esta alternativa se implementó como *procedure*, identificada como coexisting.quickCoTemopralPaths (o quickCoTemporalPaths), la cual recibe los mismos parámetros que coTemopralPaths, permitiendo la obtención de mejores resultados para una misma interfaz.

A continuación, se presenta el pseudocódigo de este algoritmo. Como para el cálculo de caminos se utilizan funciones built-in de Neo4j de la cual no se dispone la complejidad, es difícil calcular exactamente la complejidad del algoritmo.

4.2 Índices

A lo largo del proyecto, se desarrollaron dos tipos de índices, ambos coexistiendo con el grafo indexado en la misma base de datos. Como cualquier otro índice, el

Algorithm 2 quickCPath: Computes Continuous Paths from a Source Node to a Destination Node

Input: A graph G, a source node x, the minimum path length L_{min}, the maximum path length L_{max}, a relationship name $relName$, and a destination node y, a search interval $searchInterval$.

Output: A list of Continuous Paths S.

```
Initialize a list of paths S.
Initialize a list of paths P.
P := { p | p is a path of length L from x to y, for L_min ≤ L ≤ L_max}
for each p ∈ P do
  if ⋂_i interval(p_i) ∩ searchInterval ≠ ∅ then
    S.insert(p)
  end if
end for
return S
```

objetivo de su uso es permitir recuperar información más rápidamente, en este caso tratándose de caminos continuos.

Ambos índices sólo permiten indexar caminos con aristas de un mismo tipo, y, para evitar guardar caminos que no son consultados frecuentemente, se debe restringir el intervalo de indexación al momento de su creación y cuando se los consulte. El intervalo de indexación es aquel a partir del cual se determina si un camino continuo va a pertenecer al índice o no: si la intersección entre el intervalo de indexación y el intervalo de validez del camino es no vacía, entonces este camino es indexado. De lo contrario, el camino se descarta.

4.2.1 Índice "Tree"

El primer acercamiento a una estructura de índice es la que denominamos "Tree" y consiste en lo que se podría llamar un "árbol" de tres niveles, cada uno con un tipo de nodo:

1. **ROOT :** es el nivel más alto, y el cual indica por medio de un único nodo si existe algún índice creado para el grafo en cuestión. Los nodos de este nivel se llaman nodos Root y no tienen propiedades.
2. **INDEX:META:Relationship :** el segundo nivel, cada nodo indica si un índice fue creado para el grafo para un tipo de relación determinada, para un intervalo, y longitudes mínima y máxima determinados. El tipo de relación se especifica en el mismo tipo del nodo. Por ejemplo, para la relación Friend, este nivel tendría nodos del tipo :INDEX:META:Friend. Está conectado al nodo del nivel ROOT vía una arista de tipo :meta. Los nodos de este nivel se llaman nodos Meta y tienen las siguientes propiedades:
   - **From:** Fecha de inicio del período de indexación
   - **To:** Fecha de finalización del período de indexación
   - **MinLength:** Mínima longitud de los caminos de este índice
   - **MaxLength:** Máxima longitud de los caminos de este índice

3. **INDEX :** cada nodo contiene toda la información necesaria para reconstruir un camino. Según el tipo de relación de los caminos, su longitud y su período de validez, se conectan a un nodo de tipo INDEX:META:Relationship por medio de una arista de tipo :type. Los nodos de este nivel se llaman nodos Index y tienen las siguientes propiedades:
   - **From:** Fecha de inicio del período de validez del camino
   - **To:** Fecha de finalización del período de validez del camino
   - **Length:** Longitud del camino
   - **Source:** ID del nodo de origen
   - **Destination:** ID del nodo de destino
   - **Intermediate:** IDs de nodos intermedios

   Como sólo se puede especificar una fecha de inicio y una de finalización, para los caminos que tengan períodos de validez disjuntos es necesario crear un nodo Index por cada intervalo de validez. Es decir, cada nodo Index representa un único par <camino, intervalo>, donde el intervalo tiene un único elemento. Por ejemplo, si el período de validez de un camino es [2010-2015, 2017-2018, 2020-2021], se requieren tres nodos Index para indexarlo, que sólo varíen en los campos From y To.
   En la implementación, para cada una de las propiedades de los nodos Index existe un índice B-tree de Neo4j[3]. Esto permite acelerar notablemente las consultas de reconstrucción de caminos.

Para los IDs de los nodos, se utiliza el valor de una propiedad llamada “id”, y no los IDs nativos de Neo4j. Esto, como ya se mencionó, es importante ya que permite el copiado de bases de datos de un host a otros, así como la validez del índice luego de este proceso, ya que los IDs nativos pueden cambiar según la instancia que se ejecute.

La Figura 8 ilustra la estructura de un índice "Tree". El nodo Root indica que el índice existe. En este caso, como se indica en el nodo Meta, están indexados todos los caminos cuyas relaciones son de tipo "Friend", son válidos entre 2000 y 2020 (inclusive), y tienen una longitud entre 2 y 3 (inclusive). Hay un total de 4 pares <camino, intervalo> que cumplen estas condiciones, representados por los 4 nodos Index. Por ejemplo, el nodo Index de la derecha representa al camino continuo <(19, 88, 89, 18), [2004-2009]>.

4.2.1.1 Creación del Índice

Se implementó una *procedure* para la creación de índices que permite especificar el tipo de relación sobre la cual se quiere indexar caminos, es decir, el tipo de aristas de los caminos (por ejemplo, “Amistad” en una red social). También se especifican la máxima longitud de los caminos y su período de validez. A partir de estos parámetros, se crea un nodo Index por cada camino, además de un nodo Meta que permite acelerar las búsquedas. Esta *procedure*, a su vez, hace uso de coTemporalPaths para obtener los caminos indexados.

[3] Neo4j 3.5.17 permite la creación de índices simples y compuestos para las propiedades de nodos y aristas.

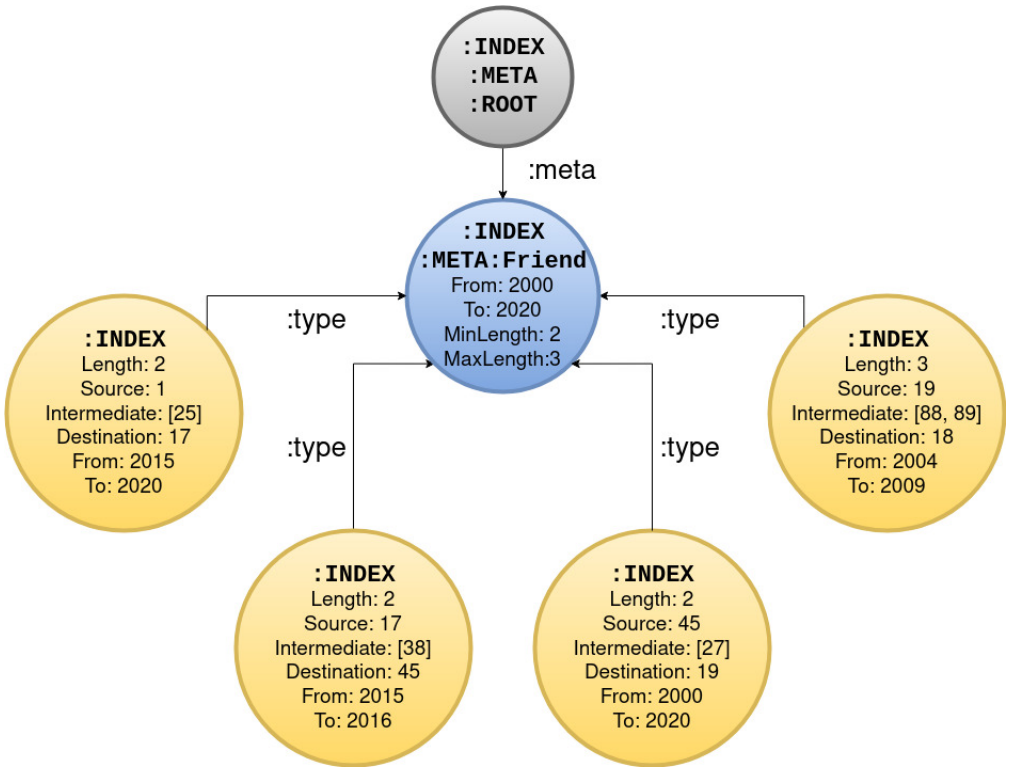

Fig. 8. Parte de un índice Tree para un modelo de red social como el de la Figura 3. El intervalo de indexación, indicado en el nodo META (azul), es de 2000 a 2020.

El Algoritmo 3 se utiliza para la creación de un índice "Tree". Este mismo, en resumen, lleva a cabo los siguientes pasos:

1. Verifica que exista un nodo Root. Si no existe se lo crea.
2. Verifica que exista un nodo Meta que para la relación, el intervalo y las longitudes mínima y máxima especificadas (es decir, que estas propiedades del nodo tengan el mismo valor que se busca). Si no existe se lo crea. También se crea una arista :meta del nodo Root al nodo Meta.
3. Por cada nodo Object del grafo, utiliza coTemoralPaths para calcular los caminos continuos cuyos tipo de relación, longitud e intervalos de validez estén dentro de los parámetros ingresados. Por cada camino obtenido se crea un nodo Index con los datos correspondientes.

El crecimiento del índice puede ser desmesurado en grafos con un grado promedio mínimo de los nodos. Por ejemplo, en un grafo modelando una red social con 2500 usuarios, de los cuales cada uno tiene como máximo 8 amigos, para un período de 5 años (la red social tiene relaciones que comprenden un período de al menos 20 años), los nodos Index requeridos para caminos de longitud 2 son alrededor de 7000, mientras que para caminos de longitud 4, superan los 14000. Los tiempos de creación también crecen demasiado para grafos medianamente grandes y longitudes de caminos mayores a 3.

4.2.1.2 Retrieve Paths

Una primera implementación de un algoritmo de obtención de caminos fue la manera más simple y directa: buscar nodos Index que tuvieran los nodos de origen y destino, longitud e intervalos de validez deseados. Eso es relativamente simple en Cypher, y por lo tanto también en la implementación de *procedures*.

Sin embargo, algo que puede suceder frecuentemente es que el intervalo de búsqueda de los caminos tenga una intersección no vacía con el intervalo de validez de un índice, pero que no esté totalmente incluido en este. En ese caso, una parte de los caminos buscados está indexada, mientras que hay otra que debe buscarse por separado (sin un índice).

En el Algoritmo 4 se presenta el pseudocódigo del algoritmo de recuperación de caminos directamente a partir de un índice "Tree".

4.2.1.3 Construyendo caminos de longitud arbitraria a partir de caminos de longitud 2: retrievePathsConcat

A pesar de la relativamente buena performance del algoritmo implementado para la reconstrucción de caminos a partir de nodos del índice, a veces es inaceptable el crecimiento de este último para caminos de longitud mayor a 2. Como una alternativa superadora en términos de espacio, se implementa un algoritmo que consiste esencialmente en la concatenación de caminos indexados de longitud 2 para reconstruir caminos de cualquier longitud. Para los caminos de longitud impar n, se obtienen los caminos de longitud par n-1 y se les concatenan los caminos de longitud 1 - es decir, las aristas - pertinentes. La intuición

Algorithm 3 Creates a Tree Index for a Temporal Graph

Input: A relationship name $relationshipName$, an interval Int_{index}, a temporal graph G, a temporal index I

Output: A modified Temporal Index I

if $\nexists_{N_R} \in I \mid N_R$ is a :ROOT node **then**
Create a new :INDEX:META:ROOT node N_R
end if
if $\nexists_{N_{meta}} \in I \mid N_{meta}$is a :META:$relationshipName$ node |
$N_{meta} = \{$
$From$:Int$_{index}$.$From$,
To: $Int_{index}.To$
$\}$ **then**
Create such N_{meta}
Create a :meta edge from N_{meta} to N_R
end if
Let $IDs_{Object} \coloneqq \{$o.id$|o isa : Objectnode\}$
for each id$_{obj} \in IDs_{Object}$ **do**
let $L_{pairs} \coloneqq quickCoTemporalPaths(G, I, x, null, 2, 2, Int_{index})$,
$where$ x.id = id$_{obj}$
for each $(path, interval) \in L_{pairs}$ **do**
Create an :INDEX node N_{index} |
N$_{index} = \{$
Source: id_{obj},
Intermediate: [$path_1.id$],
Destination: $path_2.id$,
From: $interval.From$,
To: $interval.To$,
$\}$
Create a :type edge from N_{index} to N_{meta}
end for
end for
return I, G

Algorithm 4 retrievePaths: Computes Continuous Paths from a Source Node to a Destination Node Using Tree Index

Input: A graph G, a Tree Temporal Index I, a source node x, a destination node y, the minimum path length L_{min}, the maximum path length L_{max}, and a relationship name $relName$, a search interval $searchInterval$.

Output: A map M whose keys are paths and whose values are intervals, representing continuous paths.

Initialize a list of nodes N_M.
$N \coloneqq \{$
$iNode_{meta} \in N_M \mid$ iNode$_{meta}$ is a META node $\wedge$
$iNode_{meta}.MaxLength \geq L_{max} \wedge$
$iNode_{meta}.Source = x.id \wedge$
$iNode_{meta}.Destination = y.id \wedge$
$intersects(searchInterval, [iNode_{meta}.From, iNode_{meta}.To])$
$\}$
Initialize a map M
for each $iNode_{meta} \in N$ **do**
  for each $iNode \in I \mid hasRelationship(iNode, iNode_{meta}, 'type')$ **do**
    $M[getPath(iNode)].insert(getInterval(iNode))$
  end for
end for
return M

detrás de la correctitud del algoritmo se presenta más adelante. Los resultados son similares a los de la primera implementación, aunque lo importante es que se puede llegar a caminos de longitudes que requerirían una gran cantidad de espacio para indexarse directamente, por lo que se ahorra memoria y se agilizan las búsquedas de nodos en bases de datos de tamaños considerables. Para el pseudocódigo de este algoritmo, primero se presentan dos funciones auxiliares, los Algoritmos 12 y 13 en el Apéndice D.

Con estas funciones, el Algoritmo 5 permite calcular caminos a partir de un índice "Tree" concatenando caminos de longitud 2.

El algoritmo es simple. Para cada longitud en el rango deseado, se construyen caminos de dicha longitud a partir de secuencias de nodos Index (que representan caminos de longitud 2) concatenables. Dos nodos Index u y v son concatenables cuando u.Destination = v.Source.

Para longitudes impares, se hace lo mismo, sólo que añadiendo una arista del tipo indexado al final, llamémosla r. Llamemos s y t a los extremos inicial y final de r. Entonces, r tiene que ser tal que la propiedad "id" de s sea igual a last.Destination, donde last es el último nodo de la lista de nodos Index. Si la consulta incluye un nodo de destino, entonces t debe ser este nodo.

Luego, se debe verificar que los caminos generados a partir por medio de concatenación sean continuos. Esto se prueba con la intersección de los intervalos de validez de los nodos Index de cada secuencia en el caso par. En el caso impar, en esta intersección se debe contar el intervalo de la última relación. La

reconstrucción de caminos es natural a partir de la información disponible en este punto.

Algorithm 5 retrievePathsConcat: Computes Continuous Paths from a Source Node to a Destination Node Using a length-2 Tree Index

Input: A graph G, a Tree Temporal Index I, a source node x, a destination node y, the minimum path length L_{min}, the maximum path length L_{max}, and a relationship name $relName$, a search interval $searchInterval$.

Output: A map M whose keys are paths and whose values are intervals, representing continuous paths.

Initialize a map M
for $length \in [L_{min}, L_{max}]$ **do**
  if $length\%2 = 0$ **then**
    let $C \coloneqq \{$
    $L \mid$ L is a list of INDEX nodes $\mid$
    $\forall_{N_i \in L},\ N_i.Length = 2 \wedge$
    $\forall_i \in [1, length(L) - 1],\ L[i].Destination = L[i+1].Source \wedge$
    $L.length = floor(\frac{length}{2}) \wedge$
    (y = null $\vee$ $L[length(L)].Destination = y.id$)
    $\}$
    for each $L \in C$ **do**
      if $\bigcap_i interval(L[i]) \cap searchInterval \neq \varnothing$ **then**
        $M[compute2Path(L)].insert(\bigcap_i interval(L[i]))$
      end if
    end for
  else
    let $C \coloneqq \{$
    $(L, R) \mid$ L is a list of INDEX nodes followed by a $: relName$ relationship R $\mid$
    $\forall_{N_i \in L},\ N_i.Length = 2 \wedge$
    $\forall_i \in [1, length(L) - 1],\ L[i].Destination = L[i+1].Source \wedge$
    $L.length = floor(\frac{length}{2}) + 1 \wedge$
    $L[length(L)].Destination = sourceNode(R).id$
    (y = null $\vee$ $endNode(R) = y$)
    $\}$
    for each $L \in C$ **do**
      if $\bigcap_i interval(L[i]) \cap searchInterval \neq \varnothing$ **then**
        $M[compute2PathWithFinalRelationship(L, R)].insert(\bigcap_i interval(L[i]))$
      end if
    end for
  end if
end for
return M

4.2.1.4 Sobre la correctitud de la concatenación de caminos continuos

A continuación se presenta una intuición respecto de una prueba de la correctitud del método de concatenación de caminos continuos presentado más arriba.

Para empezar, se tiene un intervalo T al cual se quiere restringir la búsqueda de caminos. La prueba asumiría que se cuenta con todos los caminos continuos de longitud 2 que tengan un período de validez cuya intersección con un intervalo T es no vacía. Estos caminos son acíclicos en todos los casos. Llamemos P_T al conjunto de estos caminos.

Se quiere probar que cualquier camino de longitud 2N (acíclicos cuyo intervalo de validez tiene una intersección no vacía con T, con N > 1) puede ser reconstruido a partir de caminos de longitud 2 con los que ya se cuenta.

Esta prueba podría hacerse por absurdo: supongamos que, de hecho, existe tal camino continuo C y que no puede ser formado a partir de los caminos de P_T. En un principio, el camino C tiene (C.length / 2) subcaminos de longitud 2 que, concatenados (eliminando ciclos resultantes), abarcan la totalidad de los nodos de C.

Por ejemplo, si

C = (Nodo1)→(Nodo2)→(Nodo3)→(Nodo4)→(Nodo5),

entonces tiene los siguientes subcaminos de longitud 2:

C1 = (Nodo1)→(Nodo2)→(Nodo3)

C2 = (Nodo2)→(Nodo3)→(Nodo4)

C3 = (Nodo3)→(Nodo4)→(Nodo5)

En este caso, concatenando C1 y C3 se tiene:

(Nodo1)→(Nodo2)→(Nodo3)→(Nodo3)→(Nodo4)→(Nodo5)

Eliminando ciclos, se obtiene C. Esto sólo sucede con C1 y C3.

Asumiendo verdadera esta propiedad, debe darse que

$C1.interval \cap C3.interval \cap T \neq \emptyset$

ya que, por cómo está definido el intervalo de validez de un camino continuo,

$C.interval = C1.interval \cap C3.interval$

Pero entonces C1 y C3 pertenecen a P_T. ¡Absurdo!

(La prueba es análoga para caminos de cualquier longitud par)

Como el absurdo vino de suponer que existe un camino continuo que no puede ser formado a partir de los caminos de P_T, entonces se concluye que cualquier camino de longitud par puede ser obtenido a partir de la concatenación de nodos de longitud 2.

Se puede usar un argumento similar para probar que, contando con la totalidad de los caminos continuos de longitud 2 y de longitud 1 (es decir, aristas) cuyos intervalos tengan una intersección no vacía con T, pueden obtenerse los caminos continuos de todas las longitudes tales que la intersección de su intervalo con T sea no vacía.

4.2.2 Índice "concat"

Dado que en muchas aplicaciones y casos de prueba resultaron en una mejor performance de coTemporalPaths comparado con retrievePathsConcat, y dadas las complicaciones para crear índices para caminos de longitud mayor a 2 en

grafos de prueba sin una gran cantidad de nodos y/o no tan conexos (por ejemplo, sólo 2500 nodos, cada uno con un promedio de 25 aristas salientes), se concibió otro tipo de índice, que, en cierta medida, comparte características con el grafo temporal en sí. En este nuevo tipo de índice,

1. Existen **nodos INDEX:META:Relationship (o nodos Meta)**, los cuales permiten comprobar si caminos válidos dentro de un cierto período de tiempo fueron indexados. Al igual que en el índice "Tree", el tipo de relación (Relationship) se especifica en el tipo de nodo. Tienen las siguientes propiedades:
   - **From:** Fecha de inicio del período de indexación
   - **To:** Fecha de finalización del período de indexación
2. Existen **nodos INDEX:Relationship (o nodos Index)**, cada uno de los cuales hace referencia a un único camino válido en un único intervalo de tiempo (sea i un nodo INDEX:Relationship, este intervalo se identifica como interval(i)). Es decir, es la misma información que en el índice "Tree" pero se agrega el tipo de relación al tipo de nodo, para no depender del nodo META para obtener este dato. Por ejemplo, para los caminos cuyo tipo de relación es Friend, los nodos Index tendrían el tipo :INDEX:Friend. La longitud de este tipo de nodos es siempre 2, y contienen las siguientes propiedades:
   - **Source:** id del nodo de inicio del camino
   - **Intermediate:** id del nodo intermedio del camino
   - **Destination:** id del nodo final del camino
   - **From:** Fecha en la que inicia el período de validez del camino
   - **To:** Fecha en la que finaliza el período de validez del camino, o 'Now' en caso de que sea válido hasta la actualidad.

   En la implementación, al igual que en el índice "Tree", para cada una de las propiedades de estos nodos (Index) existe un índice B-tree de Neo4j.
3. Un nodo u de tipo INDEX:Relationship se conecta a otro v del mismo tipo a partir de aristas de tipo :concat, si y sólo si
   - u.Destination = v.Source
   - u.interval $\cap$ v.interval $\neq \emptyset$

De esta manera, sólo es necesario calcular los caminos de longitud 2 para indexar un período, y se podría lograr una mayor rapidez en la reconstrucción de caminos de cualquier longitud que con retrievePathsConcat, al utilizar las aristas de tipo concat. Para el índice "concat", se planteó realizar consultas de la misma manera que en el grafo original, siendo que la reconstrucción de caminos debería requerir un menor esfuerzo computacional, dado que para (reconstruir) caminos de longitud N, se deberían encontrar en el índice sólamente caminos de longitud del orden de N/2 (porque cada nodo en el índice es un camino de longitud 2). En base a esto se pensaron dos algoritmos que imitarían en el índice la lógica de quickCotemporalPaths y coTemporalPaths. Estos son, respectivamente, quickConcat y BFSConcat, descritos en las siguientes secciones.

En la Figura 9, se puede ver a partir del nodo Meta (en azul) que se indexaron todos los caminos de longitud 2 válidos en algún momento entre 2000 y 2020 (inclusive). Es posible a partir del camino que empieza en el nodo A y termina en el nodo B reconstruir el camino continuo válido en el intervalo [2015-2016]

cuyos nodos Object tienen, respectivamente según su orden, ids 1, 25, 17, 38, 45, 27 y 19.

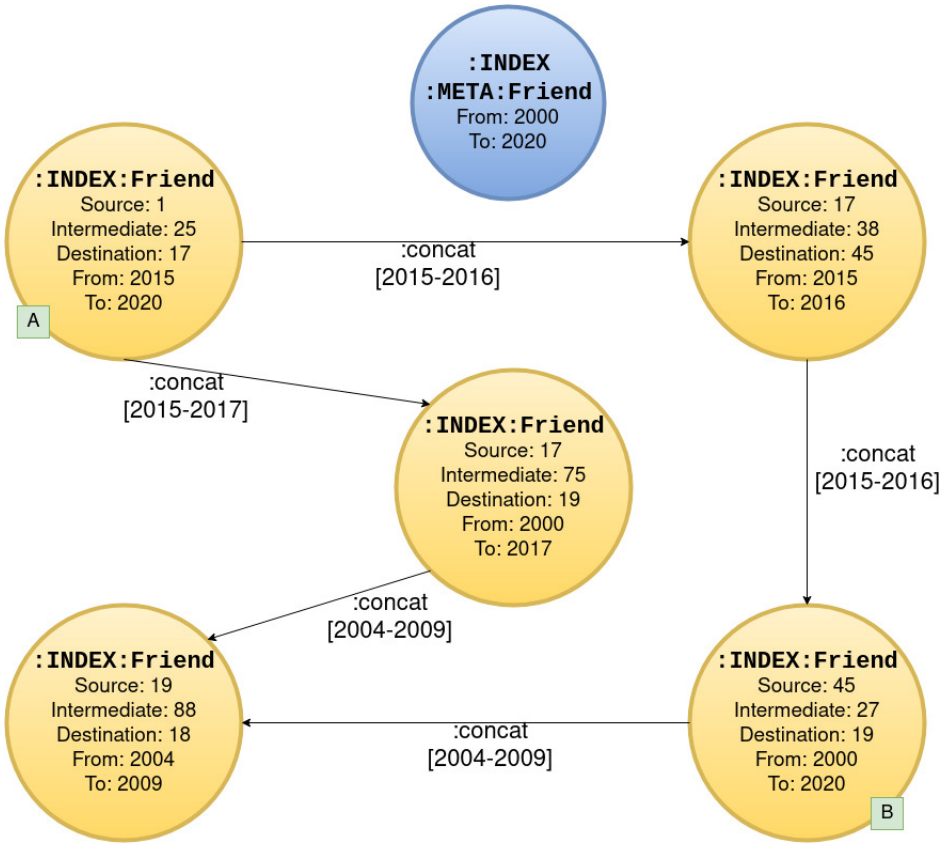

Fig. 9. Parte de un índice concat para un modelo de red social como el de la Figura 3. El intervalo de indexación, indicado en el nodo META (azul), es de 2000 a 2020. Es posible a partir del camino que empieza en el nodo A y termina en el nodo B reconstruir el camino continuo <(1, 25, 17, 38, 45, 27, 19), [2015-2016]>.

4.2.2.1 Creación del Índice

Para la creación del índice, se crea un nodo para cada par camino-intervalo de la misma manera que en el índice "Tree", sólo que sin las aristas type. En cambio, se agrega un paso extra para la creación de aristas concat, garantizando lo explicado en el punto 3 de la Sección 4.2.2. Esta última función se detalla en el Algoritmo 6. A partir de esta función, se presenta el algoritmo de creación para índices "concat" en Algoritmo 7.

En resumen, el proceso de creación de índices "concat" es el siguiente:

1. Verifica que exista un nodo Meta que para la relación y el intervalo especificados (es decir, que estas propiedades del nodo tengan el mismo valor que se busca). Si no existe se lo crea.
2. Por cada nodo Object del grafo, utiliza coTemoralPaths para calcular los caminos continuos de longitud 2 cuyos tipo de relación e intervalos de validez estén dentro de los parámetros ingresados. Por cada camino obtenido se crea un nodo Index con los datos correspondientes.
3. Para cada par de nodos Index creados en el paso anterior, llamémoslos u y v, verifica que
   - u.Destination = v.Source
   - u.interval ∩ v.interval ≠ ∅

   Si se cumplen estas dos condiciones, entonces se crea una arista concat de u a v, llamémosla r, tal que
   - r.interval = u.interval ∩ v.interval

Algorithm 6 Connect New Nodes: Connects new index nodes with index nodes of an existing Graph Index

Input: A graph index I, a set of new :INDEX nodes S
Output: A modified graph index I

for each $n \in S$ **do**
  let $C_{out} \coloneqq \{n_o \in I \cup S \mid n.Destination = n_o.Source\}$
  for each $n_o \in C_{out}$ **do**
    if $[N.From, N.To] \cap [n_o.From, n_o.To] \neq \varnothing$ **then**
      Create a :concat relationship c_R from N to n_o in I
      $c_R = [N.From, N.To] \cap [n_o.From, n_o.To]$
    end if
  end for
  let $C_{in} \coloneqq \{n_i \in I \cup S \mid n_i.Destination = n.Source\}$
  for each $n_i \in C_{in}$ **do**
    if $[N.From, N.To] \cap [n_i.From, n_i.To] \neq \varnothing$ **then**
      Create a :concat relationship c_R from N to n_i in I
      $c_R = [N.From, N.To] \cap [n_i.From, n_i.To]$
    end if
  end for
end for
return I

4.2.2.2 Quick Retrieve Paths Concat (quickConcat)

Para consultas con una cantidad de caminos relativamente baja (menor a 20), en tests provisorios se vio una performance muy superior de quickCoTemporalPaths por sobre los otros algoritmos, por lo que se planteó utilizar la misma

Algorithm 7 Creates a Graph Index for a Temporal Graph

Input: A relationship name $relationshipName$, an interval Int_{index}, a temporal graph G, a temporal index I
Output: A modified Temporal Index I
Output: A modified Temporal Index I
if $\nexists_{N_{meta}} \in I \mid N_{meta}$ is a :META:$relationshipName$ node $\mid$
$N_{meta} = \{$
$From$:Int$_{index}$.$From$,
To: $Int_{index}.To$
$\}$ **then**
Create such N_{meta}
end if
Let $IDs_{Object} \coloneqq$ {o.id$\mid$ o is an :Object node}
Initialize an empty set S
for each $id_{obj} \in IDs_{Object}$ **do**
let $L_{pairs} \coloneqq quickCoTemporalPaths(G, I, x, null, 2, 2, Int_{index})$,
$where$ x.id = id$_{obj}$
for each $(path, interval) \in L_{pairs}$ **do**
Create an :INDEX node $N_{index} \mid$
N$_{index} = \{$
Source: id_{obj},
Intermediate: [$path_1.id$],
Destination: $path_2.id$,
From: $interval.From$,
To: $interval.To$,
$\}$
$S.insert(N_{index})$
end for
end for
$connectNewNodes(I, S)$
return I, G

estrategia en la reconstrucción de caminos en el índice. De esta manera, cuando se utilizara una función built-in de Neo4j para calcular caminos de longitud N entre dos nodos (para luego podarlos de acuerdo a las condiciones de períodos y longitud), ahora, en vez de considerar N+1 nodos, se considerarían alrededor de N/2 nodos por camino. Consecuentemente, los tiempos de reconstrucción deberían disminuir considerablemente. A esta función se la llama "Quick Retrieve Paths Concat", o quickConcat. Los Algoritmos 14 y 15 del Apéndice D son funciones auxiliares que se utilizan en el pseudocódigo de quickConcat, definido en el Algoritmo 8.

El algoritmo aplica una lógica muy parecida a la de quickCPath a los nodos de un índice "concat": primero se calculan los caminos (de longitudes en el rango buscado) utilizando funciones built-in de Neo4j, luego se descartan los que no sean válidos dentro del intervalo de búsqueda. Esto es posible ya que existen las aristas concat entre los nodos Index.

Sin embargo, ya que los nodos Index representan únicamente caminos de longitud par, para los caminos de longitud impar se debe buscar la última arista en el grafo (es decir, se busca una arista entre los últimos dos nodos Object del camino).

4.2.2.3 BFS Retrieve Paths Concat (BFSConcat)

Dada la relativamente buena performance de coTemporalPaths en caminos de longitud mayor a 5, en especial para aquellos en los cuáles sólo se especifica la fuente, se pensó que se podría utilizar una estrategia similar en el índice "concat", permitiendo justamente comprimir los caminos de longitud N en subcaminos (de longitud N/2) de nodos que a su vez representan caminos de longitud 2. Como la estrategia utilizada en coTemporalPaths es básicamente una búsqueda BFS que poda (al momento de la búsqueda) los caminos que no cumplen con las restricciones de períodos de validez, longitud y/o ciclos, se creó un algoritmo que se comporta de manera parecida en el índice "concat", llamado "BFS Retrieve Paths Concat", o BFSConcat. El Algoritmo 9 explicita el pseudocódigo.

El algoritmo BFSConcat aplica la misma estrategia que coTemporalPaths (ver Sección 2.6.2 antes de leer esta sección) en los nodos Index y las relaciones concat entre ellos. Sin embargo, hay algunas diferencias a destacar:

- Cada nodo Index se traduce, en el paso inicial, a tres nodos Object con sus correspondientes referencias a nodos previos (porque representa un camino de longitud 2). En el paso inicial, la frontera va a tener más de un nodo, ya que se toman todos los nodos Index que tienen como Source al nodo inicial, y se los descompone en sus ternas de nodos. Cuando se extiende la frontera a partir de un nodo Index, sin embargo, sólo importan los dos últimos nodos del camino representado por el índice, entonces el nodo Index se traduce a sólo dos nodos Object.
- Los nodos (Object) de la frontera tienen un nodo Index asociado. Un mismo nodo Index va a estar asociado a 2 o 3 nodos Object. Si u es un tal nodo

Algorithm 8 quickRetrievePathsConcat: Computes Continuous Paths from a Source Node to a Destination Node Using a graph index

Input: A graph G, a Graph Temporal Index I, a source node x, a destination node y, the minimum path length L_{min}, the maximum path length L_{max}, and a relationship name $relName$, a search interval $searchInterval$.

Output: A map M whose keys are paths and whose values are intervals, representing continuous paths.

Initialize a map M
for $length \in [L_{min}, L_{max}]$ **do**
  if $length\%2 = 0$ **then**
    $let\ C \coloneqq \{$
      $p \mid p$ is a path of :concat relationships $|$
        $(length(p) = floor(\frac{length}{2}) - 1 \vee length == 2 \wedge length(p) = 1) \wedge$
        $p_1.Source = x.id \wedge$
        $y = null \vee p_{length(p)}.Destination = y.id$
    $\}$
    for each $p \in C$ **do**
      if $\bigcap_i interval(p_i) \cap searchInterval \neq \varnothing$ **then**
        $M[graphIndexPathToList(p)].insert(\bigcap_i interval(p_i))$
      end if
    end for
  else
    $let\ C \coloneqq \{$
    $(p, r) \mid p$ is a path of :concat relationships and $r$is a :relName relationship $|$
        $(length(p) = floor(\frac{length}{2}) - 1 \vee length == 3 \wedge length(p) = 1) \wedge$
        $p_1.Source = x.id \wedge$
        $p_{length(p)}.Destination = startNode(r).id \wedge$
        $y == null \vee endNode(r).id = y.id$
      $\}$
    for each $(p, r) \in C$ **do**
      if $\bigcap_i interval(p_i) \cap r.interval \cap searchInterval \neq \varnothing$ **then**
        $M[graphIndexPathWithFinalRelationshipToList(p, r)].insert(\bigcap_i interval(p_i) \cap r.interval)$
      end if
    end for
  end if
end for
return M

Object, decimos que u.indexNode es el nodo Index que tiene asociado. Esto permite extender el camino a partir de nodos Index y aristas concat.
- Sólo se extienden los nodos frontera u tales que u.P.length sea par. Es decir, cuando u.id = u.indexNode.Destination.
- En cada iteración, cuando se desencola un nodo u tal que u.P.length es par,
  - Si se le pueden agregar dos nodos al camino (es decir, si luego de esto su longtiud se encuentra dentro del intervalo de búsqueda), se realiza esta extensión por medio de nodos Index y aristas concat (se detalla mejor en el pseudocódigo). Para este paso se tiene en cuenta el intervalo de validez del nodo Index para determinar si se extiende o no la frontera.
  - Si se le puede agregar un nodo al camino, se realiza esta extensión por medio de una arista entre nodos Object.
- Cuando se desencola un nodo u, independientemente de si su longitud es par o impar, se verifica que su longitud esté dentro del rango buscado, y, si lo está (y este nodo es igual al nodo final del camino en el caso de una búsqueda source-destination), entonces se lo agrega a la lista de soluciones S.

4.3 Creación, actualización y eliminación de aristas de un grafo con índice temporal

También se presentan algoritmos que permiten mantener los índices actualizados en caso de agregar, eliminar o actualizar nuevas aristas. En el caso de un grafo temporal, cabe realizar las siguientes aclaraciones sobre lo que implican las operaciones de creación, actualización y eliminación de aristas:

- La creación de una arista hace referencia justamente a la creación de una relación entre dos nodos Object entre los cuales no había habido ninguna relación antes. Es decir, efectivamente se crea una nueva arista en el grafo.
- La actualización de una arista sólo puede darse en el caso de que exista una arista entre dos nodos Object cuyo período de validez haya finalizado, es decir, que no sea válido en el presente. En este caso, el efecto de actualizar la arista implica agregar al período de validez un intervalo que comience en el momento en el cual se realizó la actualización y continúe en el presente ('Now' en términos de la implementación). Esto implica que una relación que había dejado de ser válida en un punto ahora vuelve a serlo.
- La eliminación de una arista no es ni lógica ni física, sino que se trata de hacer que una arista que sea válida en el presente deje de serlo. Dicha relación pasaría a ser válida hasta el momento de la eliminación.

El pseudocódigo para creación o actualización se encuentra definido en el Algoritmo 10, mientras que el de eliminación se encuentra en Algoritmo 11.

Para la creación o actualización de aristas, el proceso es el siguiente (el tipo de las aristas es el ingresado como parámetro, salvo donde se indique lo contrario):

1. Se identifican los extremos inicial y final de la arista, llamémoslos u y v, ambos nodos Object.

Algorithm 9 BFSConcat

Input: A graph G, a Tree Temporal Index I, a source node x, a destination node y, the minimum path length L_{min}, the maximum path length L_{max}, and a relationship name $relName$, a search interval $searchInterval$.

Output: A list of Continuous Paths S.

```
let N_start := {
      N | N is an INDEX node ∧
      N.Source = x.id ∧
      searchInterval ∩ [N.From, N.To] ≠ ∅
   }
Initialize a queue F with frontier nodes in the following fashion
for each N_s ∈ N_start do
   Create a path P := [N_s.Source, N_s.Intermediate, N_s.Destination]
   let P_ip :={ path: P, interval: [N_s.From, N_s.To, indexNode : N_s]}
   F.push(P_ip)
end for
while F ≠ ∅ do
   let u := F.getFirst()
   if length(u.path) ≥ L_min ∧ (y == null ∨ y.id == getLastNodeId(u)) then
      S.insert(u)
   end if
   if length(u.path)%2 == 0 then
      if length(u.path) + 2 ≤ L_max then
         let i_N := u.indexNode
         letR_concat := {r | r is a relationship of type :concat outgoing from i_N}
         for each r ∈ R_concat do
            let intersection := u.interval ∩ r.interval
            if intersection ∩ searchInterval ≠ ∅ then
               leti_N2 := endNode(r)
               if ¬u.path.contains(i_N2.Intermediate)∧¬u.path.contains(i_N2.Destination)
               then
                  F.add({path : u.path.extend([i_N2.Intermediate, i_N2.Destination]), interval :
                  intersection, indexNode : i_N2})
               end if
            end if
         end for
      end if
      if length(u.path) + 1 ≥ L_min ∧ length(u.path) + 1 ≤ L_max then
         letR := { r |r is a relationship of type :relationshipName outgoing from a
         node with id = getLastNodeId(u)}
         for each r ∈ R do
            letintersection:= u.interval ∩ r.interval
            if intersection ∩ searchInterval ≠ ∅ then
               letv := endNode(r)
               if ¬u.path.contains(v.id) then
                  F.add({path: u.path.extend([v.id]), interval: intersection, indexNode:
                  null})
               end if
            end if
         end for
      end if
   end if
end while
return S
```

2. Si existe una arista r entre u y v que no sea válida en la actualidad (Ver Apéndice B), se la hace válida. De lo contrario, se crea una nueva arista (válida) entre u y v.
3. Luego de esto, es necesario actualizar el índice (de caminos de longitud 2 únicamente). Los pasos siguientes pueden realizarse en cualquier orden:
   – Por cada nodo t tal que existe una relación válida de t a u, se crea un nodo Index que represente el camino (t, u, v).
   – Por cada nodo w tal que existe una relación válida de v a w, se crea un nodo Index que represente el camino (u, v, w).
4. Para estos nuevos nodos creados, se agregan las relaciones necesarias en el índice "concat" tales que se garantice el punto 3 de la Sección 4.2.2.

Para la eliminación de aristas, se siguen estos pasos:

1. Se parte de una arista válida en la actualidad que se quiere "eliminar", llamémosla r. Llamemos t_d al momento de comenzar a ejecutar esta función.
2. Se toman todos los nodos Index que representen caminos válidos en la actualidad que incluyan a esta arista (es decir, nodos i tales que i.To = 'Now'), y se modifica su intervalo de validez para que termine en t_d, es decir, no sean más válidos en la actualidad.
3. Cualquier relación de tipo concat que tenga como extremo a alguno de los nodos del paso anterior y sea válida también se modifica para que su intervalo de validez termine en t_d.

4.4 Integración con TGQL

A más alto nivel, lo que se busca es proveer a TGQL de funcionalidades de indexación. Para esto es necesario extender el lenguaje con reglas que permitan reconocer sentencias de código para la creación, actualización, utilización y eliminación de índices. Se cuenta con una implementación que permite consultar el índice antes de realizar una consulta directa a la base de datos. Como sólo se tiene en cuenta la función cPath, esto sólo aplica a llamados a esta función en TGQL. De esta manera, agregando las reglas correspondientes en la declaración de sintaxis de antlr4 y ampliando la implementación del analizador sintáctico, se puede traducir cada llamada a cPath a una consulta en Cypher con la siguiente lógica:

1. Si existe un índice para los parámetros con los que se llamó a cPath, devolver caminos reconstruidos a partir del índice
2. Si no existe tal índice, buscar los caminos utilizando coTemporalPaths

Es decir, se logró integrar consultas al índice para caminos continuos al lenguaje TGQL. También es posible realizar las consultas de creación para ambos tipos de índices, y de creación, actualización y eliminación de aristas. No obstante, el foco del proyecto pasa a ser la comparación de performance de las *procedures* que implementan las funciones mencionadas en las secciones 4.1 a 4.3 inclusive

Algorithm 10 Creates or Updates Edge

Input: A source node x, a destination node y, a relationship name $relName$, a Temporal Graph G, a Temporal Index I

Output: A modified graph index I, a modified graph G

let t_d be the current time
if $\exists_r \mid r$ is a : $relName$ edge $\wedge\ startNode(r) = x \wedge endNode(r) = y \wedge r.To \neq Now$ **then**
  $r.interval.insert(t_d, Now)$
else if $\nexists_r \mid r$ is a : $relName$ edge $\wedge\ startNode(r) = x \wedge endNode(r) = y$ **then**
  Create a new edge r going from x to y
  $r.interval = (t_d, Now)$
  if $\nexists_{N_{meta}} \mid N_{meta}.To = Now$ **then**
    return I, G
  end if
  Initialize an empty set S
  for each edge r_{in} incoming to $x \mid \mathrm{r}_{in}.To = Now$ **do**
    Create an index node $\mathrm{N}_i = \{$
    $From$:t_d,
    To: Now,
    Length: 2,
    Source: $startNode(r_{in}).id$,
    Intermediate: [$x.id$] if I is a tree index, $x.id$ if I is a graph index
    Destination: $y.id$
    $\}$
    $S.insert(N_i)$
  end for
  for each edge r_{out} outgoing from $y \mid \mathrm{r}_{out}.To = Now$ **do**
    Create an index node $\mathrm{N}_i = \{$
    $From$:t_d,
    To: Now,
    Length: 2,
    Source: $x.id$,
    Intermediate: [$y.id$] if I is a tree index, $y.id$ if I is a graph index
    Destination: $endNode(r_{out}).id$
    $\}$
    $S.insert(N_i)$
  end for
end if
if I is a Graph Index **then**
  $connectNewNodes(I, S)$
end if
return I, G

Algorithm 11 Removes an Edge from a Temporal Graph

Input: An edge r, a Temporal Graph G, a Temporal Index I
Output: A modified Temporal Index I, a modified Temporal Graph G

let t_d be the current time
let $S \coloneqq \{\mathrm{n}_i \mid n_i$ is an INDEX node $\wedge n_i.Source = startNode(r).id \wedge n_i.Intermediate = endNode(r).id \wedge n_i.To = Now\}$
let $D \coloneqq \{\mathrm{n}_i \mid n_i$ is an INDEX node $\wedge n_i.Intermediate = startNode(r).id \wedge n_i.Destination = endNode(r).id \wedge n_i.To = Now\}$
for each $n_i \in S \cup D$ **do**
  $n_i.To = t_d$
end for
if $IisaGraphIndex$ **then**
  for each $n_i \in S \cup D$ **do**
    for each $rel \mid rel$ is a relationship with an endpoint in i_n **do**
      $rel.interval.replace(Now, t_d)$
    end for
  end for
end if
return I, G

dentro del entorno de Neo4j. En línea con esto, todas las pruebas son realizadas a partir de consultas en el lenguaje Cypher directamente sobre una base de datos de Neo4j. Las extensiones a TGQL pueden encontrarse en el Apéndice A.

5 Pruebas

Se realizaron pruebas sobre tres datasets que modelan redes sociales, cada uno de diferente tamaño y conectividad entre nodos. Uno de ellos fue creado a partir de datos reales de Twitter, mientras que los otros dos son generados artificialmente por medio de un generador de datasets basado en el presentado en [5].

5.1 Twitter Dataset

Este dataset se generó a partir de lecturas periódicas de datos de twitter. En particular, lo que se hizo fue, partiendo desde la cuenta @ITBA, obtener las últimas 10 cuentas a las cuales siguiera. Luego, para cada una de estas 10 cuentas, tomar las 10 últimas cuentas que siguiera cada una, y así hasta llegar a una profundidad de 4, según una modalidad BFS. El orden de las cuentas (para determinar cuáles son las 10 últimas cuentas) es de más a menos reciente, es decir, se toman las 10 últimas cuentas a las cuáles siguió (y sigue actualmente) el usuario en cuestión. A este proceso se le llamará "lectura de Twitter" o "lectura" de ahora en adelante. A partir de esto, se pueden determinar los nodos Object y las relaciones entre ellos, para las cuales se utilizó la etiqueta "Follows". Haciendo

esto en distintos puntos en el tiempo, se pueden determinar los intervalos de validez de las relaciones Follows de la siguiente manera: Siendo t1 y t2 dos instantes, donde t1 < t2 y no existe un t3 tal que t1 < t3 < t2 y se haya realizado una lectura de Twitter en t3, y sean u y v dos nodos Object,

- Si entre u y v no existe relación alguna al empezar la lectura de Twitter en t2, pero durante esta lectura se tiene que v es una de las últimas 10 cuentas a las cuales siguió u, entonces se crea una relación Follows de u a v con intervalo de validez [t2, Now].
- Si entre u y v existe una relación al empezar la lectura de Twitter en t2 tal que el intervalo más reciente de su período de validez es de la forma [ti, tf], donde tf es distinto de Now, y en la misma lectura se determina que v es uno de las últimas 10 cuentas que siguió u, entonces se modifica en este intervalo el valor de tf por Now. En el caso en el que tf fuera Now, este valor no se modifica.
- Si existe una relación Follows de u a v que sea válida en el tiempo t1 pero no lo sea en t2 (si y sólo si v es uno de las 10 primeras cuentas seguidas por u en t1 pero no en t2), entonces esta misma relación pasa de tener un intervalo [td, Now] a tener un intervalo [td, t2].

A partir de esto, con lecturas entre abril y julio, se pudo construir un grafo de más de 30000 nodos Object (37746 exactamente), modelando una parte de Twitter centrada en el usuario @ITBA. Sobre este grafo se probó construir ambos tipos de índices, tanto el "Tree" como el "concat", ambos sólamente para caminos de longitud 2, abarcando todo el período de vida del grafo (desde Abril de 2021 hasta Agosto de 2021). La cantidad de caminos de longitud 2 indexados fue 88052.

5.2 Social Network Model

El segundo grupo consiste de datasets donde las cantidades de nodos no superan los 40000, mientras al mismo tiempo se busca abarcar más tipos de nodos en cuanto a cantidad de aristas salientes y entrantes, caminos continuos que los contengan y otros criterios más específicos. La razón de la reducción en la cantidad de nodos fueron las limitaciones para generar grafos (y sus correspondientes índices) frente a costos computacionales desmesurados al aumentar la conectividad de los grafos. Los resultados presentados, sin embargo, corresponden a un único dataset que es generado de la siguiente manera (los parámetros son configurables):

- Primero se crea una componente C1 de N nodos, para cada uno de los cuales se garantiza que el grado de salida sea como mínimo RFriend. También se garantiza que exista al menos una cantidad parametrizable de caminos continuos para ciertas longitudes (parametrizables).
- Se crea una componente C2 que N/10 nodos cuyo mínimo grado de salida es RFriend/10. También, para cada longitud de caminos continuos especificados

para C1, se garantiza en C2 la cantidad de caminos continuos especificados para esa longitud dividido 10. Es decir, C2 modela una parte del grafo menos conexa, donde las dimensiones son de alrededor del 10
- Por cada nodo u de C2 se crea una relación Friend de dirección aleatoria entre u y un nodo v de C1 elegido al azar.
- Además, se crea un nodo llamado "estrella" que tiene una cantidad determinada de aristas Friend entrantes (la cantidad de este tipo de nodos, Star, al igual que la mínima cantidad de aristas entrantes que tienen, ROutstar, también es configurable). Se garantiza que este nodo tenga también un número (bajo) de aristas salientes.
- Asimismo, se busca un nodo de entre los ya creados y se crean aristas salientes del mismo, simulando un usuario que sigue a muchos otros.

Las pruebas se corrieron sobre dos datasets de diferentes tamaños. A continuación, sus configuraciones: MEDIUM:

- N = 35000
- RFriend = 25
- Star = 1
- ROutstar = 20000

En total se tienen 38500 nodos Object y 1229171 caminos de longitud 2 (los cuales también fueron indexados). SMALL:
N = 5000
RFriend = 15
Star = 1
ROutstar = 1000 En total se tienen 5500 nodos Object y 37683 caminos de longitud 2. En ambos datasets sólo se realizaron pruebas con el índice "concat".

6 Resultados

A continuación se presentan los resultados más relevantes de pruebas llevadas a cabo sobre cada uno de los datasets descritos anteriormente. En el resto del informe, por simplicidad, para abreviar el término #(caminos continuos) se utilizará la expresión #CC.

6.1 Twitter Dataset

Para cada tipo de índice, se realizaron dos tipos de pruebas: una con caminos de los que se especifican tanto Origen como Destino (source-destination), y una con caminos donde sólo se especifica el Origen (source-only). Para los caminos source-destination, se procedió a realizar consultas para cálculo de caminos continuos en un par distinto por cada longitud, ya que no se encontró un par de nodos que tuvieran entre sí caminos de todas las longitudes a considerar. Para los caminos source-only, se tomó un único nodo y se fueron calculando los caminos que empiezan en éste para cada longitud de entre 2 y 9.

6.1.1 Índice "Tree"

Los resultados para el uso del índice "Tree" pueden verse en las Tablas 1 y 2, donde se resaltan, por cada longitud de camino, los tiempos de cada método y la cantidad de caminos continuos en el resultado. Estos resultados se grafican en las Figuras 10 y 11.

Source-Destination				
Length	retrievePathsConcat	coTemporalPaths	quickCPath	#CC
2	13,5	44	15,25	1
3	13,5	18,5	11,5	1
4	33,25	169,75	37,25	1
5	22,75	563,5	25,25	4
6	121,75	180,75	151,5	15
7	1296,25	4656	1495	219
8	136,75	1297,75	210	6
9	73	37,5	159,75	1
10	93,25	5393,75	168,25	2
11	1246,75	15833	1937,25	34

Table 1. Resultados para caminos Source-Destination del dataset de Twitter con índice "Tree"

Source-Only				
Length	retrievePathsConcat	coTemporalPaths	quickCPath	#CC
2	422,5	317	459	155
3	3734,75	3450,75	3757,5	1431
4	1145,75	1160,75	1185,25	327
5	1780,75	1735,75	2530	385
6	2473,25	2498,5	2843,75	481
7	2949,5	2769,75	5486,75	526
8	2416,25	2136,5	4080	412
9	7001,25	5863	15078	1596

Table 2. Resultados para caminos Source-only del dataset de Twitter con índice "Tree"

Para caminos source-destination, puede observarse que en casi la totalidad de los casos, los tiempos de ejecución son menores si se utiliza quickCPath o quickConcat, siendo esta última la ganadora para caminos de longitud más alta, como se puede ver en la Figura 10.

En el caso de los caminos source-only, queda claro a partir de la Figura 11 que no es beneficioso crear un índice "Tree" en el grafo para mejorar los tiempos. Asimismo, quickCPath (que no utiliza un índice), queda totalmente descartada.

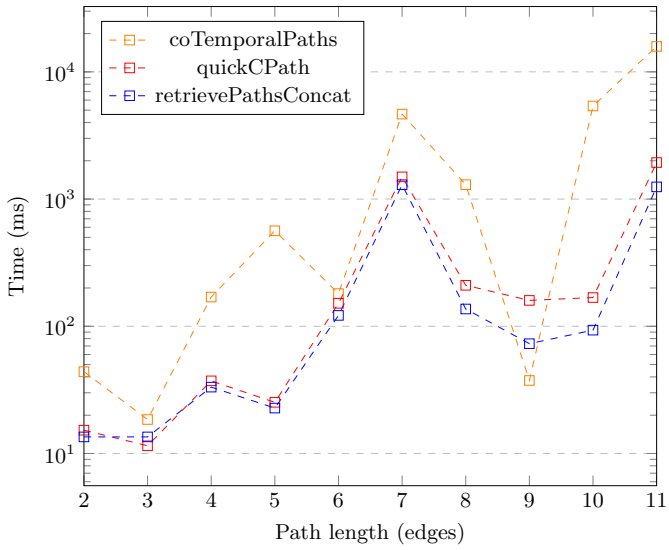

Fig. 10. Twitter - Tiempo vs. Longitud de Camino - Caminos Source-Destination - Índice "Tree"

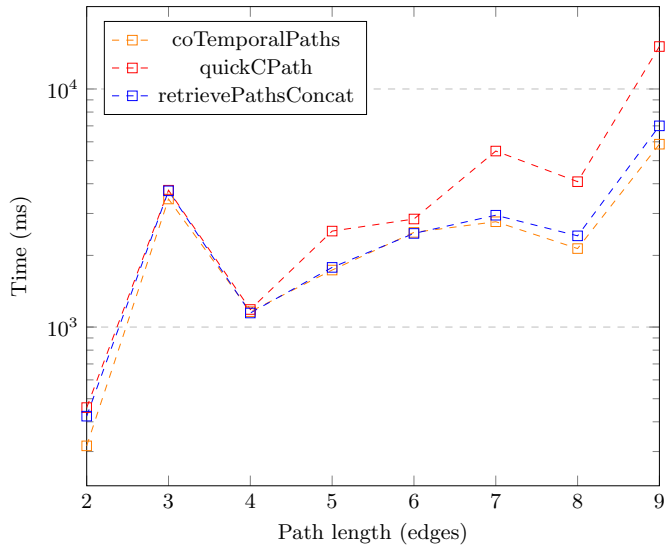

Fig. 11. Twitter - Tiempo vs. Longitud de Camino - Caminos Source-only - Índice "Tree"

6.1.2 Índice “concat”

Los resultados para el uso del índice “concat” pueden observarse en las en las Tablas 3 y 4 y en las Figuras 12 y 13.

Source-Destination					
Length	coTemporalPaths	quickCPath	quickConcat	BFSConcat	#CC
2	45	27,5	13,75	96,75	1
3	18,75	14,25	12,25	28	1
4	121,75	10,5	11,5	67	1
5	471,25	115,25	60,5	562,5	4
6	180,5	164,5	142,75	158,75	15
7	3954,5	1339,75	1124,5	3623,5	219
8	1093	190	88,75	298,25	6
9	34,75	133	32,5	55,25	1
10	4593,25	187,75	49	1432,5	2
11	14208,25	1803	528,5	14394	34

Table 3. Resultados para caminos Source-Destination del dataset de Twitter con índice "concat"

Source-Only					
Length	coTemporalPaths	quickCPath	quickConcat	BFSConcat	#CC
2	408,5	378	392	535,25	155
3	3531,25	3630,5	3362,25	3447,75	1431
4	1154	1140,75	1039,25	1140,75	327
5	1568,5	2432	1633,75	1508,5	385
6	2198,75	2817,75	2248	2251	481
7	2569,75	5198,75	2806,75	2504,25	526
8	2084,25	3880	2231,75	2036,5	412
9	5783,5	14227,5	6273,25	6153,75	1596

Table 4. Resultados para caminos Source-only del dataset de Twitter con índice "concat"

En cuanto a los caminos source-destination, puede observarse (ver Figura 12) que, en casi la totalidad de los casos, quickConcat tiene una performance superior a la de coTemporalPaths, incluso reduciendo los tiempos a menos de la mitad en la mayoría de estos. También cabe destacar la performance de quickCPath para caminos de longitud menor a 7.

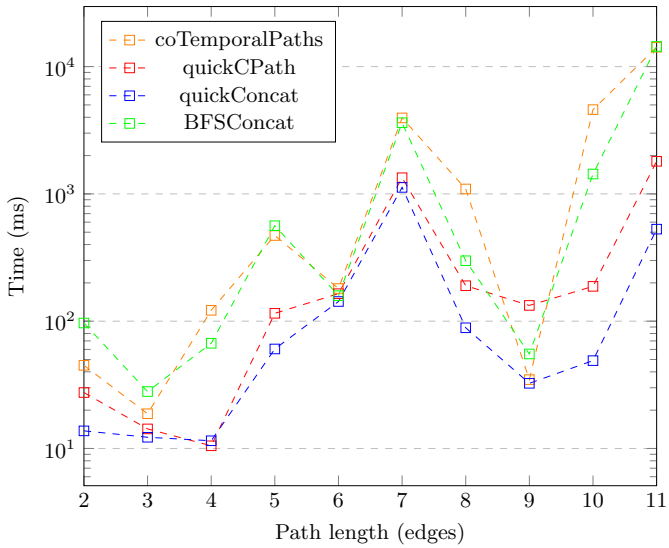

Fig. 12. Twitter - Tiempo vs. Longitud de Camino - Caminos Source-Destination - Índice "concat"

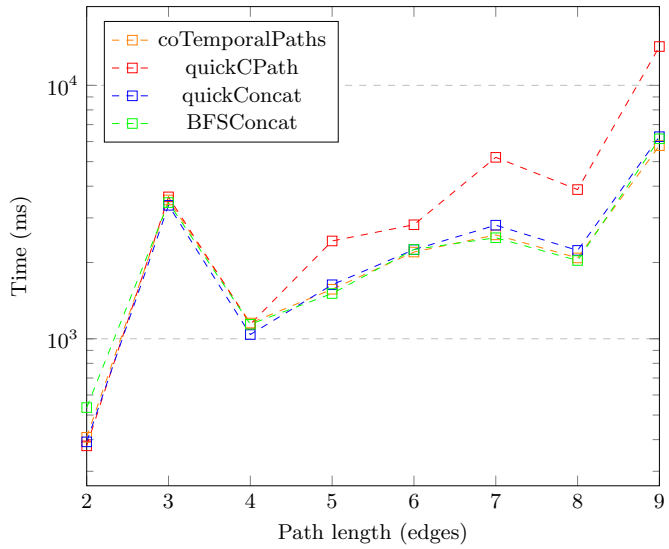

Fig. 13. Twitter - Tiempo vs. Longitud de Camino - Caminos Source-only - Índice "concat"

En el caso de los caminos source-only es claro que ninguna alternativa a coTemporalPaths mejora considerablemente los tiempos de ejecución (Figura 13).

6.2 Social Network Model - MEDIUM

En el caso del dataset del segundo grupo, se buscó caracterizar los nodos según su grado de entrada y salida, de manera que se pudiera determinar si este factor es un determinante del tiempo de las consultas. Entre todos los nodos, se tienen las siguientes clases:

- STAR: se trata de los nodos que tienen una gran cantidad de aristas de entrada, distintivamente mayor que el promedio. En este dataset hay un único nodo STAR, que tiene grado de entrada de 23860.
- FOLLOWER: estos nodos, como contraparte de los nodos STAR, tienen un grado relativamente alto de salida. En este dataset existe un único nodo FOLLOWER, que tiene un grado de salida de 1394.
- BIG-REGULAR: son nodos que se encuentran en el subgrafo C1 del grafo, es decir, la parte de mayor conectividad del grafo.
- SMALL-REGULAR: son nodos que se encuentran en el subgrafo C2 del grafo, es decir, la parte de menor conectividad del grafo.

A partir de estas clases de nodos, se pueden clasificar los caminos a evaluar según la clasificación de sus extremos. Se consideraron 3 clases de caminos con resultados interesantes:

1. BIG-REGULAR → BIG-REGULAR: Como su nombre lo indica, se trata de caminos que van desde un nodo BIG-REGULAR a otro nodo BIG-REGULAR.
2. REGULAR → FOLLOWER → STAR: Estos caminos comienzan en un nodo REGULAR y terminan en un nodo STAR. Tienen la particularidad de que, para los pares de nodos tomados como extremos, existe al menos un camino que pasa por un (el) nodo FOLLOWER.
3. FOLLOWER → STAR: Se trata de los caminos que van desde un nodo FOLLOWER a un nodo STAR.

6.2.1 MEDIUM - Clase 1

Para este caso, se tomaron dos pares de nodos. Para cada método, se obtuvieron los tiempos promedio de 5 ejecuciones de cada uno de los métodos mencionados en la tabla. Al mismo tiempo, se tiene en cuenta la cantidad de caminos continuos (#CC) y la cantidad de caminos en total (paths) entre los dos nodos. Los resultados pueden verse en las Tablas 5 y 6.

En ambos casos se puede observar un claro ganador en la función quickConcat (ver Figuras 14 y 15). La naturaleza del crecimiento de los tiemops de ejecución respecto de la longitud de los caminos es exponencial, pero esto sólo se da porque

Par 1						
Length	coTemporalPaths	quickCPath	quickConcat	BFSConcat	#CC	#paths
2	23,4	5,8	3,8	21,8	0	1
3	84	34,8	12,6	94	2	34
4	539,6	698	20,8	163,2	3	507
5	3648,6	16590,8	352,8	3457,6	31	10406
6	22959	391973,4	711,2	6819,8	161	224871

Table 5. Resultados para el Par 1 de la Clase 1 del dataset MEDIUM Social Model.

Par 2						
Length	coTemporalPaths	quickCPath	quickConcat	BFSConcat	#CC	#paths
2	18,8	6,4	5,4	35	3	4
3	152,2	38,8	13,2	180	3	27
4	1347	641,4	23,4	381,6	6	451
5	8283,4	15524,2	339,8	7683,8	38	9273
6	46438,4	366199,6	811,4	13949,2	219	204215

Table 6. Resultados para el Par 2 de la Clase 1 del dataset MEDIUM Social Model.

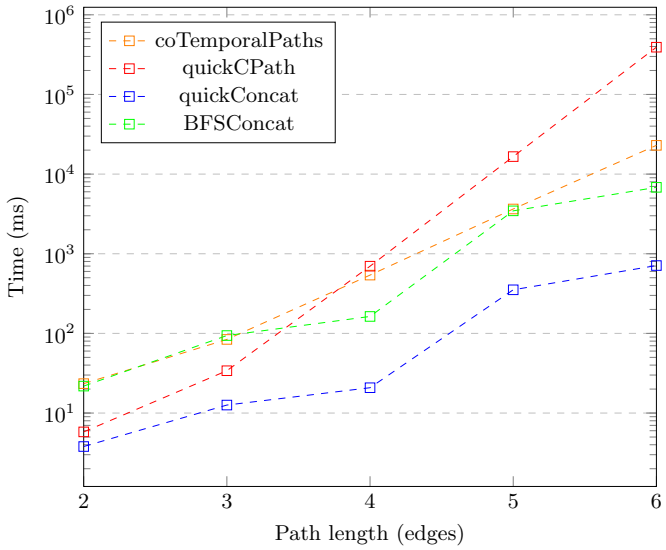

Fig. 14. Clase 1 - Tiempo vs. Longitud de Camino - Par 1

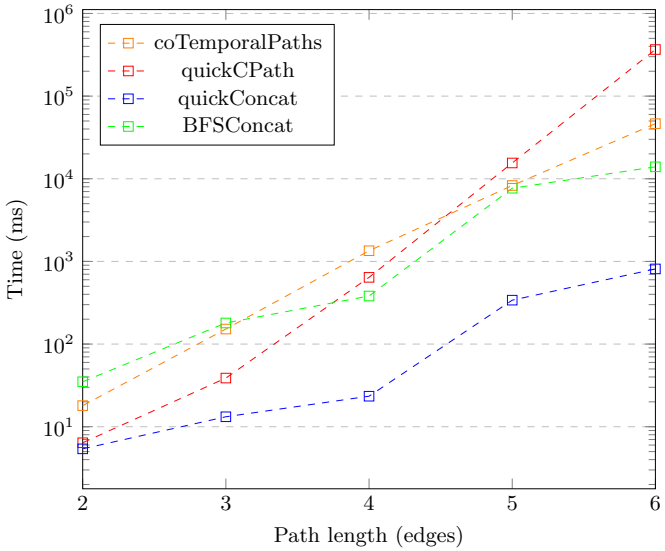

Fig. 15. Clase 1 - Tiempo vs. Longitud de Camino - Par 2

el aumento de caminos continuos también es exponencial a medida que aumenta la longitud, para este grafo. Como puede verse en este y en el resto de los resultados, los tiempos de ejecución dependen fuertemente de la cantidad de caminos continuos, salvo en el caso de quickCPath, donde parecieran depender de la cantidad total de caminos.

6.2.2 MEDIUM - Clase 2

Para la segunda clase, también se consideraron dos pares de nodos. Los resultados obtenidos pueden verse en las Tablas 7 y 8.

Par 1						
Length	coTemporalPaths	quickCPath	quickConcat	BFSConcat	#CC	paths
2	17,6	231,4	81,4	21	11	36
3	114,6	3324,4	821	127,6	93	1867
4	778,8	54721,6	4179,2	411	720	34503
5	5895	>300000	32687	5649	4479	670830
6	47030,6	>300000	177293,2	28638,8	35514	13875416

Table 7. Resultados para el Par 1 de la Clase 2 del dataset MEDIUM Social Model.

Par 2						
Length	coTemporalPaths	quickCPath	quickConcat	BFSConcat	#CC	paths
2	53	232,6	82,4	129,2	6	22
3	1159,4	3902	1476,4	1300,6	916	1439
4	7829,4	59490,8	8865,2	5798,4	5924	23140
5	40680,2	1112084	56050	41293	28596	394900
6	209160,8	NaN	293181,2	152786,4	135962	NaN

Table 8. Resultados para el Par 2 de la Clase 2 del dataset MEDIUM Social Model.

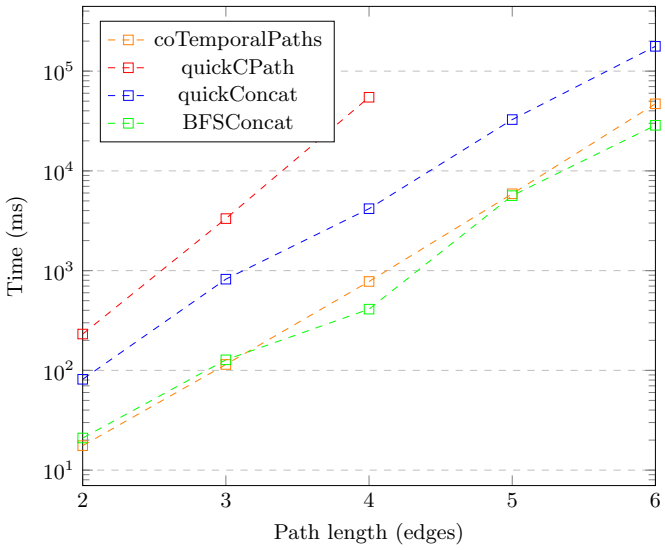

Fig. 16. Clase 2 - Tiempo vs. Longitud de Camino - Par 1

Para esta clase, los resultados fueron sustancialmente distintos a los de la primera. En este caso, quickConcat no tuvo una performance definitivamente mejor, y esta incluso fue claramente peor a coTemporalPaths (Figura 16) en ciertos casos. Lo que pudo observarse fue una performance superadora de BFS-Concat para caminos más largos y de longitud par, aunque no lo es por mucho, como puede verse en ambas Figuras 16 y 17. Otra observación es que en el Par 1, el ratio #CC/#paths es considerablemente menor al del Par 2.

6.2.3 MEDIUM - Clase 3

Como en este dataset hay sólo un nodo STAR y uno FOLLOWER, se tuvo en cuenta sólo este par para evaluar resultados, los cuales pueden observarse en

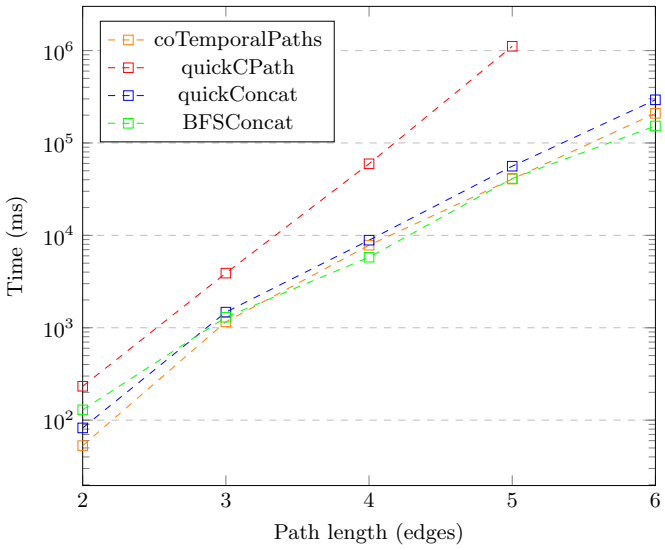

Fig. 17. Clase 2 - Tiempo vs. Longitud de Camino - Par 2

la Tabla 9. Entre otras cosas, esta tabla pone en evidencia la gran cantidad de caminos continuos que existen entre los dos nodos de la Clase 3.

Length	coTemporalPaths	quickCPath	quickConcat	BFSConcat	#CC	paths
2	1077,6	833,8	663,6	1508	887	946
3	7183,8	7974,8	5006,6	7945	5782	12452
4	37585,6	84942,4	27261	26878	27486	158103
5	192485,4	NaN	147569,2	197891,4	129641	2423565
6	517204	NaN	237862	554983	615661	43520407

Table 9. Resultados para el par FOLLOWER → STAR del dataset MEDIUM Social Model.

En este caso, como puede verse en la Figura 18, las performances de todos los métodos fueron similares, con quickConcat siendo levemente superior en la mayoría de los casos.

6.2.4 Otros datasets e índices

Se realizaron pruebas para los mismos tipos de caminos, tanto en MEDIUM con varios índices de períodos más cortos (de un mes de duración) como en SMALL. Para el caso de MEDIUM, independientemente del tamaño del índice, las performances relativas de los métodos se mantuvieron iguales. En el caso del

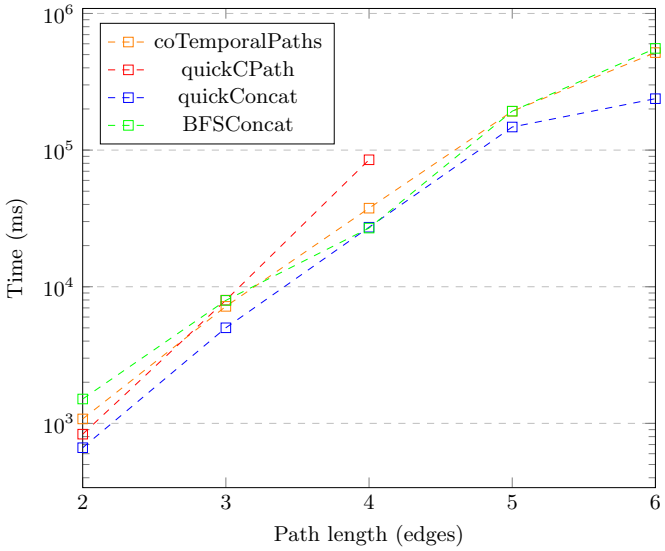

Fig. 18. MEDIUM - Clase 3 - Tiempo vs. Longitud de Camino

dataset SMALL, se pudo observar siempre una gran disminución de los tiempos de quickConcat respecto de coTemporalPaths, independientemente del tipo de camino. Por ejemplo, los resultados de quickConcat y coTemporalPaths para los caminos de la Clase 3 pueden verse en la Tabla 10, y la clara mejora en performance introducida por quickConcat se grafica en la Figura 19.

Length	coTemporalPaths	quickConcat	#CC	paths
2	267	6,4	154	169
3	1511	32	715	2696
4	7818	287,8	3648	56862
5	29032	1297,8	19350	1195918
6	187439	10924	100739	25460598

Table 10. Resultados de quickConcat y coTemporalPaths para caminos de Clase 3 del dataset SMALL Social Model.

6.3 Explicación de los resultados

Los resultados pueden entenderse mejor teniendo en cuenta la siguiente clasificación de las funciones de cálculo de caminos continuos en grupos:

1. Por un lado, se tienen las funciones que utilizan una estrategia BFS. Se trata de coTemporalPaths y BFSConcat. Para calcular caminos, estas uti-

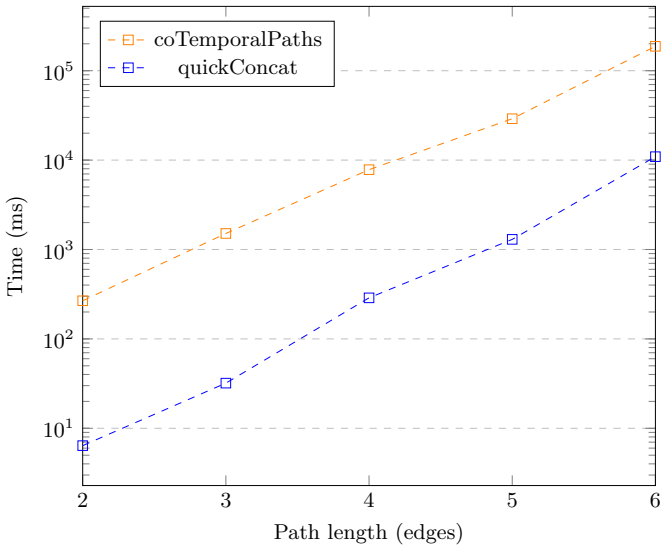

Fig. 19. SMALL - Clase 3 - Tiempo vs. Longitud de Camino

lizan, como fue dicho más arriba, una estrategia BFS. La diferencia con una búsqueda BFS normal, sin embargo, es que, al extender la frontera, los nodos que no generen un camino continuo de longitud en el rango especificado que tenga origen en el nodo fuente especificado no son tenidos en cuenta. De esta manera, se ahorra al máximo la cantidad de caminos evaluados, pero se tiene un overhead en la reconstrucción final de los mismos a partir del árbol resultante.

2. Por otro lado, se tienen las funciones que utilizan la función de cálculo de caminos built-in de Neo4j. Estas son quickCPath y quickConcat. La estrategia implica obtener todos los caminos de las longitudes deseadas que empiecen en un nodo especificado (y terminen en otro nodo especificado, si lo está), y luego eliminar los que no sean continuos y/o no cumplan con los requisitos inferidos a partir de los parámetros de la función.
3. Finalmente, se cuenta con una función única en su clase, treeConcat. Esta función busca caminos concatenando directamente los nodos de un índice (en la implementación sólo funciona con nodos de longitud 2). A alto nivel, lo que se hace es concatenar caminos de longitud 2 uniendo los que terminan en un nodo X con los que empiezan en ese mismo nodo X. Después de este paso, se tienen los mismos caminos que en las funciones del segundo grupo, por lo que también hay que eliminar los que no cumplan con los requisitos especificados.

Al mismo tiempo, es necesario considerar que, para los métodos que utilizan el índice "concat", los caminos de longitud par se calculan más rápidamente que los de longitud impar. Esto se debe a que, al concatenar caminos de longitud 2, sólo

pueden construirse directamente a partir del índice otros caminos de longitud par. En el caso de que la longitud sea impar, estos se computan obteniendo la última arista del camino en el grafo sobre el cual se calcula el índice. Esto implica un aumento en el costo computacional, lo cual se refleja en varios resultados, y sobre todo en los de BFSConcat.

6.3.1 Performance de quickConcat y su relación con la cantidad de caminos continuos Source-Destination

Por lo general, se pudo observar una muy buena performance de quickConcat en caminos de Clase 1, llegando esta a ser más de 10 veces más rápida que coTemporalPaths para algunas consultas. Si se incluyen los resultados de la base de datos SMALL, se puede observar que lo que tienen en común los pares de nodos para los cuales quickConcat es claramente la mejor alternativa es que la cantidad de caminos continuos encontrados es relativamente "baja". De esta manera, incluso para pares de nodos de Clases 2 y 3, mientras en MEDIUM pueden observarse resultados parecidos a o peores que con coTemporalPaths, en SMALL estos son definitivamente mejores a favor de quickConcat.

En vistas a la ambigüedad que representa el concepto de "baja" cantidad de caminos continuos, se propuso obtener un posible umbral para esta, determinado por la proporción de resultados positivos obtenidos en distintas pruebas para consultas que tuvieran como resultado una cantidad de caminos menor a este mismo umbral. Más genéricamente, se buscó también un umbral para la razón #CC/Object, donde Object es la cantidad de nodos Objeto del grafo en cuestión. A partir de 54 pruebas incluyendo los datasets de Twitter y Social Model - MEDIUM y SMALL -, se determinó que, para una razón #CC/Object menor o igual a 1.5%, los tiempos de quickConcat son menores a los de coTemporalPaths en el 90% de los casos, y disminuyen los tiempos de coTemporalPaths a menos de la mitad en el 81% de estos. Para las bases de datos utilizadas, esto se corresponde con una cantidad de caminos continuos de alrededor de 700. Así, se cuenta con condiciones concretas en las que sea conveniente utilizar el índice.

La dependencia de quickConcat respecto de la cantidad de caminos continuos tiene una explicación clara: los caminos de longitud 2 cuya información está contenida en el índice son únicamente caminos continuos. De esta manera, al buscar este tipo de caminos, se considera una menor cantidad de caminos no continuos que luego son descartados. Si bien coTemporalPaths tampoco considera todos los posibles caminos no continuos, el hecho de que el método BFS haga aumentar los caminos una arista a la vez implica una mayor probabilidad de estar considerando más caminos no continuos a la hora de realizar el cálculo.

En relación con esto, se recomienda mantener una ventana de indexación reducida y específica, ya que esto facilita disminuir la cantidad de caminos continuos indexados.

6.4 Conclusiones

La viabilidad del uso de un índice para caminos continuos en un grafo temporal depende de la tolerancia del usuario al crecimiento del mismo índice, que depende de la cantidad de caminos continuos que pertenecen al intervalo de indexación y cuya longitud se encuentra dentro del rango indexado.

En la mayoría de los casos de uso, este crecimiento es tal que indexar caminos de longitudes mayores a 2 no es costeable en términos tanto temporales (para generar el índice) como espaciales. Esta limitación, sin embargo, es subsanada por el hecho de que se pueden concatenar caminos de longitud 2 para formar otros de mayor longitud, pudiendo mantener una buena performance en ciertos casos.

En particular, es posible utilizar el índice "concat" para optimizar consultas cuya cantidad de resultados es menor a un 1.5% de la cantidad de nodos Objeto, a partir de la concatenación de caminos indexados de longitud 2 (quickConcat).

Por otro lado, es posible utilizar métodos built-in de Neo4j sin un índice (quickCPath) para optimizar los tiempos de las consultas. Esto puede verse, sin embargo, sólo en consultas entre pares de nodos tales que la cantidad de caminos totales entre ellos sea reducida (menor a 50 en la práctica).

El desarrollo futuro podría incluir mejoras en el almacenamiento de los caminos continuos, por ejemplo, conteniendo todos los intervalos de validez de un camino entre dos nodos Object en un mismo nodo del índice, a diferencia de la implementación actual, que crea un nodo del índice por cada intervalo.

Podrían investigarse mejoras introduciendo la distribución de la base de datos, por ejemplo, dedicando una partición exclusivamente a los nodos del índice y otra a los nodos del grafo original (y sus respectivas relaciones).

Asimismo, podría investigarse la performance y requerimientos de espacio y tiempo de creación de un índice con nodos de longitudes 2 y 3 únicamente para ciertos nodos fuente en períodos más reducidos a los utilizados en el presente proyecto (por ejemplo, como máximo un mes). El hecho de contar con caminos de longitud 3 permitiría no tener que buscar una relación en el grafo original al calcular caminos de longitud impar.

Finalmente, sería interesante el desarrollo de un índice que, en vez de almacenar todos los caminos para todos los pares de nodos en un intervalo dado, sólo conserve una parte reducida pero relevante de estos, como por ejemplo, los caminos más cortos, los válidos por el mayor período de tiempo, etc. Un buen punto de partida puede ser el SA-Index (structure- and attribute-aware index), presentado en [20].

A Nuevos Comandos de TGQL

A continuación, se presentan los cambios y adiciones realizados sobre la implementación anterior de TGQL. Algunas aclaraciones que se evidencian en los ejemplos de cada comando:

- Los nombres de relaciones para los comandos que contienen cláusulas del estilo "ON [relationshipName]" van entre comillas simples (ver secciones siguientes).
- Las fechas se escriben entre comillas simples
- Para la traducción de comandos de creación y conexión de índices es necesaria una conexión a una base de datos, ya que se requiere calcular los IDs máximo y mínimo de los nodos de esta. Esto es debido a que ejecutar una consulta de creación de índices sobre todos los nodos a la vez puede producir errores de agotamiento de memoria del heap. Por esto, se adaptaron estas *procedures* para soportar creación o conexión de índices de a baches (definidos según un rango de IDs) traduciendo una consulta TGQL a una lista de consultas de Cypher. Sin embargo, en los ejemplos, por claridad, se muestra una única consulta (es decir, no se aprovecha la capacidad de dividir las consultas).

A.1 Consulta cPath

Sintaxis

```
cPath(
    (node1Name)-[:relationshipType(*length)?]->(node2Name),
    fromDate,
    toDate
)
```

Ejemplo

```
SELECT p.path[0].attributes.Name as from,
p.path[3].attributes.Name as to,
p.interval as interval
MATCH (n:Person),(n2:Person),
p = cPath((n)-[:Friend*3]->(n2), '2010-03-07', '2014-03-08')
```

Traducción a Cypher

```
MATCH (n:Object {title: 'Person'}),(n2:Object {title: 'Person'})
CALL graphindex.coTemporalPathsExist(
    n,n2,3,3,
    {
        edgesLabel:'Friend',
        between:'2010-03-07-2014-03-08',
        direction:'outgoing'
    })
YIELD value as index_condition
CALL apoc.when(index_condition,
"CALL graphindex.quickRetrievePathsConcat(
    _n,_n2,3,3,
    {
        edgesLabel:'Friend',
        between:'2010-03-07-2014-03-08',
        direction:'outgoing'
    })
    YIELD path,interval RETURN path,interval",
"CALL coexisting.coTemporalPaths(
    _n,_n2,3,3,
    {
        edgesLabel:'Friend',
        between:'2010-03-07-2014-03-08',
        direction:'outgoing'
    })
    YIELD path,interval RETURN path,interval",
{_n:n,_n2:n2}) YIELD value as internal\_v0
WITH {path: internal_v0.path, interval: internal_v0.interval} as p
RETURN p.path[0].attributes.Name as from,
       p.path[3].attributes.Name as to,
       p.interval as `interval`
```

A.2 Creación de un índice "Tree"

Sintaxis

```
CREATE INDEX ON [relationshipName]
BETWEEN [startTime] AND [endTime]
(MAX LENGTH [maxLength])?
FOR TREE INDEX
```

Ejemplo

```
CREATE INDEX ON 'Friend' BETWEEN '2018' AND '2019'
FOR TREE INDEX
```

Traducción a Cypher

```
CALL graphindex.createTreeIndex(
    'Friend', '2018', '2019', 0, 115300
)
```

A.3 Creación de un índice "concat"

Sintaxis

```
CREATE INDEX ON [relationshipName]
BETWEEN [startTime] AND [endTime]
FOR GRAPH INDEX
```

Ejemplo

```
CREATE INDEX ON 'Friend'
BETWEEN '2018' AND 'Now'
FOR GRAPH INDEX
```

Traducción a Cypher

```
CALL graphindex.createIndex('Friend','2018','Now',0, 115300)
```

A.4 Conexión de un índice "concat"

Sintaxis

```
CONNECT INDEX ON STRING BETWEEN [startTime] AND [endTime]
```

Ejemplo

```
CONNECT INDEX ON 'Friend' BETWEEN '2018' AND 'Now'
```

Traducción a Cypher

```
CALL graphindex.connectIndex(
    'Friend','2018','Now',230188, 1230188, 230188, 1230188
)
```

A.5 Creación/Actualización de una nueva arista

Sintaxis

```
CREATE OR UPDATE (obj1Name) - [relName:relType] -> (obj2Name)
FOR (GRAPH|TREE) INDEX
MATCH (obj1Name: obj1Type)
MATCH (obj2Name: obj2Type)
WHERE\_CLAUSE
```

Ejemplo 1 - Índice "concat"

```
CREATE OR UPDATE (o1) -[:Friend] -> (o2)
FOR GRAPH INDEX
MATCH (o1:Person) MATCH (o2:Person)
WHERE o1[id] = 172
```

Traducción a Cypher

```
MATCH (o1:Object {title: 'Person'})
MATCH (o2:Object {title: 'Person'})
WHERE o1.id = 172
CALL graphindex.concatInsertOrUpdateEdge(o1,o2,Friend)
RETURN 'All Done'
```

Ejemplo 2 - Índice "Tree"

```
CREATE OR UPDATE (o1) -[:Friend] -> (o2)
FOR TREE INDEX
MATCH (o1:Person) MATCH (o2:Person)
WHERE o1[id] = 172
```

Traducción a Cypher

```
MATCH (o1:Object {title: 'Person'})
MATCH (o2:Object {title: 'Person'})
WHERE o1.id = 172
CALL graphindex.treeInsertOrUpdateEdge(o1,o2,Friend)
RETURN 'All Done'
```

A.6 Eliminación de una arista

Sintaxis

```
DELETE [relationshipName]
FOR (GRAPH|TREE) INDEX
MATCH (obj1Name: obj1Type) - [relName:relType] -> (obj2Name: obj2Type)
WHERE\_CLAUSE
```

Ejemplo 1 - Índice "concat"

```
DELETE r
FOR GRAPH INDEX
MATCH (o1:Person) -[r:Friend] -> (o2:Person)
WHERE o1[id] = 379
```

Traducción a Cypher

```
MATCH (o1:Object {title: 'Person'}) - [r:Friend] -> (o2:Object {title: 'Person'})
WHERE o1.id = 379
CALL graphindex.concatDeleteEdge(r)
RETURN 'All Done'
```

Ejemplo 2 - Índice "Tree"

```
DELETE r
FOR TREE INDEX
MATCH (o1:Person) -[r:Friend] -> (o2:Person)
WHERE o1[id] = 379
```

Traducción a Cypher

```
MATCH (o1:Object {title: 'Person'}) - [r:Friend] -> (o2:Object {title: 'Person'})
WHERE o1.id = 379
CALL graphindex.treeDeleteEdge(r)
RETURN 'All Done'
```

A.7 Eliminación de un índice

Sintaxis

```
DELETE (TREE | GRAPH)? INDEX
(ON [relName])?
(BETWEEN [startTime] AND [endTime])?
```

Ejemplo 1 - Borrar cualquier índice

```
DELETE INDEX
```

Traducción a Cypher

```
CALL graphindex.deleteAllIndices();
```

Ejemplo 2 - Borrar índice "concat"

```
DELETE GRAPH INDEX ON 'Friend'
```

Traducción a Cypher

```
CALL graphindex.deleteGraphIndex('Friend');
```

Ejemplo 3 - Borrar índice "Tree"

```
DELETE GRAPH INDEX ON 'Friend' BETWEEN '2018' AND '2019'
```

Traducción a Cypher

```
CALL graphindex.deleteTreeIndex('Friend', '2018', '2019');
```

B Granularidades

Existen cuatro tipos de granularidades para los intervalos en TGQL (y asimismo soportados por las *procedures* implementadas en Java). Estos se encuentran en la Tabla 11.

Nombre	Formato	Ejemplo
Year	YYYY	'2018'
Year-Month	YYYY-MM	'2018-03'
Date	YYYY-MM-DD	'2018-03-01'
DateTime	YYYY-MM-DD HH:MM	'2018-03-01 15:15'

Table 11. Granularidades

Además de soportar estos formatos, también se puede representar que algo es actual a partir del valor 'Now', que, intuitivamente, sólo puede ser utilizado en el extremo final de un intervalo.

C *Procedures* de Neo4j

C.1 Funciones que no requieren un índice

C.1.1 coexisting.coTemporalPaths

Es la función original para el cálculo de caminos continuos.

Parámetros

- **startNode:** Nodo de inicio del camino
- **endNode:** Nodo de finalización del camino. Posiblemente null.
- **minLength:** Longitud mínima del camino.
- **maxLength:** Longitud máxima del camino.
- **configuration:** Un mapa con los siguientes parámetros obligatorios:
  - **edgesLabel:** Nombre de la relación del camino.
  - **between:** Intervalo de búsqueda con formato 'startTime—endTime'.
  - **direction:** Dirección del camino. En el presente trabajo se considera únicamente el valor 'outgoing'.

Ejemplo de Invocación

```
MATCH (n1:Object) MATCH (n2:Object)
WHERE n1.id = 2074 and n2.id = 2056
CALL coexisting.coTemporalPaths(
    n, n2, 3, 3, {
        edgesLabel:'Friend',
        between:'2010-03-07|2014-03-08',
        direction:'outgoing'
    })
```

C.1.2 coexisting.coTemporalPaths.exists

Permite determinar si existen caminos continuos entre dos nodos.

Parámetros

- **startNode:** Nodo de inicio del camino
- **endNode:** Nodo de finalización del camino. Posiblemente null.
- **minLength:** Longitud mínima del camino.
- **maxLength:** Longitud máxima del camino.
- **configuration:** Un mapa con los siguientes parámetros obligatorios:
  - **edgesLabel:** Nombre de la relación del camino.
  - **between:** Intervalo de búsqueda con formato 'startTime—endTime'.
  - **direction:** Dirección del camino. En el presente trabajo se considera únicamente el valor 'outgoing'.

Ejemplo de Invocación

```
MATCH (n1:Object) MATCH (n2:Object)
WHERE n1.id = 2074 and n2.id = 2056
CALL coexisting.coTemporalPaths.exists(
    n, n2, 3, 3, {
        edgesLabel:'Friend',
        between:'2010-03-07|2014-03-08',
        direction:'outgoing'
    })
```

C.1.3 coexisting.quickCoTemporalPaths

Es la implementación de quickCPaths.

Parámetros

- **startNode:** Nodo de inicio del camino
- **endNode:** Nodo de finalización del camino. Posiblemente null.
- **minLength:** Longitud mínima del camino.
- **maxLength:** Longitud máxima del camino.
- **configuration:** Un mapa con los siguientes parámetros obligatorios:
  - **edgesLabel:** Nombre de la relación del camino.
  - **between:** Intervalo de búsqueda con formato 'startTime—endTime'.
  - **direction:** Dirección del camino. En el presente trabajo se considera únicamente el valor 'outgoing'.

Ejemplo de Invocación

```
MATCH (n1:Object) MATCH (n2:Object)
WHERE n1.id = 2074 and n2.id = 2056
CALL coexisting.quickCoTemporalPaths.exists(
    n, n2, 3, 3, {
        edgesLabel:'Friend',
        between:'2010-03-07|2014-03-08',
        direction:'outgoing'
    })
```

C.2 Funciones para índices

C.2.1 graphindex.coTemporalPathsExist

Determina si existe un índice para los caminos buscados a partir de la existencia de un nodo Meta correspondiente.

Parámetros

- **startNode:** Nodo de inicio del camino
- **endNode:** Nodo de finalización del camino. Posiblemente null.
- **minLength:** Longitud mínima del camino.
- **maxLength:** Longitud máxima del camino.
- **fromDate:** Fecha de inicio del intervalo de búsqueda.
- **toDate:** Fecha de finalización del intervalo de búsqueda.
- **configuration:** Un mapa con los siguientes parámetros obligatorios:
  - **edgesLabel:** Nombre de la relación del camino.
  - **direction:** Dirección del camino. En el presente trabajo se considera únicamente el valor 'outgoing'.

Ejemplo de Invocación

```
MATCH (n1:Object) MATCH (n2:Object)
WHERE n1.id = 2074 and n2.id = 2056
CALL graphindex.coTemporalPathsExist(
    n, n2, 3, 3, '2010-03-07', '2014-03-08', {
        edgesLabel:'Friend',
        direction:'outgoing'
    })
```

C.2.2 graphindex.concatDeleteEdge

Permite eliminar una arista, actualizando un índice "concat".

Parámetros

- **edge:** Arista a eliminar.

Ejemplo de Invocación

```
MATCH (u1:Object) -[r:Friend]-> (u2:Object)
WHERE u1.id = 40 and u2.id = 43
CALL graphindex.concatDeleteEdge(r)
RETURN "All done"
```

C.2.3 graphindex.concatInsertOrUpdateEdge

Permite crear o actualizar una arista, actualizando un índice "concat".

Parámetros

- **startNode:** Nodo inicial de la arista.
- **endNode:** Nodo final de la arista.
- **relationshipType:** Tipo de relación de la arista.

Ejemplo de Invocación

```
MATCH (u1:Object) match (u2:Object)
WHERE u1.id = 4132 and u2.id = 4126
CALL graphindex.concatInsertOrUpdateEdge(u1, u2, 'Friend')
RETURN 'done'
```

C.2.4 graphindex.connectIndex

Crea una arista entre dos nodos u y v tales que

- u.Destination = v.Source
- u.interval $\cap$ v.interval $\neq \emptyset$

Parámetros

- **relationshipType:** Tipo de la relación indexada.
- **startTime:** Momento de inicio del intervalo de indexación.
- **endTime:** Momento de finalización del intervalo de indexación.
- **r1LowId:** Id nativo más bajo de entre los nodos a conectar a tener en cuenta como extremos iniciales de las relaciones a crear.
- **r1HighId:** Id nativo más alto de entre los nodos a conectar a tener en cuenta como extremos iniciales de las relaciones a crear.
- **r2LowId:** Id nativo más bajo de entre los nodos a conectar a tener en cuenta como extremos finales de las relaciones a crear.
- **r2HighId:** Id nativo más alto de entre los nodos a conectar a tener en cuenta como extremos finales de las relaciones a crear.

Ejemplo de Invocación

```
CALL graphindex.connectIndex(
    'Friend','2018','Now',230188, 1230188, 230188, 1230188
)
```

C.2.5 graphindex.createIndex

Crea un índice "concat".

Parámetros

- **relationshipType:** Tipo de la relación indexada.
- **startTime:** Momento de inicio del intervalo de indexación.
- **endTime:** Momento de finalización del intervalo de indexación.
- **startId:** Id nativo más bajo de entre los nodos Object que se tienen como fuente para el cálculo de caminos.
- **endId:** Id nativo más alto de entre los nodos Object que se tienen como fuente para el cálculo de caminos.

Ejemplo de Invocación

```
CALL graphindex.createIndex('Friend','2018','Now',0, 115300)
```

C.2.6 graphindex.createOriginalIndex

Crea un índice "Tree" permitiendo definir mínima y máxima longitud de caminos.

Parámetros

- **relationshipType:** Tipo de la relación indexada.
- **minLength:** Longitud mínima de los caminos a indexar.
- **maxLength:** Longitud máxima de los caminos a indexar.
- **startTime:** Momento de inicio del intervalo de indexación.
- **endTime:** Momento de finalización del intervalo de indexación.
- **startId:** Id nativo más bajo de entre los nodos Object que se tienen como fuente para el cálculo de caminos.
- **endId:** Id nativo más alto de entre los nodos Object que se tienen como fuente para el cálculo de caminos.

Ejemplo de Invocación

```
CALL graphindex.createOriginalIndex('Friend', 2, 3, '2018','Now',0, 115300)
```

C.2.7 graphindex.createTreeIndex

Crea un índice "Tree" para caminos de longitud 2.

Parámetros

- **relationshipType:** Tipo de la relación indexada.
- **maxLength:** Longitud máxima de los caminos a indexar.
- **startTime:** Momento de inicio del intervalo de indexación.
- **endTime:** Momento de finalización del intervalo de indexación.
- **startId:** Id nativo más bajo de entre los nodos Object que se tienen como fuente para el cálculo de caminos.
- **endId:** Id nativo más alto de entre los nodos Object que se tienen como fuente para el cálculo de caminos.

Ejemplo de Invocación

```
CALL graphindex.createOriginalIndex('Friend', 3, '2018','Now',0, 115300)
```

C.2.8 graphindex.deleteAllIndices

Elimina completamente un índice.

Parámetros

Esta función no tiene parámetros.

Ejemplo de Invocación

```
CALL graphindex.deleteAllIndices()
```

C.2.9 graphindex.deleteGraphIndex

Elimina un nodo "concat" para una relación.

Parámetros

- **relationshipType:** Tipo de la relación indexada.

Ejemplo de Invocación

```
CALL graphindex.deleteGraphIndex('Friend')
```

C.2.10 graphindex.deleteTreeIndex

Permite eliminar un nodo Meta con cierto intervalo de validez y sus nodos Index relacionados (por medio de aristas :type).

Parámetros

- **relationshipType:** Tipo de la relación indexada.
- **fromDate:** Fecha de inicio del intervalo de búsqueda.
- **toDate:** Fecha de finalización del intervalo de búsqueda.

Ejemplo de Invocación

```
CALL graphindex.deleteTreeIndex('Friend', '2018', '2019')
```

C.2.11 graphindex.quickRetrievePathsConcat

Implementación de quickConcat. Sólo puede ser utilizada con índices "concat".

Parámetros

- **startNode:** Nodo de inicio del camino
- **endNode:** Nodo de finalización del camino. Posiblemente null.
- **minLength:** Longitud mínima del camino.
- **maxLength:** Longitud máxima del camino.
- **configuration:** Un mapa con los siguientes parámetros obligatorios:
  - **edgesLabel:** Nombre de la relación del camino.
  - **between:** Intervalo de búsqueda con formato 'startTime—endTime'.
  - **direction:** Dirección del camino. En el presente trabajo se considera únicamente el valor 'outgoing'.

Ejemplo de Invocación

```
MATCH (n1:Object) MATCH (n2:Object)
WHERE n1.id = 2074 and n2.id = 2056
CALL graphindex.quickRetrievePathsConcat(
    n, n2, 3, 3, {
        edgesLabel:'Friend',
        between:'2010-03-07|2014-03-08',
        direction:'outgoing'
    })
```

C.2.12 [Deprecada] graphindex.retrievePaths

Implementación de retrievePaths. Sólo puede ser utilizada con índices "Tree".

Parámetros

- **startNode:** Nodo de inicio del camino
- **endNode:** Nodo de finalización del camino. Posiblemente null.
- **minLength:** Longitud mínima del camino.
- **maxLength:** Longitud máxima del camino.
- **fromDate:** Fecha de inicio del intervalo de búsqueda.
- **toDate:** Fecha de finalización del intervalo de búsqueda.
- **configuration:** Un mapa con los siguientes parámetros obligatorios:
  - **edgesLabel:** Nombre de la relación del camino.
  - **direction:** Dirección del camino. En el presente trabajo se considera únicamente el valor 'outgoing'.

Ejemplo de Invocación

```
MATCH (u1:Object) MATCH (u2:Object)
WHERE u1.id = 2074 and u2.id = 2056
CALL graphindex.retrievePaths(
    u1, u2, 2, 3, '2010', '2015',
    {
        edgesLabel:'Friend',
        direction:'outgoing'
    })
YIELD path, interval
RETURN path, interval
```

C.2.13 graphindex.retrievePathsBFSConcat

Implementación de BFSConcat. Sólo puede ser utilizada con índices "concat".

Parámetros

- **startNode:** Nodo de inicio del camino
- **endNode:** Nodo de finalización del camino. Posiblemente null.
- **minLength:** Longitud mínima del camino.
- **maxLength:** Longitud máxima del camino.
- **configuration:** Un mapa con los siguientes parámetros obligatorios:
  - **edgesLabel:** Nombre de la relación del camino.
  - **between:** Intervalo de búsqueda con formato 'startTime—endTime'.
  - **direction:** Dirección del camino. En el presente trabajo se considera únicamente el valor 'outgoing'.

Ejemplo de Invocación

```
MATCH (n1:Object) MATCH (n2:Object)
WHERE n1.id = 2074 and n2.id = 2056
CALL graphindex.retrievePathsBFSConcat(
    n, n2, 3, 3, {
        edgesLabel:'Friend',
        between:'2010-03-07|2014-03-08',
        direction:'outgoing'
    })
```

C.2.14 graphindex.retrievePathsConcat

Implementación de retrievePathsConcat (source-destination). Sólo puede ser utilizada con índices "Tree".

Parámetros

- **startNode:** Nodo de inicio del camino
- **endNode:** Nodo de finalización del camino. Posiblemente null.
- **minLength:** Longitud mínima del camino.
- **maxLength:** Longitud máxima del camino.
- **fromDate:** Fecha de inicio del intervalo de búsqueda.
- **toDate:** Fecha de finalización del intervalo de búsqueda.
- **configuration:** Un mapa con los siguientes parámetros obligatorios:
  - **edgesLabel:** Nombre de la relación del camino.
  - **direction:** Dirección del camino. En el presente trabajo se considera únicamente el valor 'outgoing'.

Ejemplo de Invocación

```
MATCH (u1:Object) MATCH (u2:Object)
WHERE u1.id = 2074 and u2.id = 2056
CALL graphindex.retrievePathsConcat(
    u1, u2, 7, 7, '2010', '2015',
    {
        edgesLabel:'Friend',
        direction:'outgoing'
    })
YIELD path, interval
RETURN path, interval
```

C.2.15 graphindex.sourceOnlyRetrievePathsConcat

Implementación de retrievePathsConcat (source-destination). Sólo puede ser utilizada con índices "Tree".

Parámetros

- **startNode:** Nodo de inicio del camino
- **minLength:** Longitud mínima del camino.
- **maxLength:** Longitud máxima del camino.
- **fromDate:** Fecha de inicio del intervalo de búsqueda.
- **toDate:** Fecha de finalización del intervalo de búsqueda.
- **configuration:** Un mapa con los siguientes parámetros obligatorios:
  - **edgesLabel:** Nombre de la relación del camino.
  - **direction:** Dirección del camino. En el presente trabajo se considera únicamente el valor 'outgoing'.

Ejemplo de Invocación

```
MATCH (u1:Object)
WHERE u1.id = 2074
CALL graphindex.sourceOnlyRetrievePathsConcat(
    u1, 7, 7, '2010', '2015',
    {
        edgesLabel:'Friend',
        direction:'outgoing'
    })
YIELD path, interval
RETURN path, interval
```

C.2.16 graphindex.treeDeleteEdge

Permite eliminar una arista, actualizando un índice "Tree".

Parámetros

- **edge:** Arista a eliminar.

Ejemplo de Invocación

```
MATCH (u1:Object) -[r:Friend]-> (u2:Object)
WHERE u1.id = 40 and u2.id = 43
CALL graphindex.treeDeleteEdge(r)
RETURN "All done"
```

C.2.17 graphindex.treeInsertOrUpdateEdge

Permite crear o actualizar una arista, actualizando un índice "concat".

Parámetros

- **startNode:** Nodo inicial de la arista.
- **endNode:** Nodo final de la arista.
- **relationshipType:** Tipo de relación de la arista.

Ejemplo de Invocación

```
MATCH (u1:Object) match (u2:Object)
WHERE u1.id = 4132 and u2.id = 4126
CALL graphindex.treeInsertOrUpdateEdge(u1, u2, 'Friend')
RETURN 'done'
```

D Algoritmos Auxiliares

Algorithm 12 compute2Path: Computes a list of node ids from a list of length-2 index nodes

Input: A list of length-2 index nodes L_{in}
Output: A list of node ids L_{out}

Let $L_{out} \coloneqq [L_{in_1}.Source]$
for each $N_{index} \in\in L_{in}$ **do**
  $L_{out}.extend([N_{index}.Intermediate[1], N_{index}.Destination])$
end for
return L_{out}

Algorithm 13 compute2PathWithFinalRelationship: Computes a list of node ids from a list of length-2 index nodes and a final relationship

Input: A list of length-2 index nodes L_{in}, a relationship R
Output: A list of node ids L_{out}

Let $L_{out} \coloneqq [L_{in_1}.Source]$
for each $N_{index} \in L_{in}$ **do**
  $L_{out}.extend([N_{index}.Intermediate[1], N_{index}.Destination])$
end for
$L_{out}.insert(endNode(R).id)$
return L_{out}

Algorithm 14 graphIndexPathToList: Computes a list of node ids from a list of length-2 index nodes

Input: A path of :concat relationships P
Output: A list of node ids L_{out}

Let $L_{out} \coloneqq [P_1.Source]$
for each $N_{index} \in nodes(P)$ **do**
  $L_{out}.extend([N_{index}.Intermediate[1], N_{index}.Destination])$
end for
return L_{out}

Algorithm 15 graphIndexPathWithFinalRelationshipToList: Computes a list of node ids from a list of length-2 index nodes and a final relationship

Input: A path of :concat relationships P, a relationship R
Output: A list of node ids L_{out}

Let $L_{out} \coloneqq [P_1.Source]$
for each $N_{index} \in nodes(P)$ **do**
  $L_{out}.extend([N_{index}.Intermediate[1], N_{index}.Destination])$
end for
$L_{out}.insert(endNode(R).id)$
return L_{out}

References

1. DB-Engines Ranking. https://db-engines.com/en/ranking/graph+dbms, 2021.
2. J. Byun. Enabling time-centric computation for efficient temporal graph traversals from multiple sources. *IEEE Transactions on Knowledge and Data Engineering*, 2020.
3. J. Byun, S. Woo, and D. Kim. ChronoGraph: Enabling Temporal Graph Traversals for Efficient Information Diffusion Analysis over Time. *IEEE Transactions on Knowledge and Data Engineering*, 32:424–437, 2020.
4. A. Campos, J. Mozzino, and A. Vaisman. Towards Temporal Graph Databases, 2016.
5. A. Debrouvier, E. Parodi, M. Perazzo, V. Soliani, and A. Vaisman. A Model and Query Language for Temporal Graph Databases. *VLDB*, 2020.
6. R. Elmasri. The Time Index: An Access Structure for Temporal Data. *VLDB*, 1991.
7. R. Elmasri, Y. J. Kim, and G. Wuu. Efficient implementation techniques for the time index. *[1991] Proceedings. Seventh International Conference on Data Engineering.*, 1991.
8. K. D. Foote. Graph Databases: Updates on Their Growing Popularity. https://www.dataversity.net/graph-databases-updates-on-their-growing-popularity/, 2021.
9. A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Schuster, P. Selmer, and H. Voigt. Updating graph databases with Cypher. *Proceedings of the VLDB Endowment*, 12:2242—-2254, 2019.
10. S. Huang, J. Cheng, and H. Wu. Temporal Graph Traversals: Definitions, Algorithms, and Applications, 2014.
11. K. Kusu and K. Hatano. Recurrent Path Index for Efficient Graph Traversal. *2019 IEEE International Conference on Big Data (Big Data)*, 2019.
12. B. Liu and B. Hu. Path Queries Based RDF Index. *2005 First International Conference on Semantics, Knowledge and Grid*, 2005.
13. D. Lomet and B. Salzberg. The performance of a multiversion access method. *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data - SIGMOD '90*, 1990.
14. J. Pokorný, M. Valenta, and M. Troup. Graph Pattern Index for Neo4j Graph Databases. *Communications in Computer and Information Science*, page 69–90, 2019.
15. F. Rizzolo and A. A. Vaisman. Temporal XML: modeling, indexing, and query processing. *The VLDB Journal*, 17:1179–1212, 2007.
16. S. Sakr and G. Al-Naymat. Graph indexing and querying: a review. *International Journal of Web Information Systems*, 6:101–120, 2010.
17. S. Srinivasa. Data, Storage and Index Models for Graph Databases. *Advances in Data Mining and Database Management*, page 47–70, 2012.
18. F. Wei-Kleiner. Tree decomposition-based indexing for efficient shortest path and nearest neighbors query answering on graphs. *Journal of Computer and System Sciences*, 82:23–44, 2016.
19. L. Yan, P. Zhao, and Z. Ma. Indexing temporal RDF graph. *Computing*, 101:1457–1488, 2019.
20. L. Zhu, W. Keong Ng, and J. Cheng. Structure and attribute index for approximate graph matching in large graphs. *Information Systems*, 36:958–972, 2011.