Universidad de la República

Facultad de Ingeniería

Migración automática de bases de datos relacionales a bases de datos de grafos

Informe de Proyecto de Grado presentado por

Augusto Alonso y Elizabeth Lofredo

como requisito de graduación de la carrera de Ingeniería en Computación
de Facultad de Ingeniería de la Universidad de la República

Supervisor

Lorena Etcheverry

Montevideo, 24 de abril de 2023

II

Resumen

En los últimos años el uso de bases de datos distintas a bases de datos relacionales, como por ejemplo las bases de datos de grafos, ha aumentado y ganado popularidad. Un desafío claro que se presenta ante esta tendencia, es la migración de bases de datos relacionales ya existentes a estos nuevos formatos.

Este trabajo pone foco en una de estas migraciones: las migraciones hacia bases de datos de grafos. Su principal propósito es implementar una herramienta que transforme de manera automática, con la menor intervención manual posible, cualquier base de datos PostgreSQL a una base de datos ArangoDB.

El trabajo se dividió en tres objetivos. El primero consistió en realizar un relevamiento de documentación sobre metodologías y herramientas existentes que realicen transformaciones de bases de datos relacionales a bases de datos de grafos, independientemente del motor que utilicen. Luego, se realizó una investigación más minuciosa al material encontrado, así como también la aplicación de pruebas para verificar el correcto funcionamiento de estos métodos y herramientas. Este paso fue de particular importancia ya que permitió sentar las bases sobre las cuales se construyó la herramienta. Finalmente, el último objetivo consistió en implementar la herramienta teniendo en cuenta los hallazgos de los objetivos anteriores. Esto comprendió la investigación de una arquitectura apropiada, el diseño del algoritmo que realice la transformación automática y su implementación, el diseño de la interfaz gráfica y la construcción de pruebas con las cuales se evaluó su desempeño frente a las otras herramientas y métodos encontrados.

Palabras clave: Bases de datos relacionales, Bases de datos de grafos, Manejadores de bases de datos, PostgreSQL, ArangoDB, Neo4j, Direct Mapping, Electron.js, React.js, Node.js, SQL, AQL

Tabla de contenidos

Capítulo 1

Introducción

1.1. Contexto y motivación

Las bases de datos relacionales han sido el estándar para el almacenamiento de la información digital durante muchos años. Su estandarización, velocidad y fácil manipulación de los datos, entre otros factores, han hecho que sean adoptadas en la mayoría de los sistemas de información existentes hoy en día.

Sin embargo, en los últimos años, otros tipos de bases de datos, cómo las bases de datos documentales, bases de datos clave-valor, o bases de datos de grafos; han empezado a ganar popularidad entre los usuarios debido a distintas ventajas que presentan frente a las bases de datos relacionales. En particular, las bases de datos de grafos han ganado tracción en realidades donde las relaciones son muy importantes, cómo por ejemplo, redes sociales o sistemas de prevención de fraude en entidades financieras[1].

Estos tipos de bases de datos, dieron lugar al concepto de persistencia políglota; una estrategia de diseño de sistemas de almacenamiento de datos que consiste en utilizar diferentes tecnologías de bases de datos para almacenar diferentes tipos de datos. (ArangoDB, 2020)

Debido a la complejidad que requiere mantener distintas bases de datos de distintos tipos que son utilizadas por un mismo sistema, surge también el concepto de base de datos multi-modelo, el cual permite tener esta persistencia políglota en una sola base de datos. Una de ellas es ArangoDB, una base de datos multi-modelo nativa de código abierto que soporta pares

[1] Graph DBMS increased their popularity by 500 % within the last 2 years - https://neo4j.com/news/graph-dbms-increased-their-popularity-by-500percent (Último acceso: 4 de Marzo, 2023)

clave/valor, grafos y/o documentos.

Resulta interesante la idea de transformar un sistema de persistencia políglota que utilice distintos tipos de bases de datos a una base de datos ArangoDB. Esto contemplaría la transformación de distintos tipos de bases de datos a bases de datos ArangoDB. Un acercamiento posible consiste en construir una herramienta que realice la transformación automática de bases de datos relacionales a la representación de grafos en ArangoDB.

1.2. Objetivos

Este trabajo tendrá tres objetivos. El primero consta de investigar el estado del arte en cuanto a las transformaciones de bases de datos relacionales a bases de datos de grafos. Esto abarca la investigación de distintos métodos para realizar transformaciones, así como también de las herramientas disponibles al momento de realizar este trabajo.

El segundo objetivo será poner a prueba los métodos y herramientas encontrados durante la investigación. Los mismos serán evaluados en diferentes realidades para analizar su comportamiento.

Finalmente, el tercer objetivo de este trabajo será construir una herramienta que realice la transformación de una base de datos relacional en PostgreSQL, uno de los manejadores de bases de datos relacionales más populares en la actualidad, a una base de datos de grafos compatible con la base de datos multi-modelo ArangoDB. Para esto, además de la implementación de dicha herramienta, se realizarán pruebas tanto de completitud como de corrección para validar el resultado final.

1.3. Resultados

Como resultado de este trabajo se obtiene el relevamiento de las propuestas y herramientas existentes a la fecha enfocadas en la transformación de bases de datos relacionales a bases de datos de grafos. También la evaluación de estas propuestas detallando sus características, funcionamiento y desempeño bajo diferentes realidades.
Finalmente se presenta el prototipo de una herramienta que brinda una solución para la transformación automática de bases de datos relacionales en PostgreSQL a una base de datos de grafos ArangoDB.

1.4. Estructura del documento

Este trabajo de proyecto de grado se divide en seis capítulos principales, el contenido que se encuentra en cada uno de ellos es detallado a continua-

ción.

El capítulo 2 presenta una visión general de los conceptos fundamentales necesarios para el entendimiento de el proyecto, como lo son conceptos relacionados a bases de datos relacionales y bases de datos de grafos.

En el capítulo 3 se presentan trabajos previos relacionados al objetivo de este proyecto. Para dichos trabajos, se analizan los métodos utilizados, sus implementaciones en caso de encontrarse disponibles y se ejecutan y evalúan realizando un análisis de los mismos. Finalmente se presentan las conclusiones extraídas de estas evaluaciones.

En el capítulo 4 se presenta la solución propuesta, desde su diseño hasta su implementación. Se mencionan las decisiones tomadas durante la creación de la misma y se describe su lógica principal explicando el paso a paso de la transformación. También se detalla su estructura y las tecnologías utilizadas, comenzando por la biblioteca que contiene la lógica principal de las transformaciones continuando por la estructura de la aplicación final presentada al usuario.

En el capítulo 5 se describen las pruebas realizadas para evaluar la completitud y corrección de la solución propuesta en el capítulo anterior.

Finalmente, en el capítulo 6, se presentan las conclusiones extraídas de este trabajo de proyecto de grado así como con las posibles líneas de trabajo futuro que se pueden seguir a partir de él mismo.

Capítulo 2

Conceptos preliminares

En éste capítulo se presentan distintos conceptos que serán útiles para poder entender en mayor profundidad los contenidos de las siguientes secciones. Éstos conceptos están directamente ligados a las dos protagonistas principales de este trabajo; las bases de datos de grafos y las bases de datos relacionales.

Para ayudar a entender mejor las definiciones, se presenta en la Figura 2.1 un pequeño modelo entidad-relación de ejemplo. En esta realidad existe la entidad *persona* que se identifica por su nombre, por lo que no pueden existir dos personas con un mismo nombre. Las personas a su vez, conocen a otras personas, y esto se representa mediante la relación *conoce*. A su vez existe la entidad *lugar*, que también está identificada por su nombre, y los lugares pueden ser visitados por personas, lo cuál se representa mediante la relación *visita*.

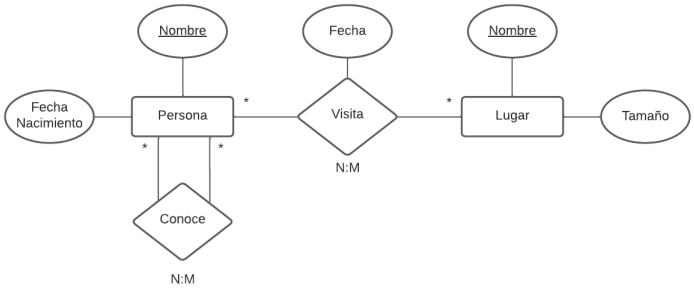

Figura 2.1: MER (Modelo entidad-relación) de ejemplo.

2.1. Bases de datos relacionales

Los conceptos relacionados a bases de datos relacionales introducidos a continuación fueron extraídos de (Elmasri y Navathe, 2010).

Uno de los principales conceptos dentro del mundo de las bases de datos relacionales es el concepto de **esquema relacional**. Se define un esquema relacional R, denotado como *R(A_1, A_2, ..., A_n)*, al par $< R, A >$, donde R es un nombre de relación, y A es un conjunto de atributos A_1, A_2, ..., A_n. En la realidad planteada anteriormente, un esquema relacional, por ejemplo, es la entidad $persona$ cuyos atributos son $nombre$ y $fecha\ de\ nacimiento$.

Las bases de datos relacionales, generalmente se componen por un conjunto de esquemas relacionales denominado **esquema de base de datos relacional**. Sea un conjunto de esquemas relacionales *S* = R_1, R_2, ..., R_m y un **conjunto de restricciones de integridad** IC definido sobre S, se entiende como **esquema de base de datos relacional** al par *<S, IC>*. En el ejemplo propuesto, los esquemas relacionales que existen son $persona$, $lugar$, $conoce$ y $visita$.

Siguiendo con el ejemplo, la entidad $persona$ solo posee los atributos $nombre$ y $fecha\ de\ nacimiento$ y puede surgir la pregunta: *¿cómo identificar a una persona si pueden existir dos personas con el mismo nombre y la misma fecha de nacimiento?*. La respuesta a esta pregunta es mediante el uso de una **clave primaria**.

Antes de definir una clave primaria, es necesario saber que es una clave en el mundo de las bases de datos relacionales. Sea *R(A_1, A_2, ..., A_n)* un esquema relacional, una **superclave** SK es un subconjunto de atributos $A_i, ..., A_j$ tal que para cada par de tuplas t_1 y $t_2 \in R$, $t_1[SK] \neq t_2[SK]$. Luego, una **clave** es también una superclave con la particularidad de que si uno de sus atributos es removido del conjunto de atributos, el mismo deja de ser una superclave. Finalmente, una **clave primaria** es una clave elegida especialmente para identificar una tupla dentro de una relación. Es importante destacar que si una clave tiene múltiples atributos se denomina **clave compuesta**.
En el ejemplo presentado, la clave primaria está compuesta solamente por el atributo $nombre$, por lo tanto no pueden existir dos personas con el mismo nombre en esa realidad. Para solucionar esto se puede introducir un nuevo atributo único que identifique a las personas, cómo por ejemplo su cédula de identidad.

Como se aprecia en la realidad planteada, una persona puede conocer a una o más personas, y también puede visitar uno o más lugares en distintas fechas. Para representar esto se utiliza el concepto de clave foránea. Una **clave foránea** especifica una restricción de integridad referencial en-

tre dos esquemas de relación R_1 y R_2. Un conjunto de atributos FK en un esquema de relación R_1 es una clave foránea de R_1 que referencia a la relación R_2 si satisface una de las siguientes propiedades: 1. Los atributos en *FK* referencian a la relación R_2 2. Un valor de FK en una tupla t_1 del estado actual $r_1(R_1)$ ocurre como valor de una clave primaria PK para una tupla t_2 en el estado actual $r_2(R_2)$, o dicho valor es nulo. En el primer caso se tiene que $t_1[FK] = t_2[PK]$ y se dice que la tupla t_1 referencia a la tupla t_2.

Una base de datos puede ser tanto leída como escrita, y al ser escrita, el estado de la misma cambia, por lo tanto se dice que una base de datos tiene varios estados a lo largo de su existencia. Se entiende como estado de una base de datos relacional E_S al conjunto de estados relacionales $DB = r_1, r_2, ..., r_m$ donde cada r_i es un estado de un esquema relacional R_i y satisface las restricciones de integridad de la base de datos.

Para cerrar los conceptos preliminares relacionados a las bases de datos relacionales, se debe presentar el concepto más importante de todos; el de base de datos relacional. Una **base de datos relacional** es la combinación del esquema relacional S y estado relacional actual E_S.

2.2. Bases de datos de grafos

Recientemente las bases de datos de grafos han ganado popularidad en la comunidad del software. Un grafo es básicamente un conjunto de nodos y aristas que conectan a estos nodos. Dentro de las bases de datos de grafos podemos encontrar dos tipos; aquellas basadas en el estándar *Resource Description Framework* (RDF por sus siglas en inglés)[1] y aquellas basadas en los *property graphs*. Este trabajo se centrará en la segunda.

Un **property graph** es un conjunto de nodos que contienen propiedades representadas como *<atributo, valor>* y un conjunto de aristas que conectan dichos nodos estableciendo una relación entre ellos. Estas aristas pueden tener etiquetas para representar mejor la relación entre nodos, y también pueden contener propiedades representadas como *<atributo, valor>*. En el contexto de la realidad planteada, una instancia de un property graph luciría como la Figura 2.2

En este caso, las entidades $persona$ y $lugar$ están representadas con nodos, y estos nodos contienen las mismas propiedades que las tablas en la base de datos relacional. Por otro lado, las entidades $conoce$ y $visita$ son representadas como aristas, y los atributos de relación son agregados también en dicha arista.

[1] https://www.w3.org/RDF/

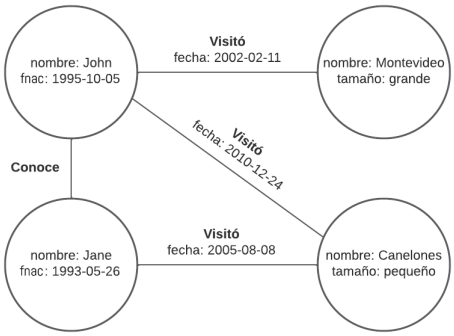

Figura 2.2: Ejemplo de una instancia de un property graph

A continuación se presenta una definición más formal sobre los *property graphs* extraída de (Angles, 2018). Sea L un conjunto infinito de etiquetas (para nodos y aristas), P un conjunto infinito de nombres de propiedades, V un conjunto infinito de valores atómicos, y T un conjunto finito de tipos de datos (por ejemplo, *entero*). Dado un conjunto X, se asume que el conjunto $SET^{+}(X)$ es el conjunto de todos los subconjuntos de X, excluyendo al conjunto vacío. Dado un valor $v \in V$, la función $type(v)$ retorna el tipo de datos de v. Un property graph es una tupla $G = (N, E, \rho, \lambda, \sigma)$ donde:

1. N es un conjunto finito de nodos (también llamados vértices)
2. E es un conjunto finito de aristas tal que E no tiene elementos en común con N
3. $\rho : E \to (N \times N)$ es una función total que asocia a toda arista $e \in E$ con un par de nodos $n_1, n_2 \in N$
4. $\lambda : (N \cup E) \to SET^{+}(L)$ es una función parcial que asocia nodos y aristas con un conjunto de etiquetas de L
5. $\sigma : (N \cup E) \times P \to SET^{+}(V)$ es una función parcial que asocia nodos y aristas con propiedades, y para cada propiedad se le asigna un conjunto de valores de V

2.3. Manejadores de bases de datos

Por último, para poder interactuar con una base de datos, se utilizan los sistemas manejadores de bases de datos (DBMS por sus siglas en inglés). Un sistema manejador de bases de datos es una colección de programas que permiten la definición, construcción y manipulación de bases de datos. (Elmasri y Navathe, 2010).

Existen una gran variedad de DBMS, tanto para las bases de datos relacionales (RDBMS) como para las bases de datos de grafos (GDBMS). Sin embargo, este trabajo pondrá foco en los siguientes manejadores: PostgreSQL[2], uno de los manejadores de bases de datos relacional más populares; Neo4j[3] el manejador más usado en el mundo de las bases de datos de grafos según Database Trends and Applications[4]; y ArangoDB[5], un manejador de bases multi-modelo de código abierto.

ArangoDB es considerado un manejador multi-modelo ya que soporta múltiples tipos de bases de datos, como bases de datos documentales, de grafos y de clave valor. Fue lanzado en 2012 bajo el nombre de AvocadoDB y luego fue renombrado a ArangoDB[6]. El mismo posee una gran cantidad de funcionalidades disponibles y tiene un rendimiento muy competitivo frente a otros manejadores de bases de datos que tienen mayor popularidad como Neo4J y OrientDB. [7]. Además, posee su propio lenguaje para consultas denominado AQL por su nombre en inglés ArangoDB Query Language.

[2] https://www.postgresql.org/

[3] https://neo4j.com/

[4] DBTA Readers' Choice Awards Winners (2022) - https://www.dbta.com/Editorial/Trends-and-Applications/DBTA-Readers-Choice-Awards-Winners-2022-154324.aspx?PageNum=4 (Último acceso: 4 de Marzo, 2023)

[5] https://www.arangodb.com/

[6] https://dbdb.io/db/arangodb (Último acceso: 4 de Marzo, 2023)

[7] https://www.arangodb.com/wp-content/uploads/2018/03/ArangoDB_vs_Neo4j_2018-1.pdf/ (Último acceso: 4 de Marzo, 2023)

Capítulo 3

Trabajos relacionados

A lo largo de este capítulo serán presentados trabajos relacionados con el objetivo de este proyecto: la transformación de bases de datos relacionales a bases de datos basadas en grafos. De la revisión de la literatura surgen dos propuestas, por un lado *"R2PG-DM: A direct mapping from relational databases to property graphs"* (R.-A. Stoica, 2019) que utiliza como metodología de transformación el mapeo directo, continuando con *"Converting Relational to Graph Databases"* (De Virgilio, Maccioni, y Torlone, 2013) que presenta una metodología para la conversión de bases relacionales a bases de grafos enfocada en el desempeño de ciertas consultas. Además, se presentará la herramienta Neo4j ETL tool[1] proporcionada por Neo4j. Por último, el capítulo finaliza con la evaluación de los trabajos anteriores y algunas conclusiones sobre los mismos.

3.1. R2PG-DM: A direct mapping from relational databases to property graphs

En esta tesis, para la cual también se ha presentado el articulo *"On Directly Mapping Relational Databases to Property Graphs"* (R. Stoica, Fletcher, y Sequeda, 2019), los autores introducen un método de mapeo directo de bases de datos relacionales a bases de datos de grafos, más específicamente, property graphs. Esto significa que la transformación puede ser realizada automáticamente sin la necesidad de intervención manual.

El método planteado es independiente al dominio y al esquema de la base de datos de origen y transforma el contenido de una instancia de origen en una instancia destino, es decir, dada una instancia de base de datos relacional, se retorna su correspondiente instancia en una base de datos de grafos.

[1] https://neo4j.com/labs/etl-tool/

Éste método surge a partir de la comparación de una base de datos relacional y un grafo. Cómo se menciona en la sección 2, una tupla en una base de datos relacional es equivalente a un nodo en un grafo, y una clave foránea que relaciona a una tupla con otra es equivalente a una arista que conecta un nodo con otro. Entonces, interpretar el grafo como una base de datos relacional y viceversa resulta natural, y esto es algo que los autores destacan, ya que muchos métodos referenciados en éste artículo se enfocan en la mejora del rendimiento de las consultas, lo cuál es cuestionado por los autores ya que dichos métodos introducen un cambio en el modelo, lo cuál hace que los grafos resultantes sean muy complejos. Esto hace difícil a la tarea de comprobar que dichas transformaciones sean correctas. Dicho esto, el objetivo del método presentado es obtener una transformación que resulte natural desde el punto de vista semántico, con un grafo de salida que sea fácil de entender y cuya interpretación sea similar al esquema de relacional de la base de datos de origen.

Los autores proponen un método para la transformación de la instancia, así también como para la obtención de un esquema en forma de grafo a partir del esquema relacional de la base de datos de origen. Si bien no existe un lenguaje estándar para los esquemas de base de datos de grafos, los autores deciden que la codificación del esquema relacional en un *property graph* es la mejor opción. En este trabajo solo se presentará el algoritmo propuesto para transformar una instancia de una base de datos relacional y no para transformar el esquema, ya que escapa del objetivo principal.

El algoritmo planteado es simple y no implica reglas complejas para tomar decisiones. Toda tupla en la base de datos relacional es representada como un nodo, por lo que la base de datos de salida deberá tener la misma cantidad de nodos que la base de datos relacional tiene de tuplas. Éstos nodos no están vacíos sino que en el momento de la creación se le agregan propiedades, las cuales son generadas a partir de los atributos de una relación y los valores de sus respectivas tuplas. Esto permite que todos los nodos y sus propiedades sean creados en una sola recorrida de la base de datos.

Finalmente, para generar las aristas, se utilizan las claves foráneas como parámetro. Para cada arista encontrada en una relación, para todas las tuplas de dicha relación se busca la tupla a la que hace referencia esa clave foránea, luego se busca el nodo que se creó a partir de esta tupla y al final se genera la arista entre ambos nodos. Esto tiene la desventaja de que todos los nodos deben haber sido creados para poder crear las aristas entre ellos.

A continuación se detalla una descripción más formal del algoritmo:

1. **Creación de nodos**: Por cada tupla de una relación $r \in \mathbf{R}$ se crea un nodo en el grafo. Cada nodo tiene como etiqueta al nombre de la

relación r y tiene un identificador único generado por una función id.

2. **Creación de aristas**: Sean r_1 y r_2 dos relaciones de $\mathbf{R}$, dos nodos n_1 y n_2 tal que n_1 es un nodo generado a partir de una tupla $t_1 \in r_1$ y n_2 un nodo generado a partir de una tupla $t_2 \in r_2$. Si existe una clave foránea de r_1 a r_2 tal que la tupla t_1 está relacionada a la tupla t_2, se crea una arista desde el nodo n_1 al nodo n_2. Dicha arista contiene un identificador único generado mediante una función id y una etiqueta formada por la concatenación del nombre de la relación r_1, el símbolo $'-'$ y el nombre de la relación r_2.

3. **Creación de propiedades**: Para cada atributo de la tupla t en la relación r, se agrega una propiedad al respectivo nodo de t con el formato "$\{0\} : \{1\}$", donde $\{0\}$ es el nombre del atributo y $\{1\}$ es su valor para la tupla t.

Los autores demuestran que las siguientes cuatro propiedades son satisfechas por el grafo resultante: preservación de la información, preservación de las consultas, monotonicidad, y preservación de la semántica. A continuación se definen cada una de ellas.

1. Preservación de la información: Un mapeo directo preserva la información si ninguna información sobre la instancia de la base de datos relacional se pierde durante el proceso de transformación.

2. Preservación de las consultas: Un mapeo directo preserva las consultas si para cada consulta que se realiza en la base de datos relacional, existe una consulta equivalente en el *property graph* obtenido y los resultados de ambas consultas son similares.

3. Preservación de la semántica: Dado un esquema relacional $\mathbf{R}$, un conjunto Σ de PKs y FKs (claves primarias y foráneas) sobre $\mathbf{R}$ y su instancia correspondiente I, se dice que un mapeo preserva la semántica si genera un grafo consistente a partir de la instancia I cuando el conjunto Σ es consistente, y genera un grafo inconsistente en caso contrario.

4. Monotonicidad: Sean I_1 y I_2 dos instancias del esquema relacional $\mathbf{R}$. I_1 es un subconjunto de I_2 ($I_1 \subseteq I_2$) si para cada relación $R \in \mathbf{R}$, el conjunto de tuplas de R para la instancia I_1 es un subconjunto del conjunto de tuplas de R para la instancia I_2. Se dice que un mapeo directo es monótono si por cada par de instancias (I_1, I_2), el grafo obtenido de la instancia I_1 es un sub-grafo del grafo obtenido de la instancia I_2.

De esta manera los autores presentan un método que sigue un mapeo directo y especifican los pasos para su ejecución, a continuación se presenta un enfoque diferente para la transformación.

3.2. R2G: Converting Relational to Graph Databases

En este trabajo los autores plantean un método para la conversión de bases de datos relacionales a bases de datos de grafos partiendo desde la premisa de que, en muchos escenarios, la representación de los datos utilizando grafos es más natural, por ejemplo, en la web semántica o redes sociales. Los autores afirman que en estos escenarios donde los datos están altamente conectados una base relacional no es ideal ya que las operaciones de *JOIN* pueden llegar a ser muy complejas debido a la alta interconexión de los datos, y operaciones básicas pueden llegar a ser ineficientes y poco escalables. Por esto, su objetivo fue generar un método que dé como resultado un grafo donde estas consultas sean menos costosas. Los autores además presentan un mecanismo para la transformación de consultas, el cual no se presentará dado que escapa a los objetivos de este trabajo.

El método presentado por los autores para la transformación de base de datos relacionales permite obtener como resultado un grafo de tipo *property graph* teniendo como información inicial la base de datos en cuestión, su esquema y sus restricciones correspondientes.
Para comprender el funcionamiento de este método se presentarán algunos conceptos, comenzando con el de **grafo de esquema relacional**. Formalmente, este esquema es un grafo dirigido RG que representa al esquema relacional **R** tal que:

1. Existe un nodo $A \in N$ por cada atributo A de una relación en **R**
2. Existe una arista $(A_i, A_j) \in E$ si se cumple:
   a) A_i pertenece a una clave de una relación R en **R** y A_j no pertenece a ninguna clave de R
   b) A_i y A_j pertenecen a una clave de una relación R en **R**
   c) A_i y A_j pertenecen a R_i y R_j respectivamente, y existe una clave foránea entre $R_i.A_i$ y $R_j.A_j$

   Siendo N un conjunto de nodos y E el conjunto de aristas.

Para los nodos que forman parte de este grafo de esquema relacional los autores presentan una clasificación de tres diferentes tipos: 1. Nodo **hub** que posee más de una arista entrante, 2. Nodo **source** sin aristas entrantes y 3. Nodo **sink** que no posee aristas salientes. Luego de realizada esta clasificación, se denomina **full schema path** a los caminos que van desde un nodo de tipo $source$ a un nodo de tipo $sink$. Por último, dada una relación $R \in r$, A_i es un **atributo n2n** si pertenece a una clave $K = A_1, ..., A_n \in R$ y

para cada $A_j \in K$ existe una clave foránea entre $R.A_j$ y $R'.B$ donde $R' \in r$ y $R \neq R'$.

La idea principal del algoritmo es almacenar en un mismo nodo, valores que probablemente sean devueltos juntos en una consulta. Los autores denominan a estos valores como aquellos que pertenecen a tuplas que cumplen con la definición de tuplas *joinables*. Para comprender mejor este concepto se presenta una base de datos que sigue el modelo entidad-relación presentado en el capitulo 2 en la Figura 2.1, compuesta por las tablas 3.1 3.2 y 3.3.
Dos tuplas $t_1 \in R_1$ y $t_2 \in R_2$ se dicen **joinables** si existe una clave foránea *A* en R_1 que referencia a la clave primaria o clave única *B* en R_2 tal que $t_1[A] = t_2[B]$. Siguiendo esta definición, en el ejemplo, las tuplas *{John, Canelones, ...}* de la tabla *Visitas* son joineables con la tupla *{Canelones, pequeño}* de la tabla *Lugares* dado que Canelones es una clave primaria en ambas tablas.

nombre	fecha-nacimiento
Jane	26-05-1993
John	05-10-1995
Mark	03-11-1997

Tabla 3.1: Personas

lugar	tamaño
Canelones	pequeño
Montevideo	grande

Tabla 3.2: Lugares

persona	**lugar**	**fecha**
John	Canelones	2005-05-13
John	Canelones	2007-10-10
John	Canelones	2012-12-05
Mark	Montevideo	2013-04-21

Tabla 3.3: Visitas

Este concepto permite identificar qué valores deberían permanecer juntos, sin embargo los autores mencionan que si solo se utiliza el concepto de tuplas *joinables* es fácil generar nodos que estén sobrecargados de información, lo cual no es ideal. Para solucionar esto, introducen el concepto de **unificabilidad** para la agrupación de los datos, el cuál consiste en lo siguiente: dos valores $v1$ y $v2$ pertenecientes a una base de datos relacional r son unificables si se cumple alguno de los siguientes puntos:

1. existe una tupla t de una relación $R \in r$ tal que $t[A] = v1$ y $t[B] = v2$ y además A y B no son atributos n2n.

2. existe un par de tuplas joinables $t_1 \in R_1$ y $t_2 \in R_2$ tal que $t1[A] = v1$ y $t2[B] = v2$ y A es un atributo n2n.

3. existe un par de tuplas joinables $t_1 \in R_1$ y $t_2 \in R_2$ tal que $t1[A] = v1$ y $t2[B] = v2$ y ambos A y B no son n2n y no existe otra tupla $t3$ que sea joinable con $t2$.

En el ejemplo que se considero anteriormente las tuplas *{John, Canelones, ...}* de la tabla *Visitas* son joineables con la tupla *{Canelones, pequeño}* de la tabla *Lugares* y además son unificables dado que se cumple el caso número 2 ya que $lugar$ pertenece a una clave compuesta.

Con estos conceptos en mente, a continuación, se presenta el algoritmo planteado por los autores.

3.2.1. Algoritmo de transformación

La información inicial disponible está formada por: la base de datos relacional *r*, el esquema relacional *R*, y el conjunto *SP* que contiene todos los *full schema paths* del grafo de esquema relacional generado (*RG*). Utilizando estos datos como entrada, una vez ejecutado el algoritmo se obtendrá como salida una base de datos de grafos $g = (N, E)$.

El algoritmo itera sobre los caminos pertenecientes al conjunto SP y en cada iteración el correspondiente camino ($sp = A_i \rightarrow \ldots \rightarrow A_n$) se analiza desde el atributo $source\ A_i$ hasta el atributo $sink\ A_n$. Cada A_i corresponde a un atributo para alguna relación $R \in r$. En el ejemplo, un camino válido está formado por $sp : Visitas.persona \rightarrow Personas.nombre \rightarrow Personas.fecha - nacimiento$.

En cada iteración se actualiza el registro de los atributos visitados y se agrega el actual atributo (A_i) a un conjunto VS (en caso de que no haya sido agregado previamente en otra iteración). El atributo A_i es clasificado según **5** posibles casos que determinarán la estructura final del grafo. A continuación se detalla cada uno en aspectos generales, en la sección 3.4 se puede encontrar la evaluación de este algoritmo utilizando una base de datos de ejemplo y los detalles de cada iteración y cada atributo del camino.

Caso 1 (Nuevos nodos) El atributo A_i es de tipo *source*, y tanto A_i como A_{i+1} no fueron visitados. En este caso se generan nuevos nodos.

Para cada valor v asociado al atributo A_i, se crea un nuevo nodo y se incluye en cada nodo la propiedad $\langle A_i, v \rangle$. Por último, se agrega el atributo al conjunto de atributos visitados *VS*.

Caso 2 (Nuevos atributos) El elemento A_i es de tipo *source*, A_i aún no fue visitado, pero el siguiente atributo A_{i+1} ya fue visitado anteriormente.

En este caso nos encontramos frente a una clave foránea. Dado que A_{i+1} ya fue visitado, existe un nodo con la propiedad $\langle A_{i+1}, v\rangle$ tal que v pertenece al conjunto de valores de la base de datos para el atributoA_i . Para cada valor v se busca el nodo que contenga el par $\langle A_{i+1}, v\rangle$ y se inserta una nueva propiedad $\langle A_i, v\rangle$.

Caso 3 (Nuevos atributos) El elemento A_i aún no ha sido visitado y no es de tipo *source*, *hub* y tampoco es de tipo *n2n*.

En este caso, es necesario iterar por todos los nodos generados o actualizados cuando se analizó A_{i-1}. En cada nodo donde se insertó la propiedad $\langle A_{i-1}, v_1\rangle$ también se tiene que insertar la propiedad $\langle A_i, v_2\rangle$. Aquí hay que diferenciar si A_i y A_{i-1} están en la misma tupla o si se está siguiendo una clave foránea. Si ambos atributos están en la misma relación, significa que v_2 es v_1 (misma tupla), y en caso de que pertenezcan a relaciones distintas, se debe encontrar la tupla que contiene al valor v_1 y extraer el valor v_2 de dicha tupla.

Caso 4 (Nuevos nodos y aristas) El elemento A_i no ha sido visitado y es de tipo *hub* o es un atributo $n2n$.

Al igual que en el caso anterior, se debe iterar por todos los nodos generados o actualizados cuando se analizó el atributo A_{i-1}.
Primero, para cada valor *v* asociado al atributo actual A_i, se genera un nuevo nodo con la etiqueta y se agrega la propiedad $\langle A_i, v\rangle$. Luego se genera una arista entre el nuevo nodo y el nodo que fue generado o actualizado cuando se recorrió A_{i-1}. Además, a dicha arista se le agrega una etiqueta que se genera mediante la concatenación del nombre de la relación *R* y el atributo A_{i-1}.

Caso 5 (Nuevas aristas) El último caso se da cuando se itera sobre el último camino y en particular el último atributo.

En este caso se vinculan mediante una arista dos nodos generados en pasos anteriores.

Luego de recorrer todos los caminos el resultado es un grafo que cumple las propiedades deseadas por los autores.

3.3. Neo4j ETL Tool

Neo4j [2] es un motor de bases de datos orientado a grafos desarrollado por Neo4j, Inc. En su versión para escritorio una de las aplicaciones disponibles para instalar como accesorio (comúnmente denominado *plugin* por su nombre en inglés) es Neo4j ETL Tool[3]. Esta aplicación fue desarrollada para permitir importar los datos de sistemas relacionales a una base de datos orientada a grafos Neo4j.

La herramienta cuenta con una primera interfaz que es posible observar en la Figura 3.1 la cual permite agregar una nueva conexión con una base de datos relacional. Los sistemas soportados son: PostgreSQL, mySQL, mssql, oracle, db2 y jdbc. Luego, el usuario debe elegir la instancia de Neo4j en la cual desea importar los datos y habiendo realizado estos dos pasos se podrá seleccionar la opción de mapeo, la cual extrae la información de la base de datos relacional y genera la estructura que tendrá la base de grafos para poder presentársela al usuario.

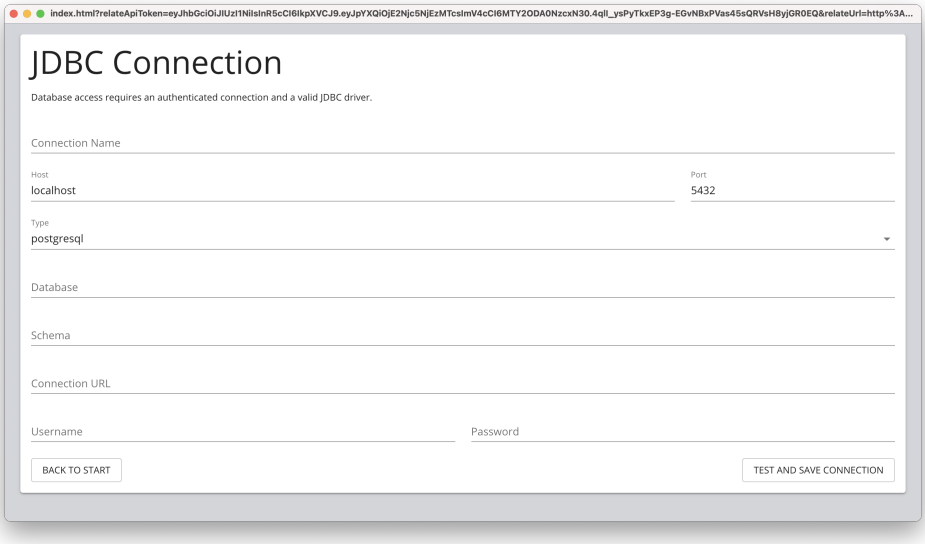

Figura 3.1: Neo4j ETL tool: Interfaz de nuevas conexiones.

Cuando el mapeo termina se le presenta al usuario una segunda interfaz de exploración y edición, que se puede observar en la Figura 3.2. En este punto los datos aún no han sido importados en la base de datos de grafos

[2] https://neo4j.com/
[3] https://neo4j.com/labs/etl-tool/

sino que el usuario podrá editar nombre de entidades y relaciones, remover aquellas que no quisiera importar y cambiar tipos si fuera necesario. La última acción es la de importar los datos una vez terminada la edición obteniendo el grafo de Neo4j resultante de la transformación.

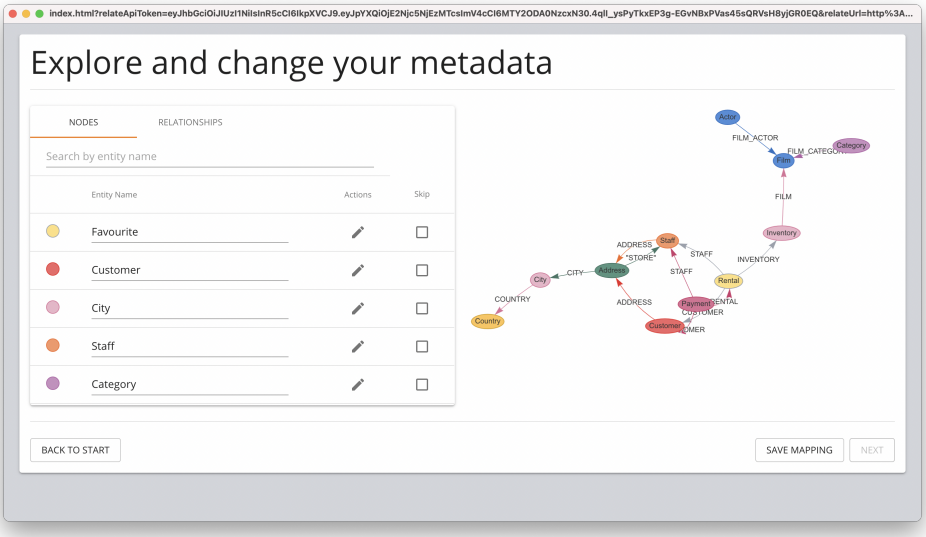

Figura 3.2: Neo4j ETL tool: Interfaz de exploración y edición.

El algoritmo utilizado detrás de esta herramienta es explicado con detalle en el blog *"Introducing the Newest RDBMS-to-Neo4j ETL Tool"* (Fernandes, 2016). A grandes rasgos el algoritmo considera los siguientes escenarios:

- Toda la información que se almacena en tablas y relaciones de la base relacional será almacenada en nodos de Neo4j los cuales tendrán propiedades. Las relaciones representadas por claves de la base relacional serán almacenadas como relaciones entre nodos, que representan las aristas del grafo.
- Tablas de relación binaria. Si dos entidades R_i y R_j que están vinculadas mediante la tabla de relación R_k (generalmente utilizada para representar relaciones $many - to - many$), las tuplas pertenecientes a R_k son representadas como aristas. En caso de que R_k solo contenga dos atributos; uno para referenciar a la tupla en R_i y otro para representar a la tupla en R_j, solo se representa mediante una arista que conecta a los respectivos nodos. Por otro lado, si la tupla contiene más atributos, es decir atributos de relación, los mismos son agregados como propiedades a la arista.

- Tablas de relación ternaria o mayor. Debido a que una arista sólo tiene dos extremos, para tablas que relacionan a tres o más entidades se deben generar nodos que interconectan a los nodos que representan dichas entidades.

Entonces, luego de analizar la base de datos relacional y diferenciar cada uno de estos escenarios la herramienta genera archivos CSV que serán importados a la base de datos de grafos mediante la herramienta *neo4j-import*[4]. Esto resulta en una base de datos de grafos que contiene los datos de la base relacional que se ha decidido migrar.

3.4. Evaluación de los enfoques

Durante esta sección, la cual quizás sea la sección más atractiva del capítulo, se evaluarán los algoritmos presentados anteriormente en las secciones 3.1, 3.2 y la herramienta Neo4j ETL Tool 3.3. El objetivo de esta evaluación será obtener una idea general de cómo se comportan y el rendimiento que tienen los diferentes enfoques, así como sus ventajas y desventajas para finalmente presentar una breve conclusión de los mismos.

Para esto, a lo largo de la sección se utilizará una base de datos de prueba que sigue el modelo entidad-relación presentado en el capítulo 2 en la Figura 2.1. La base relacional está compuesta por las tablas que se presentan a continuación.

- **Personas** representa la entidad Persona y contiene su nombre y fecha de nacimiento. El atributo *nombre* representa la clave primaria.
- **Lugares** representa la entidad Lugar y contiene el nombre de una ciudad y su tamaño. El atributo *lugar* representa la clave primaria.
- **Visitas** Representa las veces que una persona visitó un lugar. La tabla contiene el nombre de la persona, el lugar que visitó y la fecha en que visitó dicho lugar. El atributo *persona* es clave foránea a la tabla *Personas* y el atributo *lugar* es clave foránea a la tabla *Lugares*.
- **Conocidos** Es una tabla de relación que vincula a dos personas que se conocen. Los atributos *persona1* y *persona2* son claves foráneas a la tabla *Personas*.

La base presentada será utilizada con el fin de poder evaluar manualmente el grafo resultante, ya que la misma tiene un tamaño pequeño.

[4] https://neo4j.com/docs/operations-manual/current/tutorial/neo4j-admin-import/

nombre	fecha-nacimiento
Jane	26-05-1993
John	05-10-1995
Mark	03-11-1997

(a) Personas

lugar	tamaño
Canelones	pequeño
Montevideo	grande

(b) Lugares

persona	**lugar**	**fecha**
John	Canelones	13-05-2005
John	Canelones	10-10-2007
Mark	Canelones	12-05-2012
Mark	Montevideo	21-94-2013

(c) Visitas

persona1	**persona2**
John	Jane
John	Mark

(d) Conocidos

Figura 3.3: Tablas existentes en la base datos de ejemplo

Por otro lado, con inspiración en el articulo *"Which Category Is Better: Benchmarking Relational and Graph Database Management Systems"*(Cheng, Ding, Wang, Lu, y Du, 2019), se utilizará el estándar de comparación de TPC-H[5] para evaluar las herramientas presentadas en las secciones 3.1 y 3.3 siendo esta una base de datos de mayor porte que se asemeja a una base de datos real. TPC-H es un estándar de comparación que consiste en un conjunto de datos y consultas de alta complejidad sobre dichos datos, que fueron elegidos en base a su relevancia en la industria. El modelo entidad-relación que representa esta base de datos se puede observar en la Figura 3.4.

TPC-H provee una herramienta para generar el esquema de la base de datos, y también para popular la misma en base a un factor. Los factores disponibles son 1, 10, 100 y 1000, los cuáles generan una base de datos de tamaño 1Gb multiplicado por el tamaño del factor. Para este trabajo se eligió utilizar la base de datos generada con el factor 1, el cuál genera una base de datos con 8661245 tuplas.

Independientemente del factor elegido, la base de datos generada contiene las siguientes tablas:

- **Supplier** representa a los proveedores que comercializan productos.
- **Part** representa a los productos.
- **Customer** representa a los compradores.
- **Partsupp** contiene el stock y el precio de un producto para un vendedor en particular.

[5] https://www.tpc.org/tpch/

- **Orders** representa las órdenes de un producto adquirido por un comprador.
- **Lineitem** contiene datos asociados a una orden, cómo por ejemplo, que proveedor lo vendió, la cantidad que se vendió, los impuestos a pagar, etc.
- **Nation** representa a las naciones a las que pertenecen los proveedores y compradores.
- **Region** representa a las regiones a las que pertenece una determinada nación.

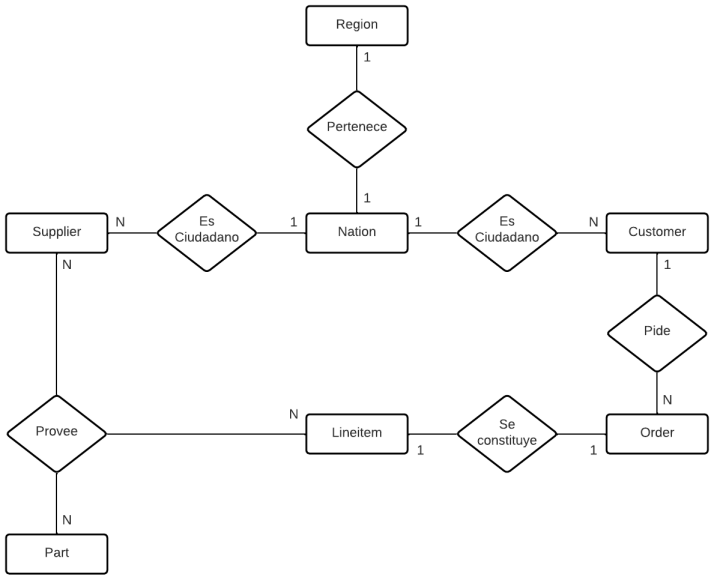

Figura 3.4: MER (Modelo entidad-relación) de TPC-H

Debido a que el método presentado en la sección 3.2 no cuenta con una implementación disponible, no se realizará la conversión de la base de datos de TPC-H para este método, por lo tanto, solo se realizará la conversión la aplicación que implementa el método presentado en la sección 3.1 y para la aplicación ETL Tool de Neo4J presentada en 3.3.

R2PG-DM: A direct mapping from relational databases to property graphs

Para la evaluación de este enfoque presentado en 3.1 se utilizó la aplicación brindada por los autores. Esta aplicación realiza la transformación

tomando como entrada una base de datos relacional y soporta los siguientes motores de bases de datos relacionales: MySQL, PostgreSQL, SQLite y Oracle. Mientras que para su salida el motor de base de datos no relacional admitido es Neo4j.

La aplicación no posee una interfaz de usuario sino que debe ser ejecutada mediante líneas de comando. Esto requiere que el usuario que desee utilizarla o evaluarla tenga un mínimo conocimiento en el área para lograr compilar la aplicación y obtener un archivo ejecutable siguiendo las instrucciones provistas, lo cual se puede ver como una desventaja.
Para lograr la ejecución, primero se realizó la instalación local de varias aplicaciones y paquetes como JAVA SE 8, Maven, entre otras; pero se encontraron algunos errores en el código que impedían la ejecución exitosa, por lo cual fue necesario editar algunas partes del código fuente para que fuera posible de utilizar. Luego de esto, fue posible compilar la aplicación y obtener un ejecutable.

Para la conversión, además de la base de datos relacional de entrada que se desea transformar, la aplicación utiliza una base de datos intermedia. En esta base de datos intermedia se almacenan los datos necesarios para poder construir la base de datos final. La base de datos intermedia está constituida por tres tablas: *Nodos*, *Aristas* y *Propiedades*. Los nodos se almacenan con sus respectivas propiedades, y las aristas son quienes conectaran dichos nodos. A partir de esta base intermedia, se generan archivos de formato *.csv* (De su nombre en inglés: comma-separated values) que luego serán importados desde la aplicación de Neo4J y permitirán generar la base de datos final.

Para evaluar la aplicación, como primer paso, se ha utilizado la base de datos de ejemplo. Para la misma, luego del análisis manual del grafo resultante 3.5 se corrobora que las entidades y relaciones de la base de datos relacional se han mantenido correctamente sin pérdida de información. Sin embargo se debe tener en cuenta que la base de datos utilizada es pequeña y con bajo volumen de datos.

En cuanto a la base de datos de TPC-H, al momento de realizar la conversión, la aplicación falla luego de varias horas de ejecución arrojando un error que especifica un error en una consulta a la base de datos intermedia utilizada para la conversión. Debido al mensaje y el tipo de error, se intuye que el fallo se debe a un error en la implementación del algoritmo pero no es posible asegurarlo con certeza y tampoco se realizó una investigación profunda de la causa ya que escapa al objetivo de este trabajo.

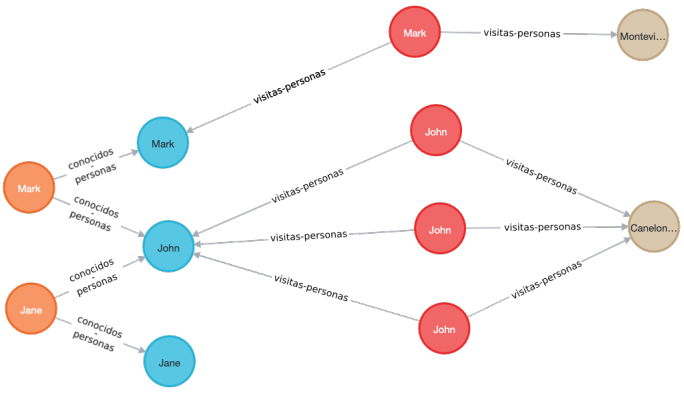

Figura 3.5: Grafo resultante de la ejecución mediante la aplicación R2PG-DM

R2G: Converting Relational to Graph Databases

El algoritmo planteado por los autores de este trabajo fue implementado en Java, lamentablemente el código de esta implementación no se encuentra disponible. Por esto, se decidió realizar una ejecución manual siguiendo el paso a paso que fue detallado por los autores y mencionado en el capítulo 3.2.1.

Para comenzar la evaluación se utiliza la base de datos de ejemplo presentada en el inicio del capítulo, constituída por las tablas $persona$, $lugares$, $visitas$ y $conocidos$. Antes de poder ejecutar el algoritmo es necesario generar el **grafo de esquema relacional** y para la obtención del mismo se utilizó la siguiente definición: dado un esquema relacional **R**, el esquema relacional de grafo RG de **R** es un grafo dirigido $\langle N, E \rangle$ tal que:

1. Existe un nodo $A \in N$ para cada atributo A de una relación R.
2. Existe una arista $(A_i, A_j) \in E$ si se cumple alguna de las siguientes condiciones:
   - A_i pertenece a una clave de la relación R en **R** y A_j no es una clave en R.

- A_i, A_j pertenecen a una clave de la relación R en **R**.
- A_i, A_j pertenecen a R_i y R_j respectivamente y existe una clave foránea entre $R_i.A_i$ y $R_j.A_j$.

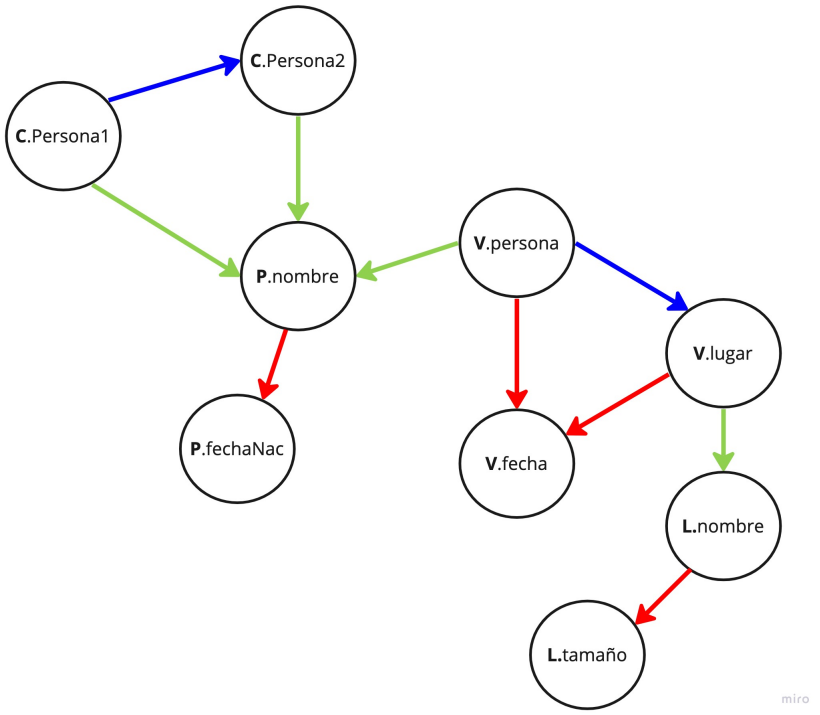

Figura 3.6: R2G: Grafo que representa el esquema relacional de la base de datos de ejemplo

Luego de aplicar esta definición a la base de datos de ejemplo se obtiene como resultado el grafo de la Figura 3.6. Se clasifican los nodos de este grafo como: hub, $sink$ o $source$ y se obtienen los **full schema paths**, esto es, los caminos que van desde un nodo $source$ a un nodo $sink$. Ambos conceptos se encuentran detallados en la sección 3.2. En la Figura 3.7 se puede observar los caminos obtenidos.

Habiendo obtenido los caminos, el algoritmo posee la información necesaria para empezar su ejecución, esto es, iterar sobre estos caminos, analizar sus atributos y comenzar la generación de nodos y aristas. A continuación se detalla la ejecución y cada iteración del algoritmo aplicado a los caminos obtenidos, comenzando por el primer camino $sp1$. Se resaltan en negrita el atributo que la iteración está analizando (A_i y su sucesor A_{i+1}) y teniendo en cuenta los casos presentados en 3.2.1 se determinará cuál es el

- $sp1 : V.persona \rightarrow P.nombre \rightarrow P.fechaNacimiento$
- $sp2 : V.persona \rightarrow V.lugar \rightarrow V.fecha$
- $sp3 : V.persona \rightarrow V.fecha$
- $sp4 : V.persona \rightarrow V.lugar \rightarrow L.nombre \rightarrow L.tamao$
- $sp5 : C.persona1 \rightarrow P.nombre \rightarrow P.fechaNacimiento$
- $sp6 : C.persona1 \rightarrow C.persona2 \rightarrow P.nombre \rightarrow P.fechaNacimiento$

Figura 3.7: Full Schema Paths

caso correspondiente. La ejecución comienza con el conjunto de atributos visitados VS vacío.

Primera iteración:

$$sp1 : \textbf{V.persona} \rightarrow \textbf{P.nombre} \rightarrow P.fechaNacimiento$$

$$A_i = V.persona,\ A_{i+1} = P.nombre$$
$$VS = \{\}$$

Dado que tanto A_i y A_{i+1} no han sido visitados A_i es $source$ se dan las condiciones del **Caso 1**.

Por lo tanto, para cada valor en el dominio de $V.persona$: $\{John, Mark\}$ (valores del atributo $persona$ en la tabla $Visitas$), se generan nuevos nodos: n1, n2. Luego, se incluyen las propiedades $\langle V.persona, John\rangle$, $\langle V.persona, Mark\rangle$ respectivamente.

Finalmente se agrega el atributo visitado al conjunto de atributos visitados $VS = \{V.persona\}$ y el conjunto final de nodos termina compuesto por $n = \{n1, n2\}$ presentado en la Figura 3.8

Segunda iteración

$$sp1 : V.persona \rightarrow \textbf{P.nombre} \rightarrow \textbf{P.fechaNacimiento}$$

$$A_i = P.nombre,\ A_{i+1} = P.fechaNacimiento$$
$$VS = \{\ V.persona\ \}$$

En esta iteración A_i no ha sido visitado y es hub, A_{i+1} tampoco ha sido visitado y es $sink$, se dan las condiciones del **Caso 4**.

Por lo tanto, se debe iterar sobre todos los nodos creados o actualizados cuando fue analizado A_{i-1}, en este caso $V.persona$, analizado en la primera

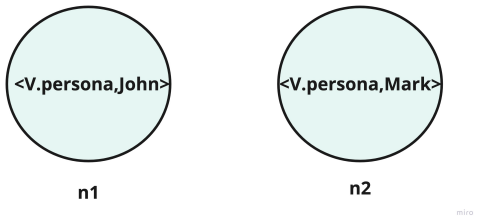

Figura 3.8: Grafo resultante - primera iteración sobre base de ejemplo.

iteración. Los nodos creados fueron: n1 y n2.
Se generan nuevos nodos por cada uno de los valores que toma el atributo A_i que corresponde a $P.nombre$ = { Jane, John, Mark }. El resultado son tres nuevos nodos: n3 = ⟨ P.nombre, Jane ⟩, n4 = ⟨ P.nombre, John ⟩, n5 = ⟨ P.nombre, Mark ⟩ y aristas desde n1,n2 hacia n3,n4,n5 con la etiqueta : $visitado\_V.persona$ como se observa en la Figura 3.9

Finalmente se agrega el atributo visitado al conjunto de atributos visitados $VS = \{V.persona, P.nombre\}$ y el conjunto final de nodos termina compuesto por $n = \{n1, n2, n3, n4, n5\}$

Tercera iteración

$$sp1 : V.persona \rightarrow P.nombre \rightarrow \textbf{P.fechaNacimiento}$$

$$A_i = P.fechaNacimiento,$$
$$VS = \{\ V.persona, P.nombre\ \}$$

En esta iteración se analiza el último atributo del camino. A_i no ha sido visitado y es $sink$. Se dan las condiciones del **Caso 3**.

Se debe iterar sobre todos los nodos que fueron creados o actualizados cuando fue analizado A_{i-1}, en este caso $P.nombre$, correspondiendo a los nodos: n3,n4 y n5.
Primero se analiza si estos atributos pertenecen a la misma relación, siendo este el caso. Luego, para cada nodo donde se insertó la propiedad $\langle A_{i-1}, v1\rangle$ se insertará la propiedad $\langle A_i, v2\rangle$, siendo $v1$ y $v2$ valores de estos atributos para una misma tupla. Por lo tanto, se inserta la fecha de nacimiento $\langle P.fechaNacimiento, \{fecha\}\rangle$ para los nodos: n3,n4,n5 ya existentes, como se observa en la Figura 3.9, donde se resaltan estas nuevas propiedades en color verde.

Finalmente se agrega el atributo visitado al conjunto de atributos visitados $VS = \{V.persona, P.nombre, P.fechaNacimiento\}$ y el conjunto final de

nodos está compuesto por $n = \{n1, n2, n3, n4, n5\}$

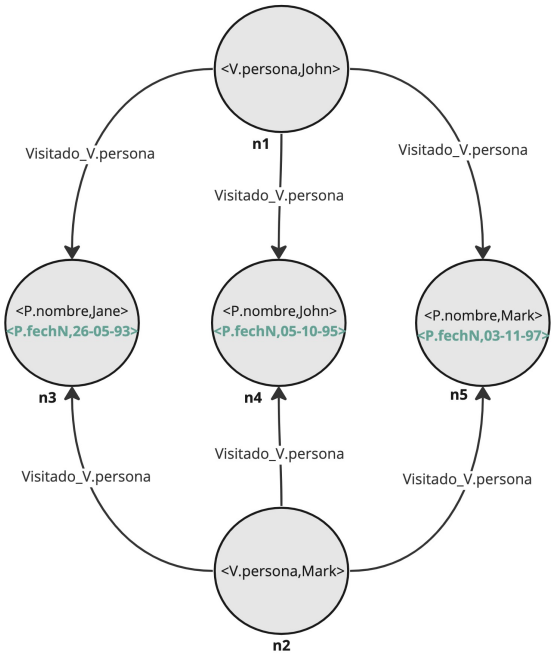

Figura 3.9: Grafos resultantes - segunda y tercera iteración respectivamente sobre base de ejemplo.

Cuarta iteración

Dado que se completó la iteración sobre el primer camino, se continúa con $sp2$, el cuál fue definido en la Figura 3.7.

$$sp2 : \textbf{V.persona} \rightarrow \textbf{V.lugar} \rightarrow V.fecha$$

$$A_i = V.persona,\ A_{i+1} = V.lugar,$$
$$VS = \{\ V.persona,\ P.nombre,\ P.fechaNacimiento\ \}$$

En esta ocasión A_i ya fue visitado, por lo cual no se generan nodos o aristas durante esta iteración.

Quinta iteración

$$sp2 : V.persona \rightarrow \textbf{V.lugar} \rightarrow \textbf{V.fecha}$$

$$A_i = V.lugar,\ A_{i+1} = V.fecha,$$
$$VS = \{\ V.persona,\ P.nombre,\ P.fechaNacimiento\ \}$$

En esta iteración A_i no ha sido visitado y es n2n, A_{i+1} tampoco ha sido visitado y es $sink$, además A_{i-1} es $source$. Se dan las condiciones del **Caso 4**.

Se debe iterar sobre cada uno de los nodos creados o actualizados cuando fue analizado A_{i-1}, en este caso, $V.persona$ correspondiendo a los nodos: n1 y n2 (creados en la primera iteración).
Se generan nuevos nodos para cada uno de los valores del atributo A_i, esto es, $V.lugar$ = { Canelones, Montevideo }. El resultado es el de dos nuevos nodos n6 = ⟨ V.lugar, Montevideo ⟩ y n7 = ⟨ V.lugar, Canelones ⟩. Luego, se generan aristas desde los nodos n1 y n2 hacia n6 y n7 donde la arista tendrá la etiqueta $visitado\_V.persona$ como se observa en la Figura 3.10.

Finalmente se agrega el atributo visitado al conjunto de atributos visitados $VS = \{V.persona, P.nombre, P.fechaNacimiento, V.lugar\}$ y el conjunto final de nodos está compuesto por $n = \{n1, n2, n3, n4, n5, n6, n7\}$

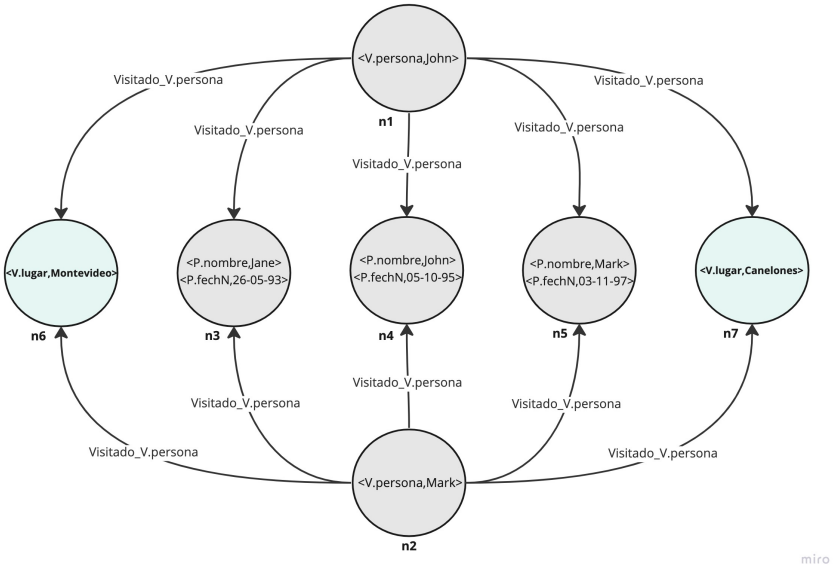

Figura 3.10: Grafo resultante - quinta iteración sobre base de ejemplo.

Sexta iteración

$$sp2 : V.persona \rightarrow V.lugar \rightarrow \textbf{V.fecha}$$

$$A_i = V.fecha,$$
$$VS = \{\ V.persona,\ P.nombre,\ P.fechaNacimiento,\ V.lugar\ \}$$

En esta iteración A_i no ha sido visitado y es sink, se dan las condiciones del **Caso 3**.

Se debe iterar sobre todos los nodos que fueron creados o actualizados cuando se analizó A_{i-1}, en este caso, $V.lugar$, esto es, los nodos: n6 y n7 creados en la quinta iteración.
Primero se analiza si estos atributos pertenecen a la misma relación, siendo este el caso. Luego, para cada nodo donde se insertó la propiedad $\langle A_{i-1}, v1\rangle$ se insertará la propiedad $\langle A_i, v2\rangle$, siendo $v1$ y $v2$ valores de estos atributos para una misma tupla. Por lo tanto, se inserta la fecha de visita $\langle V.fecha, \{fecha\}\rangle$ para los nodos n6, n7.

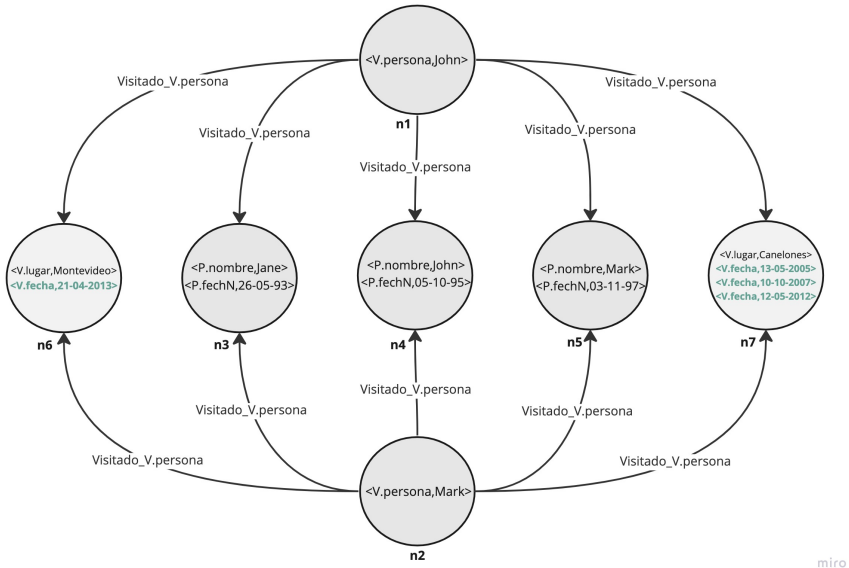

Figura 3.11: Grafo resultante - sexta iteración sobre base de ejemplo.

Luego de esta iteración se obtienen **nodos con más de una fecha de visita** como se observa en la Figura 3.10 (n6 y n7) y no es posible diferenciar a qué persona corresponde cada fecha. No está claro cómo debería proceder el algoritmo en este caso donde se da más de una inserción para el mismo nodo, ya que el valor existe en más de una tupla de la relación.

Por esta razón, se decidió ejecutar el algoritmo paso a paso sobre el ejemplo utilizado por los autores en su trabajo, el cual es utilizado de ejemplo para la descripción de el algoritmo, en busca de verificar las iteraciones que se han realizado o encontrar algún detalle que no se tuvo en cuenta en la ejecución planteada anteriormente.

Análisis sobre base de ejemplo planteada por los autores Con el objetivo de corroborar el resultado del algoritmo, se ejecutan los pasos del mismo sobre la base de ejemplo planteada por los autores representada por las tablas de la Figura 3.12. Hemos conservado los nombres originales utilizados por los autores.

User (US)

	uid	uname
t_1	u01	Date
t_2	u02	Hunt

Follower (FR)

	fuser	fblog
t_3	u01	b01
t_4	u01	b02
t_5	u01	b03
t_6	u02	b01

Tag (TG)

	tuser	tcomment
t_7	u02	c01

Blog (BG)

	bid	bname	admin
t_8	b01	Information Systems	u02
t_9	b02	Database	u01
t_{10}	b03	Computer Science	u02

Comment (CT)

	cid	cblog	cuser	msg	date
t_{11}	c01	b01	u01	Exactly what I was looking for!	25/02/2013

Figura 3.12: Base relacional utilizada en el artículo “R2G: Converting Relational to Graph Databases"

Al igual que en la evaluación anterior, a partir de la base relacional se genera el grafo de esquema relacional y se clasifican los nodos. El resultado que se ha obtenido es idéntico al de los autores por lo cual se procede a obtener los *full schema paths*. Los caminos obtenidos son los siguientes:

- $sp1 : FR.fuser \rightarrow US.uid \rightarrow US.uname$
- $sp2 : FR.fuser \rightarrow FR.fblog \rightarrow BG.bid \rightarrow BG.bname$
- $sp3 : FR.fuser \rightarrow FR.fblog \rightarrow BG.bid \rightarrow BG.admin \rightarrow US.uid \rightarrow US.uname$
- $sp4 : TG.tuser \rightarrow US.uid \rightarrow US.uname$

- $sp5 : TG.tuser \rightarrow TG.tcomment \rightarrow CT.cid \rightarrow CT.msg$
- $sp6 : TG.tuser \rightarrow TG.tcomment \rightarrow CT.cid \rightarrow CT.date$
- $sp7 : TG.tuser \rightarrow TG.tcomment \rightarrow CT.cid \rightarrow CT.cblog \rightarrow BG.bid \rightarrow BG.bname$
- $sp8 : TG.tuser \rightarrow TG.tcomment \rightarrow CT.cid \rightarrow CT.cuser \rightarrow US.uid \rightarrow US.uname$
- $sp9 : TG.tuser \rightarrow TG.tcomment \rightarrow CT.cid \rightarrow CT.cblog \rightarrow BG.bid \rightarrow BG.admin \rightarrow US.uid \rightarrow US.uname$

A continuación se detalla la ejecución y cada iteración del algoritmo aplicado a los caminos obtenidos, comenzando por $sp1$. Se resaltan en negrita el atributo que la iteración está analizando (A_i y su sucesor A_{i+1}) y teniendo en cuenta los casos presentados en 3.2.1 se determinará cual es el caso correspondiente. La ejecución comienza con el conjunto de atributos visitados VS vacío. Además se utilizará la misma nomenclatura que los autores.

Primera iteración:

$$sp1 : \textbf{FR.fuser} \rightarrow \textbf{US.uid} \rightarrow US.name$$

$$A_i = FR.fuser,\ A_{i+1} = US.uid$$
$$VS = \{\}$$

Dado que tanto A_i y A_{i+1} no han sido visitados A_i es $source$ se dan las condiciones del **Caso 1**.

Por lo tanto, para cada valor en el dominio de $FR.fuser$: $\{u01, u02\}$ (valores del atributo $fuser$ en la tabla $Follower$), se generan nuevos nodos: n1, n5. Luego, se incluyen las propiedades $\langle FR.fuser, u01\rangle$, $\langle FR.fuser, u02\rangle$ respectivamente, como se observa en la Figura 3.13.

Finalmente se agrega el atributo visitado al conjunto de atributos visitados $VS = \{FR.fuser\}$ y el conjunto final de nodos termina compuesto por $n = \{n1, n5\}$

Segunda iteración

$$sp1 : FR.fuser \rightarrow \textbf{US.id} \rightarrow \textbf{US.uname}$$

$$A_i = US.id,\ A_{i+1} = US.uname$$
$$VS = \{\ FR.fuser\ \}$$

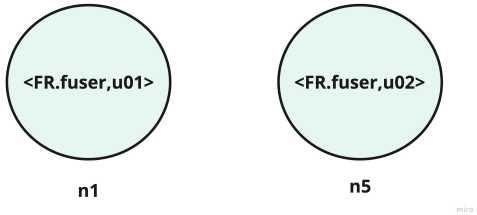

Figura 3.13: Grafo resultante - primera iteración sobre base de los autores.

En esta iteración A_i no ha sido visitado y es hub, A_{i+1} tampoco ha sido visitado y es $sink$, se dan las condiciones del **Caso 4**.

Por lo tanto, se debe iterar sobre todos los nodos creados o actualizados cuando fue analizado A_{i-1}, en este caso $FR.fuser$, analizado en la primera iteración. Los nodos creados fueron: n1 y n5.
Se generan nuevos nodos por cada uno de los valores que toma el atributo A_i que corresponde a $US.id$ = { u01, u02 }. El resultado son dos nuevos nodos: n2 = ⟨ US.id, u01 ⟩, n3 = ⟨ US.id, u02 ⟩ y aristas desde n1,n5 hacia n2,n3, con la etiqueta : $Follower\_FR.fuser$, como se observa en la Figura 3.14

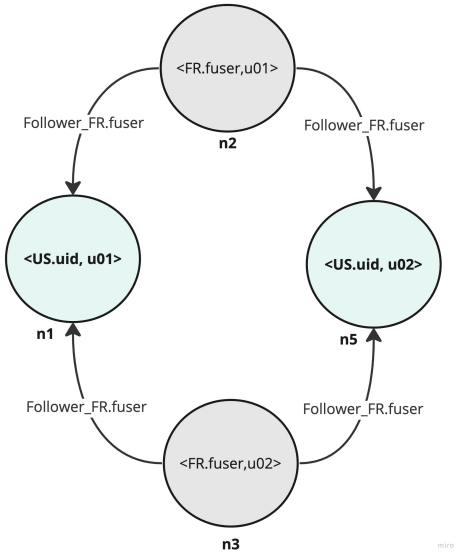

Figura 3.14: Grafo resultante - segunda iteración sobre base de los autores.

Finalmente se agrega el atributo visitado al conjunto de atributos visitados $VS = \{FR.fuser, US.id\}$ y el conjunto final de nodos termina compuesto por $n = \{n1, n5, n2, n3\}$

Luego de esta iteración se encuentra una **discrepancia** entre el estado del grafo y el resultado obtenido por los autores. El atributo $US.id$ con su respectivo valor se encuentra en los nodos n2 = ⟨ US.id, u01 ⟩ y n3 = ⟨ US.id, u02 ⟩ mientras que en el resultado de los autores este atributo y sus valores forma parte de los nodos ya creados en la primera iteración: n1 y n5.
Para que esto fuera posible, en la segunda iteración en lugar de evaluar el Caso 4 se debería evaluar el Caso 2 o el Caso 3 donde en lugar de generar nodos se agregan propiedades. Pero, ninguno de estos casos se cumplen ya que:

- El Caso 2 requiere que A_i sea source. Hipótesis que no se cumple dado que $A_i = US.id$ es hub.
- El Caso 3 requiere que A_i no sea source, hub ni n2n. Hipótesis que no se cumple dado que $A_i = US.id$ es hub.

Luego de evaluar detenidamente el trabajo presentado por los autores en busca de encontrar en qué lugar se podría haber cometido un error y llegada a la situación de no encontrar las respuestas buscadas se decidió contactar a los autores. El contacto fue vía correo electrónico, donde se les compartió un documento explicando el paso a paso de la ejecución como se ha hecho en esta sección, con detalles de cada una de las decisiones tomadas y referencia a su trabajo, planteando sobre el final la pregunta de si se ha evaluado incorrectamente el caso de la segunda iteración.

Los autores contestaron este correo muy rápidamente y con excelente disposición, para nuestra sorpresa, confirmaron la existencia de un error en el algoritmo y la ejecución que se había plantado por nuestra parte era correcta. Además, hicieron mención a que la imagen que presentan en su trabajo la cual representa el grafo resultante (Figura 5, si el lector desea corroborar) es la correcta y que el algoritmo debería arrojar ese resultado. Por lo cual, dado que no se dispone de los pasos correctos mediante los cuales los autores obtuvieron este grafo, no fue posible replicarlo.

Se concluye entonces la evaluación de este algoritmo prometedor pero para el cual se han encontrado algunos obstáculos, siendo el primero la ausencia de la implementación realizada por los autores. Luego, si bien se ha intentado analizar y evaluar manualmente el mismo, hemos encontrado inconsistencias en su ejecución. En un principio, ejecutando el mismos sobre la base de datos de ejemplo se ha encontrado un caso incierto, por lo cual se decidió ejecutarlo sobre la misma base de datos planteada por los autores donde se llegó a una discrepancia, que fue luego validada por los mismos.

Neo4j ETL Tool

Para la evaluación de esta herramienta, dado que la misma solo está disponible para escritorio, fue necesario instalar la aplicación de escritorio de Neo4j y luego agregar el complemento adicional *Neo4j ETL Tool*.

Primero se debió crear una base de datos de grafos donde se guardará el grafo resultante de la transformación, luego como primer paso de la interfaz es necesario introducir las credenciales necesarias para la conexión con la base de datos relacional PostgreSQL. A continuación, se debe elegir el esquema de base de datos que se desea migrar, esto resulta interesante dado que una base de datos que cuente con más de un esquema deberá repetir este paso para cada uno de estos esquemas.
Antes de comenzar con la transformación, se presenta una previsualización del grafo resultante donde se puede cambiar las etiquetas de los nodos y aristas. Finalmente, luego de confirmar que la pre visualización del grafo resultante es correcta, la transformación comienza.

Una de las desventajas de la aplicación que se encontró a primera vista es el hecho de que la herramienta no presenta información al usuario sobre el progreso de la transformación o si ha ocurrido algún tipo de error.

Al ejecutar la herramienta con la base de datos de ejemplo, el grafo obtenido en la previsualización, que se puede ver en la Figura 3.15 parece correcto. Todos los esquemas relacionales en la base de datos de origen fueron transformados y representados como nodos, así como también las claves foráneas fueron correctamente representadas como aristas.

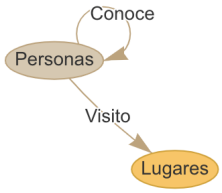

Figura 3.15: Neo4j ETL tool: Previsualización del grafo de ejemplo

Sin embargo, luego de confirmar la transformación y analizar el grafo final resultante que se puede observar en la Figura 3.16 fue posible apreciar que no todas las relaciones fueron agregadas al mismo. Para los casos de tablas que contienen atributos de relación, la aplicación no parece realizar una conversión correcta. La tabla $visitas$ 3.3 contiene dos entradas donde la persona $John$ visita $Canelones$ en dos fechas distintas, pero en el grafo de

la base de datos de salida, solo se agrega una de estas tuplas representada como una arista.

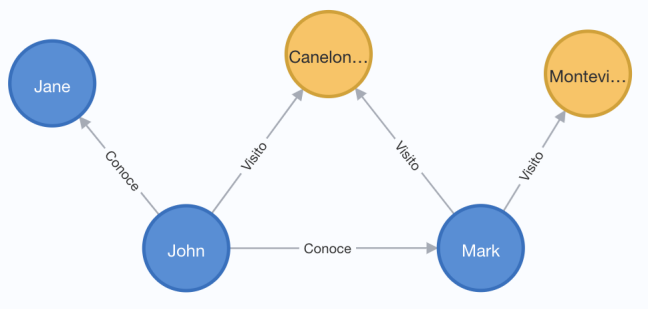

Figura 3.16: Neo4j: Grafo resultante a partir de base de datos de ejemplo

Si bien lo anterior ya concluye que la herramienta no parece funcionar de manera totalmente correcta, se procedió a probar la herramienta con la base de datos de TPC-H para evaluar su comportamiento.

Al realizar los mismos pasos mencionados anteriormente ahora sobre la base de datos de TPC-H con escala 1 y comenzar con la transformación, la aplicación transcurre varias horas de ejecución sin arrojar ningún tipo de error o información al usuario. Esto se repitió en dos equipos distintos llegando al mismo resultado: la ejecución no terminó. Por lo cual se concluye que la herramienta no está preparada para soportar transformaciones de bases de datos de mediano o gran porte como lo es la base de datos de TPC-H.

3.5. Conclusiones

Durante este capítulo se han analizado y evaluado trabajos relacionados con el objetivo de este proyecto. En un principio se evaluó el trabajo de los autores (R.-A. Stoica, 2019) basado en *Direct Mapping* el cual cuenta con una implementación en Java. Sin embargo, obtener un ejecutable donde se pudiera probar la implementación fue difícil, dado que el código posee algunos errores. Luego de varios intentos se pudo obtener este ejecutable, pero cabe destacar que la aplicación no posee una interfaz de usuario.
La evaluación de la aplicación en una base de datos de pequeño porte fue exitosa, por otro lado, para la evaluación sobre TPC-H la aplicación ha fallado. Si bien un enfoque posible hubiera sido continuar la mejora de esta herramienta, hemos optado por crear una nueva propuesta pero sí tener en cuenta *Direct Mapping* como el posible algoritmo a utilizar.

Por otro lado, el algoritmo presentado por (De Virgilio y cols., 2013) no cuenta con una implementación disponible y luego de la ejecución manual de él mismo se han encontrado algunas discrepancias con el ejemplo planteado por los autores. Esta discrepancia fue validada por los mismos luego de un intercambio de correos electrónicos. Dada esta situación, se ha optado por no tomar este algoritmo como base para la implementación de la propuesta.

Finalmente, si bien Neo4j tiene una aplicación de migración, la misma no realiza las transformaciones de manera correcta y tampoco puede completarlas cuando se trata de base de datos de mediano o gran porte.
A continuación en la Figura 3.17 se puede observar una tabla comparativa de los aspectos evaluados para cada uno de los enfoques que resume de manera visual lo ya presentado.

	R2PG-DM: A direct mapping from relational databases to property graphs	**R2G**: Converting Relational to Graph Databases	**Neo4j ETL Tool**
Implementación disponible	**Si**	**No**	**Si**
Interfaz disponible	**No**	**No**	**Si**
Ejecución en base de datos de pequeño porte	Ejecución exitosa	Ejecución manual fallida	Ejecución termina pero el grafo resultante presenta errores.
Ejecución en base de datos de gran porte TPC-H	Ejecución falla con errores.	No fue posible evaluarlo dado que no se cuenta con una implementación.	Ejecución no termina.

miro

Figura 3.17: Tabla comparativa de los tres enfoques evaluados.

Debido a estos resultados, se decidió optar por llevar a cabo una implementación propia con un motor de bases de datos distinto a Neo4j, siendo este ArangoDB. Dicha herramienta estará basada en el algoritmo de *Direct Mapping* y solo soportará bases de datos relacionales en PostgreSQL y la extensión a otros motores de bases de datos relacionales será evaluada luego de finalizada la implementación.

Capítulo 4

Propuesta

Siendo el objetivo de este proyecto investigar y crear una herramienta para la transformación automática de una base relacional a una base de grafos para el motor de base de datos ArangoDB, luego de analizar y evaluar trabajos relacionados al tema presentados en el capítulo 3 se decidió realizar una implementación propia basada en el algoritmo de *Direct Mapping*.

Antes de comenzar la implementación, se realizó una videollamada con miembros del equipo de ArangoDB donde ofrecieron su disposición para evacuar dudas que pudieran surgir y con quienes además se validó la idea. A su vez, proporcionaron artículos de utilidad para cumplir el objetivo.

En este capítulo se detalla la construcción de dicha herramienta, desde la toma de decisiones y diseño hasta su implementación.

4.1. Diseño

El diseño de la herramienta parte desde la base de que el usuario pueda de alguna manera proveer las credenciales para la conexión a ambas bases de datos (PostgreSQL y ArangoDB) y que simplemente mediante una confirmación se de comienzo a la transformación automática, obteniendo como resultado una base de datos en ArangoDB con las mismas características y datos que la base relacional en PostgreSQL.

Aplicación de escritorio Para facilitar la interacción entre el usuario y la herramienta, se realizó una aplicación de escritorio que cuenta con una interfaz visual interactiva. La decisión de optar por una aplicación de escritorio se basó principalmente en el hecho de que tener un servidor que realice la transformación y atienda potencialmente muchos pedidos de transformación a la vez no es escalable y podría generar un cuello de botella. Además en caso de querer realizar una transformación de una base de datos local,

se deberían utilizar técnicas como conexiones mediante SSH para lograrlo, lo cual no es ideal.

Transformaciones La parte principal de la herramienta consiste en las transformaciones. Por eso, el conjunto de funciones encargadas de llevar a cabo la transformación se extrajo a una biblioteca para poder ser reutilizado y, si se quisiera, consumido por otra aplicación. Además, esta extracción brinda la posibilidad de realizar la transformación mediante línea de comandos utilizando la terminal de la computadora sin necesidad de una interfaz, si el usuario lo desea.

Esta biblioteca es el núcleo de la herramienta, donde se exponen funciones que conectan con ambas bases de datos. Por un lado, extrayendo la información de todas las tablas y tuplas de la base de datos relacional de PostgreSQL para luego insertarlas en colecciones de ArangoDB. Siguiendo el algoritmo de Direct Mapping, cada tabla en PostgreSQL corresponderá a una colección de ArangoDB y cada tupla de la tabla a un nodo dentro de esta colección. Mientras que las claves foráneas se transforman en entradas dentro de una colección denominada *edges* que representan las aristas del grafo. En la figura 4.1 se presenta un diagrama de alto nivel de los pasos realizados por la biblioteca para llevar a cabo la transformación.

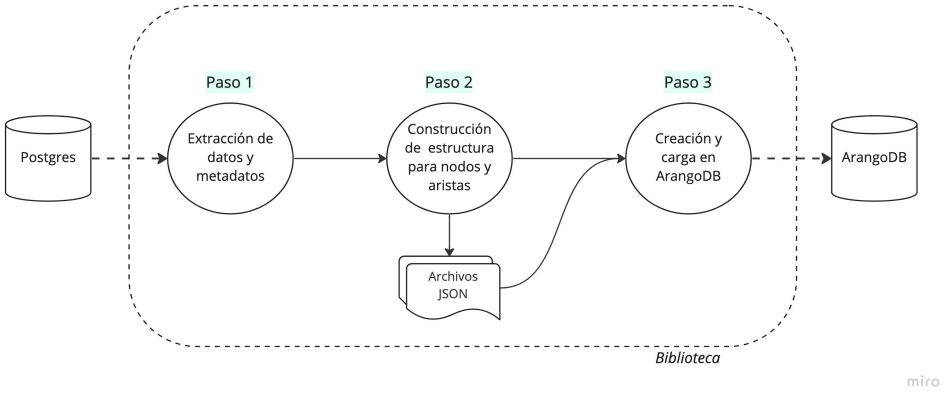

Figura 4.1: Diagrama de pasos realizados por la biblioteca para llevar a cabo la transformación.

De ahora en más, se menciona como base de datos de origen a la base de datos relacional PostgreSQL y base de datos destino a la base de datos ArangoDB.
Como se observa en la Figura 4.1 el primer paso involucra la conexión con la base de datos de origen y la extracción de información sobre sus tablas. Para

cada tabla se genera un objeto almacenado en memoria con la siguiente información:

1. *Name*: nombre de la tabla.
2. *Schema*: esquema al cual pertenece la tabla.
3. *Primary keys*: nombre de las columnas que pertenecen a la clave primaria de la tabla.
4. *Unique keys*: nombre de las columnas que pertenecen a la clave única de la tabla.
5. *Foreign keys*: nombre de las columnas que pertenecen a una clave foránea.
6. *Regular columns*: Nombre de las columnas que no forman parte de ninguna clave.

A su vez se identifica si las claves están compuestas por una o más columnas.

Luego, el segundo paso consiste en crear la estructura que dará forma a los nodos y aristas en ArangoDB, almacenando esta estructura en memoria mediante archivos JSON. Para la generación de estos archivos se deberá además realizar consultas a la base de datos de origen para extraer las tuplas de cada una de sus tablas. Estos archivos serán utilizados en el paso número tres para importar los datos en la base destino, obteniendo la transformación.

Creación de nodos La creación de una colección en la base destino mediante el uso de la biblioteca *arangojs*[1] se realiza a través de multiples archivos con formato JSON que contienen la información de cada uno de los nodos que pertenecerán a la colección que se desea crear. Primero se crea una colección en ArangoDB con el nombre correspondiente y luego se importan los datos mediante estos archivos. Dichos archivos se generan recorriendo las tablas de la base de origen, y cada tupla de la tabla relacional corresponde a un nodo en la colección creada en ArangoDB. Este recorrido que obtiene los datos de cada tupla se realiza utilizando paginado considerando el caso de bases de datos origen de alto volumen. En cada iteración se consulta una porción de los datos y para cada tupla de la tabla se genera el nodo correspondiente. ArangoDB maneja dos propiedades importantes para cada nodo:

- ***_key*** Identifica al nodo en la colección y es especificada por el usuario, en caso de que no se especifique se genera automáticamente un identificador único.

[1] https://www.npmjs.com/package/arangojs

- ***_id*** Generado automáticamente concatenando el nombre de la colección y el valor en *_key*.

Por ejemplo, si se considera la colección de nombre *Personas* y se inserta un nuevo nodo especificando *_key: 101* su atributo *_id* será *_id: personas/101*. Por lo tanto, para cada fila de una tabla de la base de datos de origen se creará un nodo en la base de destino siguiendo los siguientes puntos:

1. La clave primaria será representada en el atributo *_key*, y en caso de que la clave primaria sea compuesta, se concatenan los valores de los atributos que componen a la misma. En el caso de que la tabla no cuente con una clave primaria, se generará un identificador único que será usado en su lugar. En este último caso, se utilizó la biblioteca *uuid*[2] para generar el identificador único.

2. El resto de los atributos de la fila que no formen parte de ninguna clave serán agregados como propiedades del nodo.

3. Los atributos que corresponden a claves foráneas no serán agregados como parte de las propiedades del nodo sino que se agregaran a una colección especial denominada *edges*.

Para ejemplificar lo anterior, se utilizará la tabla *Personas* 4.1, la cual pertenece a un conjunto de tablas que siguen el modelo entidad-relación presentado en el capítulo 2. Esta tabla cuenta con una clave primaria formada por la columna “nombre“ y un atributo que representa la fecha de nacimiento de la persona.

nombre	fecha-nacimiento
Jane	26-05-1993
John	05-10-1995
Mark	03-11-1997

Tabla 4.1: Personas

lugar	tamaño
Canelones	pequeño
Montevideo	grande

Tabla 4.2: Lugares

persona	**lugar**	**fecha**
John	Canelones	13-05-2005
John	Canelones	10-10-2007
Mark	Canelones	12-05-2012
Mark	Montevideo	21-04-2013

Tabla 4.3: Visitas

[2] https://www.npmjs.com/package/uuid

Para transformar esta tabla primero se crea una nueva colección “Personas“ en la base ArangoDB. Luego se recorren todas las tuplas de la tabla comenzando con la primera. Para esta primera tupla se define el atributo *_key* del nodo como *_key: Jane* dado que es el valor de la clave primaria. En el hipotetico caso donde la clave primaria estuviera compuesta por dos atributos, por ejemplo, “Nombre“ y “Apellido“ y el valor del atributo “Apellido“ fuera “Smith“ este también pertenecería a la clave y, por lo tanto, el atributo *_key* sería su concatenación *_key: Jane-Smith*.
Luego, el atributo “fecha-nacimiento“ que no pertenece a ninguna clave es agregado como atributo del nodo. De esta manera el nodo resultante cuenta con la siguiente estructura:

```
1    {
2        "_id": "Personas/Jane"
3        "_key": "Jane"
4        "nombre": "Jane"
5        "fecha-nacimiento": "1993-05-26"
6    }
```

Esta estructura es luego guardada en un archivo JSON y se realiza el mismo procedimiento para las tuplas restantes.

Creación de aristas Como se mencionó anteriormente, los atributos de la tabla de origen que forman parte de claves foráneas formarán parte de una entrada en una colección especial que representa las aristas del grafo. Una tabla que posee un atributo que representa una clave foránea implica que cada nodo de dicha colección será relacionado mediante una arista con otro nodo perteneciente a otra (o la misma) colección (a la cual referencia la clave foránea). Por lo tanto, estas aristas se agregan a una colección denominada *edges* donde sus nodos contarán con solo dos atributos:

- ***_from*** Contiene el valor del atributo *_id* del nodo origen.
- ***_to*** Contiene el valor del atributo *_id* del nodo destino.

Si se observa la tabla Visitas 4.3 la misma cuenta con tres atributos donde todos pertenecen a la clave primaria de la tabla. Además, los atributos “persona“ y “lugar“ representan una clave foránea a las tablas *Personas* 4.1 y *Lugares* 4.2 respectivamente, siendo *Visitas* una tabla de relación.

Estas claves serán representadas como aristas, por lo cual para cada nodo generado a partir de la tabla *Visitas* se tendrán dos nuevas inserciones en en la colección *edges*. Para la primera tupla, se crea el nodo n que representa la visita de John a la ciudad de Canelones en la fecha 13-05-2005 y se generan dos aristas las cuales tienen al nodo n como origen:

- $e1$: arista que representa la clave foránea $persona$ y tiene como destino el nodo que representa a la persona de nombre John (*_key: John*).
- $e2$: arista que representa la clave foránea $lugar$ y tiene como destino el nodo que representa a el lugar Canelones (*_key: Canelones*).

Recordando que el atributo *_id* del nodo es generado automáticamente concatenando el nombre de la colección y el valor en el atributo *_key*, y además que el valor de *_key* será la concatenación de los atributos pertenecientes a la clave primaria de la tabla, el nodo que representa la clave foránea $lugar$ (correspondiente a la arista $e2$) tiene la siguiente estructura:

```
1    {
2        "_from": "Visitas/John-Canelones-13-05-2005"
3        "_to": "Lugares/Canelones"
4    }
```

Una pregunta que surgió al comienzo de la implementación fue si existiría alguna limitación a la hora de crear aristas que vinculan nodos que aún no fueron creados en ArangoDB. En caso de existir esta limitación, habría que recorrer las tablas de tal forma que elimine las dependencias a la hora de crear los nodos y las aristas ya que se debería asegurar de antemano que los nodos a relacionar existieran en la base.
Afortunadamente, ArangoDB no presenta ninguna limitación a la hora de hacer referencia a otros nodos en los atributos $\_from$ y $\_to$, sino que queda de nuestro lado la responsabilidad de que estos valores corresponden a nodos existentes en la base de datos.

Para la creación de aristas, se consideran los siguientes casos:

1. La clave foránea referencia a una clave primaria.
2. La clave foránea **no** referencia a una clave primaria. Sino que, por ejemplo, referencia a una *Unique Key*.

Se profundizará sobre el caso 2. ya que el caso 1. es el escenario que se ha contemplado hasta este momento y también es el más usual.

Caso 2. Al momento de generar el nodo perteneciente a la colección de *edges*, si la clave foránea referencia a una clave primaria es posible asumir que el identificador del nodo destino será la concatenación del nombre de la colección y el valor en la columna que representa la clave foránea, como se vio anteriormente ambos datos son conocidos. Sin embargo, si la clave foránea no referencia a una clave primaria el valor en esta columna no

representará ningún identificador de nodo destino y esto implica que será necesaria una consulta para obtener el identificador del nodo correspondiente a este valor.

Para ejemplificar y analizar este caso se modificarán las tablas *Lugares* 4.2 y *Visitas* 4.3. En la tabla *Lugares* 4.4 se agrega una columna id que representa una clave única y ahora la clave foránea $lugar$ de la tabla Visitas 4.5 referencia a esta clave única.

lugar	id	tamaño
Canelones	01	pequeño
Montevideo	02	grande

Tabla 4.4: Lugares

persona	**lugar**	**fecha**
John	01	13-05-2005
John	01	10-10-2007
Mark	01	12-05-2012
Mark	02	21-94-2013

Tabla 4.5: Visitas

El nodo generado que representa la primera fila de la tabla Lugares tendrá como valor de la propiedad *_key* la clave primaría $lugar$.

```
{
    "_id": "Lugares/Canelones"
    "_key": "Canelones"
    "lugar": "Canelones"
    "id": "01"
    "tamano": "small"
}
```

Por otro lado, el nodo que representa la primera fila de la tabla Visitas tendrá como valor de la propiedad *_key* la concatenación de los valores de su clave primaria:

```
{
    "_id": "Visitas/John-01-13-05-2005"
    "_key": "John-01-13-05-2005"
    "persona": "John"
    "fecha": "13-05-2005"
}
```

Cuando se recorre la tabla *Visitas*, luego de generar el nodo que corresponde a la primera fila y agregar el valor de todos sus atributos se analizan las claves foráneas. En este caso la clave foránea $lugar$ contiene el valor de la clave única y no el de la clave primaria de la tabla *Lugares*, por lo cual no es posible identificar el nodo destino con solo los datos de la tabla. El nodo que representa dicha relación tendrá la siguiente estructura:

```
1    {
2        "_from": "Visitas/John-01-13-05-2005"
3        "_to": "Lugares/Canelones"
4    }
```

Luego, es necesario realizar una consulta para poder obtener el valor del atributo *_key* del nodo. Esta consulta se realiza sobre la base de ArangoDB dado que el nodo destino ya habrá sido creado al momento de crear la arista. La misma es realizada al momento de la carga y sólo será necesaria en caso de que la clave foránea no haga referencia a una clave primaria.

4.2. Implementación

4.2.1. Biblioteca

Como se anticipó en el apartado anterior 4.1 la lógica principal (conjunto de funciones) que realiza las transformaciones de migración se extrajo en una biblioteca que fue implementada con *Node.js*[3]. La misma expone las siguientes funciones:

- *Init:* recibe las credenciales de las bases de datos de entrada y salida, verifica que ambas bases de datos existan, y establece una conexión con las mismas.
- *GetGraphPreview:* genera una previsualización del grafo saliente la cual consiste de nodos etiquetados que representan las tablas de la base de datos relacional y aristas que representan relaciones entre dichas tablas.
- *Migrate:* realiza la transformación desde la base de datos PostgreSQL a la base de datos ArangoDB especificadas en la función *Init*. Si el resultado es exitoso, al finalizar la ejecución se tendrá una base de datos en ArangoDB con los datos obtenidos de la base de datos relacional de PostgreSQL.

Para hacer posible la conexión con PostgreSQL se utiliza la biblioteca *pg* [4] mediante la cual se realizarán las consultas de SQL necesarias. Por otro

[3] https://nodejs.org/en/
[4] https://www.npmjs.com/package/pg

lado, para obtener la conexión con ArangoDB se utiliza *arangojs* [5] biblioteca que permite realizar la inserción de nuevas colecciones y nodos, así como consultas en AQL.

Luego de establecidas las conexiones a las bases de origen y destino, se extrae información general de las tablas de la base de entrada y es almacenada en memoria. Para esto se contemplan todos los esquemas existentes ignorando *information_schema* y *pg_catalog*. Tomando nuevamente como ejemplo la tabla *Personas* 4.1 que pertenece a la base de datos que sigue el modelo entidad-relación presentado en el capitulo 2, el objeto almacenado en memoria que contiene la meta-data de esta tabla tendrá la siguiente estructura:

```
{
      name: "Personas",
      schema: "public",
      regularColumns: [
       { name: "fecha-nacimiento" }
      ],
      primaryKey: [
       { name: "nombre" }
      ],
      uniqueKey: [],
      foreignKeys: []
}
```

Para este caso, las propiedades *uniqueKey* y *foreignKeys* están definidas como arreglos vacíos dado que la tabla no posee claves únicas ni claves foráneas, en caso de que si existieran atributos de este carácter el nombre del mismo estaría almacenado en estas propiedades.

La obtención de esta información se realiza a través de tres funciones internas de la biblioteca: *getPrimaryKey*, *getUniqueKeys* y *getForeignKeys*. Estas funciones realizan consultas SQL sobre la base de datos de origen, las mismas son presentadas en el apéndice A.1. Estas consultas utilizan la propiedad $constraint\_type$ para obtener los valores deseados.
La obtención de claves primarias y únicas es directa, pero por otro lado, la función *getForeignKeys* que obtiene las claves foráneas además verifica si la clave foránea hace referencia a una clave primaria o una clave única. Esta información es necesaria al momento de la importación de los datos en ArangoDB en casos como el que se analizó en el **Caso 2.** 4.1 para la creación de aristas, donde es necesario realizar una consulta AQL. Para entender mejor la estructura que tendrá el atributo *foreignKeys* almacenado

[5] https://www.npmjs.com/package/arangojs

en memoria, a continuación se presenta la estructura de las claves foráneas formadas por los atributos *persona* y *lugar* para la tabla *Visitas* 4.3 que pertenece a la base de datos que sigue el modelo entidad-relación presentado en el capitulo 2.

```
{
    foreignKeys: [
    {
        name: "persona",
        foreignTable: "Personas",
        pointsToPK: true,
        columns: [
            { name: "persona", foreignColumn: "nombre" }
        ]
    },
    {
        name: "lugar",
        foreignTable: "Lugares",
        pointsToPK: true,
        columns: [
            { name: "lugar", foreignColumn: "lugar" }
        ]
    }]
}
```

Para este caso, ambas claves foráneas referencian a una clave primaria, por lo cual el valor de la propiedad *pointsToPK* es verdadero y no es necesaria una consulta AQL para obtener el valor del identificador al momento de la carga en ArangoDB.

Luego de obtener toda la información necesaria sobre las tablas que componen la base de datos relacional, se procede a exportar a archivos JSON la estructura necesaria para la importación en ArangoDB. Se recorren las tablas y para cada una de ellas se obtienen las tuplas que pertenecen a la misma. Para cada tupla se realizan los siguientes pasos:

1. Se crea la estructura de un nodo donde su *key* será la composición de los atributos que pertenecen a la clave primaria.

2. Se agregan al nodo los atributos y sus valores que no corresponden a claves foráneas.

3. Se agrega en el conjunto $edges$ los atributos que pertenecen a claves foráneas.

a) Si el atributo apunta a una clave primaria se agregara en un archivo $edges$ que será importado directamente.

b) Si el atributo no apunta a una clave primaria se agregara en un archivo $query\_edges$ que requiere de una consulta a la base de datos ArangoDB antes de ser importado.

Luego, estos archivos son utilizados por la biblioteca arangojs para importar los datos en la base de datos ArangoDB, comenzando primero con los nodos, siguiendo por las aristas donde no se necesita realizar consultas a la base de datos de salida, y finalmente con las aristas que sí necesitan una consulta a la base de datos de salida. En caso de errores, los mismos son detectados y notificados por la biblioteca.

4.2.2. Aplicación

Como se mencionó en la sección 4.1 la lógica de las transformaciones se extrajo en una biblioteca que puede ser potencialmente consumida por diferentes aplicaciones. En esta propuesta hemos creado una aplicación de escritorio que consume esta biblioteca y tiene como funcionalidad principal la transformación. Para esto, se le presenta al usuario una interfaz donde ingresar la información de ambas bases de datos y luego, el usuario puede elegir entre ver una previsualización del grafo saliente o iniciar la transformación.

La aplicación cuenta con dos partes, una interfaz y un servidor quien será responsable de la conexión con la biblioteca de transformaciones. Un diagrama de la estructura se puede observar en la Figura 4.2.

Interfaz La interfaz de usuario fue implementada utilizando la librería de JavaScript *React.js*[6] y se incorporo *Electron.js* [7] para poder generar una aplicación de escritorio. Para los diseños hemos utilizado como inspiración un template de Figma[8], herramienta para generar diseños, de nombre *Multi-step Form Figma Template* [9].

Para el logo de la interfaz hemos utilizado MidJourney[10] inteligencia artificial para generar imágenes que está disponible para utilizar en su versión beta. Esta herramienta genera imágenes a partir de una lista de palabras. Dado que la propuesta transforma bases de datos en PostgreSQL hacia

[6] https://reactjs.org/
[7] https://www.electronjs.org/
[8] https://www.figma.com
[9] https://www.figma.com/community/file/1085642120902089828
[10] https://www.midjourney.com/

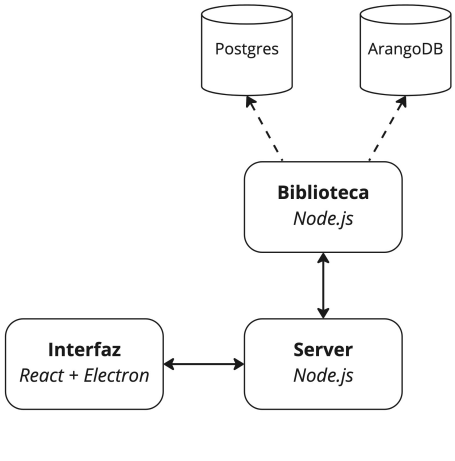

Figura 4.2: Estructura de la aplicación

ArangoDB hemos querido combinar los protagonistas de ambos logos. Para esto, hemos utilizado el siguiente comando en MidJourney.

```
1 \imagine blue elephant with green avocado, cute avocado, half
    avocado, 2D, logo
```

Este comando ha generado la imagen que se presenta en la Figura 4.3.

Figura 4.3: Logo de la aplicación generado con MidJourney

La aplicación cuenta con tres pantallas principales que constituyen los tres pasos que la aplicación posee.

1. Pantalla de ingreso de datos Figura 4.4. En donde el usuario debe ingresar la información necesaria para la conexión con ambas bases de datos. Esto es:

   a) Para la base de origen en PostgreSQL: Host, puerto, usuario, contraseña y nombre de la base de datos.

b) Para la base de destino en Arangodb: URL, usuario, contraseña y nombre de la base de datos.

2. Pantalla de previsualización Figura 4.5. En este paso el usuario puede optar por dos opciones: ver una previsualización de la base de datos de grafos o saltar este paso e iniciar la transformación. Cuando el usuario elige la primera opción, ver la previsualización del grafo saliente, se le presenta un grafo compuesto por nodos etiquetados que representan las tablas de la base de datos de origen y aristas sin etiquetar que representan relaciones entre estas tablas, como se observa en la Figura 4.6. De esta forma, el usuario puede tener una idea de cómo lucirá la base de datos saliente.

3. Pantalla de transformación Figura 4.7. Cuando el usuario inicia la transformación, se le presenta en pantalla un contenedor cuyo diseño imita a el de una consola donde se despliega información relevante, como la cantidad de tablas a transformar, las tablas que ya fueron transformadas, y algún eventual error que pueda llegar a ocurrir durante la transformación. Luego de que la transformación finaliza, se le presenta al usuario un mensaje indicándoselo.

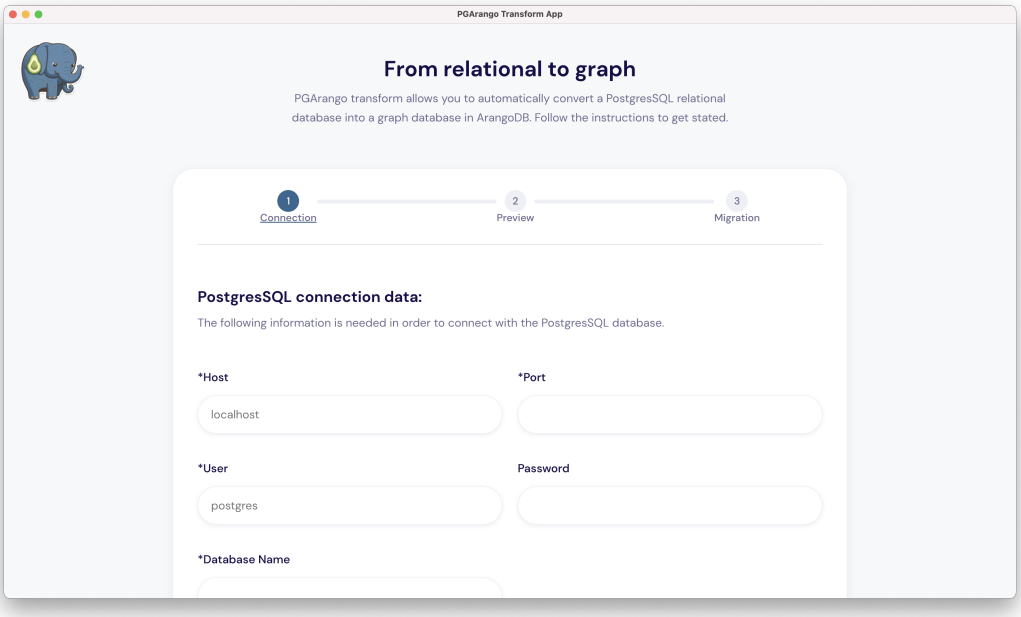

Figura 4.4: Aplicación: Pantalla de ingreso de datos necesarios para la conexión con ambas bases de datos.

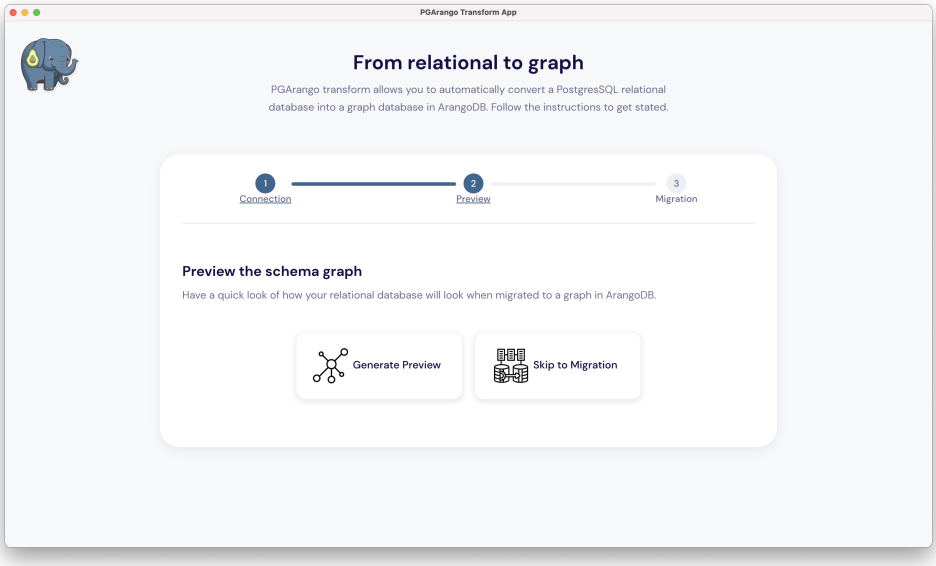

Figura 4.5: Aplicación: Opciones de previsualización del grafo o saltar a la transformación.

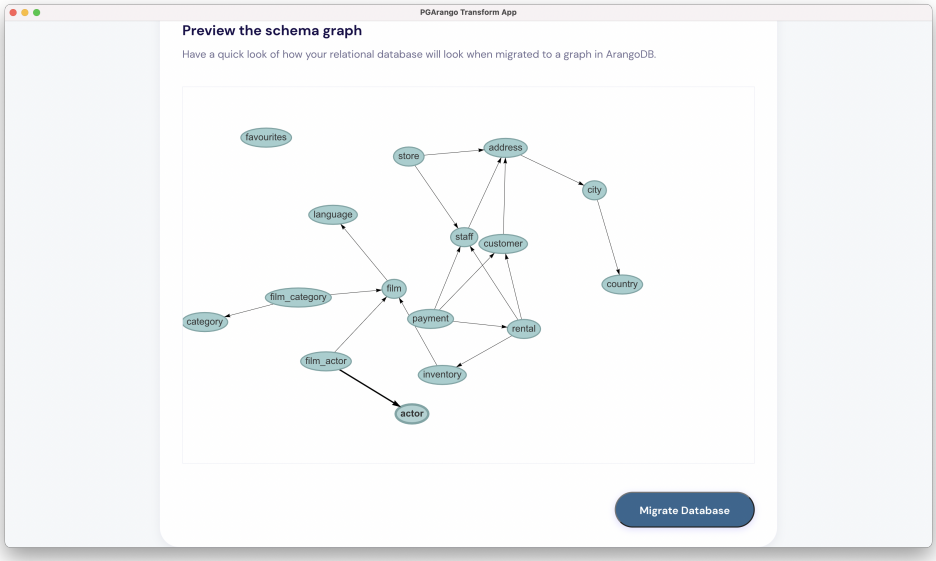

Figura 4.6: Aplicación: Opción de previsualización del grafo elegida.

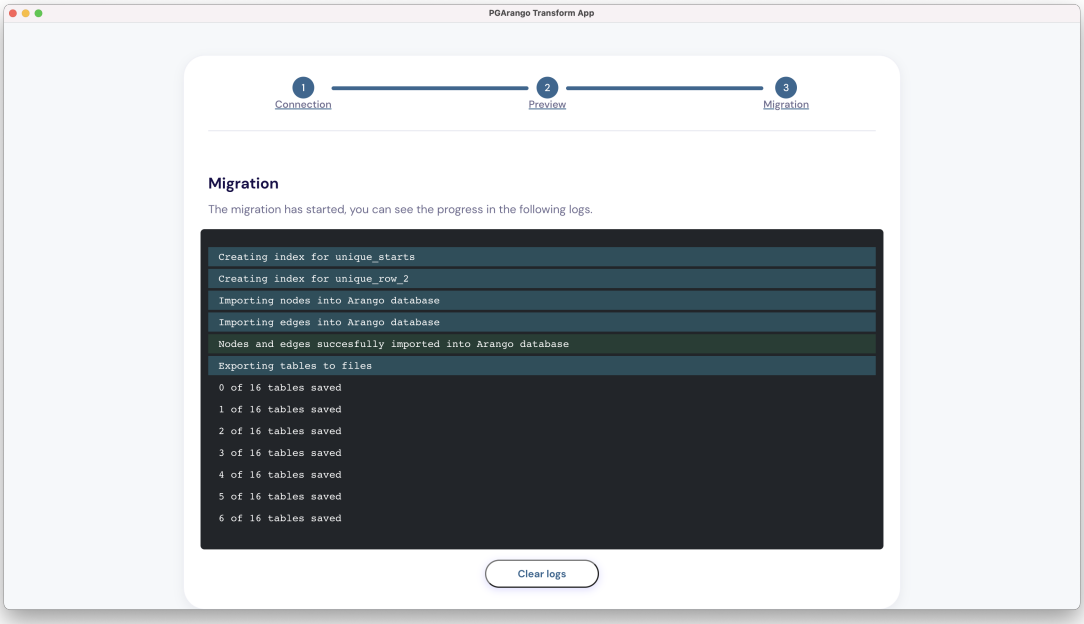

Figura 4.7: Aplicación: Pantalla de transformación con mensajes informativos.

Los mensajes informativos son una parte importante en la experiencia de usuario, dado que la transformación puede durar varios minutos dependiendo del tamaño de la base de datos relacional. Por esta razón, se decidió implementar una conexión mediante web sockets entre servidor e interfaz utilizando *socket.io*[11] lo cual permite el intercambio de mensajes en ambas direcciones. La interfaz realizará pedidos al servidor de inicialización y transformación y el servidor le provee a la interfaz el estado de los pedidos, los cuales serán desplegados al usuario.

Servidor El servidor fue implementado en *Node.js*[12] y es el encargado de consumir las funciones expuestas por la biblioteca, listadas en la sección 4.2.1. No es quien realiza las transformaciones, sino que tiene el rol de intermediario entre la interfaz de usuario y la biblioteca.
Teniendo en cuenta que el tiempo de demora ante una transformación dependerá del tamaño de la base de datos y que el recorrido de las tablas y filas de una tabla en PostgreSQL será lineal se optó por la utilización de *web sockets*. Este concepto permitirá enriquecer la experiencia de usuario desplegando información del estado de la transformación y evitando así dejar

[11] https://socket.io/
[12] https://nodejs.org/en/

al usuario en espera hasta que el proceso termine.
El servidor es el encargado de la creación y exposición del socket que utiliza la interfaz. La biblioteca cuenta con la posibilidad de recibir una función *notify*, la cual (en caso de ser provista) internamente es invocada con mensajes que reflejan el estado de la transformación. Esto permite que el servidor genere una función para envío de mensajes informativos a la interfaz utilizando el socket que ha creado.

El código fuente tanto de la biblioteca como de la aplicación se encuentran disponibles en Github bajo los nombres *arango-migration-app*[13] y *pg-arango-transform*[14].

[13] https://github.com/elizabethlofredo/pg-arango-migration-app
[14] https://github.com/augusto1024/pg-arango-transform

Capítulo 5

Experimentación

En esta sección se detallarán diferentes pruebas y experimentos realizados sobre la aplicación implementada. Por un lado se realizaron pruebas de completitud con el fin de verificar que los resultados luego de aplicado el algoritmo fueran los correctos y el grafo resultante representará correctamente la base de datos de entrada, sin pérdida de información. Luego, se detallan las pruebas realizadas para verificar el correcto funcionamiento de la aplicación utilizando diferentes tipos de bases de datos, para esto además de bases de datos de prueba de porte pequeño y mediano se utilizó el estándar de comparación TPC-H, el cual se ha mencionado en la sección 3.4 y fue utilizado para evaluar los enfoques de trabajos relacionados con el objetivo de este proyecto.

5.1. Pruebas de completitud

Las pruebas de completitud se encuentran ubicadas en la biblioteca que contiene la lógica principal de las transformaciones. Para ejecutar estas pruebas es necesario brindar la información de la base de datos de entrada y la base de datos de salida a través de un archivo de variables de entorno (.env) dado que estas pruebas utilizan las funciones expuestas por la biblioteca. Para ejecutar estas pruebas es necesario utilizar la terminal dado que son ejecutadas mediante línea de comando.
Las pruebas implementadas realizan los siguientes chequeos:

1. El número de colecciones en ArangoDB debe ser igual al número de tablas que componen la base de datos de PostgreSQL.

2. El número de filas en una tabla debe ser igual al número de nodos en la colección que representa dicha tabla.

Estas pruebas realizan una serie de consultas a las bases de ArangoDB y PostgreSQL con el fin de comparar los datos en la base de entrada con los datos de la base de datos obtenida luego de la migración. Como se mencionó en la sección 4.2.1 los esquemas *information_schema* y *pg_catalog* son ignorados.

Para ejecutar estas pruebas se utilizaron diferentes bases de datos. Por supuesto, se ha utilizado la base de datos de ejemplo mencionada a lo largo de este trabajo que sigue el modelo entidad-relación presentado en 2.1. Pero además, se ha utilizado la base de datos DVDRental [1] la cual contiene 15 tablas, entre ellas tablas de relación y una cantidad considerable de claves foráneas para validar la correcta conformación de las aristas. Esta base de datos también contiene algunas vistas. Por otro lado, también se utilizó la base de datos Adventureworks [2], una base de datos de mayor porte que contiene 11 esquemas distintos y 71 tablas.

Durante la ejecución de las pruebas sobre estas bases de datos se encontraron algunos errores y casos que no habían sido considerados a la hora de la implementación, a continuación se detallan los mismos.

Errores encontrados Uno de los primeros errores que se encontró durante la ejecución fue la pérdida de datos en la transformación donde algunas filas de las tablas relacionales no eran transformadas en nodos. Luego de investigar se encontraron algunas restricciones en el formato que debe tener el atributo *_key* en los nodos de ArangoDB donde no se permiten algunos caracteres especiales ni espacios en blanco [3]. Para solucionar esto, se implementó una función auxiliar que remueve los caracteres que no se encuentran dentro de los permitidos mediante la siguiente expresión regular: `[^A-Za-z0-9_\-:.@()+,=;$!*'%]`.
El uso de esta expresión permite que los nodos puedan ser creados en ArangoDB dado que la *_key* siempre tendrá un valor válido.

Otra discrepancia encontrada en los datos luego de la transformación está relacionada a los atributos de la base de datos relacional que contienen valores de tipo fecha y hora. En este caso, el error fue detectado mediante un atributo de fecha donde al momento de obtener los valores de dicho atributo en la tabla de PostgreSQL, la biblioteca *pg* que se ha utilizado para la conexión con la base de datos retorna por defecto el $timezone$ junto al valor del atributo, siendo en este caso la zona horaria local como se puede

[1] PostgreSQL Sample Database - https://www.postgresqltutorial.com/postgresql-getting-started/postgresql-sample-database (Último acceso: 4 de Marzo, 2023)

[2] AdventureWorks sample databases - https://learn.microsoft.com/en-us/sql/samples/adventureworks-install-configure (Último acceso: 4 de Marzo, 2023)

[3] https://www.arangodb.com/docs/stable/data-modeling-naming-conventions-document-keys.html

observar a continuación:

```
{
    "_key": "13-2-1-Thu Dec 23 2010 00:00:00 GMT-0200 (
        Uruguay Summer Time)"
}
```

Para solucionarlo, se tuvo que configurar la biblioteca mediante la función $setTypeParser$ la cual recibe un código que permite remover el $timezone$ de el resultado obtenido, aplicando esta configuración para valores que incluyan fecha y hora.

Luego, un escenario que no se había contemplado a la hora de la implementación de la aplicación fue la existencia de vistas. Las vistas, que son tablas virtuales, contienen el conjunto de resultados de una consulta que son almacenados y presentados en forma de tabla.
A primera vista estas tablas no deberían generar ningún tipo de problema dado que su estructura es igual a la de otras tablas y el algoritmo debería comportarse correctamente, pero el factor que no se había contemplado hasta el momento de la ejecución de las pruebas es que las vistas pueden tener el mismo nombre que otras tablas, lo cual presenta un error a la hora de migrar los datos. Luego de identificado este error, se tomó la decisión de ignorar las vistas durante la transformación, dado que estas son generadas por el usuario y no son datos esenciales de la base de datos, sino que pueden volver a ser generadas en ArangoDB luego de realizada la transformación si el usuario las necesita.

5.2. Pruebas de corrección

Para llevar a cabo las pruebas de corrección se ha utilizado la base de datos del estándar de comparación de TPC-H y su conjunto de consultas, estándar que ya ha sido presentado en la sección 3.4 con el fin de evaluar los enfoques y herramientas presentados en dicha sección. Siendo esta una base de datos que cuenta con un gran volumen de datos, es interesante evaluar cómo se comporta la propuesta implementada ante la transformación de la misma y luego de ejecutada la transformación corroborar si las consultas sobre la base de datos de origen (PostgreSQL) y de destino (ArangoDB) coinciden, es decir, se mantiene la información correctamente luego de ejecutada la transformación.

La primera puesta a prueba consistió en la ejecución de la transformación implementada sobre la base de datos de TPC-H. Para esto, primero fue

necesario construir esta base de datos utilizando un generador[4] que permite elegir la escala del volumen de datos a utilizar, donde se consideró que la primera escala (TPC-H 1) era adecuada para el inicio de las pruebas. El resultado de la transformación mediante la herramienta implementada fue exitoso y la herramienta logró culminar la migración completamente.

Durante la implementación, se siguieron algunas estrategias que ayudaran al rendimiento de la herramienta como lo son la paralelización de consultas a la base de datos de entrada, paralelización de escritura y fragmentación de archivos. Pero, el rendimiento y eficiencia de la transformación no fue un aspecto involucrado en los objetivos principales. A pesar de esto, es de interés conocer su rendimiento, a continuación se presentan el tiempo total y espacio de datos almacenados en memoria luego de ejecutar el algoritmo en una computadora con las siguientes especificaciones: CPU: AMD Ryzen 7 5800 X 4.7 GHz y RAM: 32GB DDR4 3200 MHz.

- Tiempo total de la transformación: 20 minutos y 46 segundos.
- Datos almacenados en memoria por la herramienta: 3.7GB de datos.

Transformación de consultas Además de un conjunto de datos, TPC-H provee un conjunto de consultas SQL que es posible ejecutar sobre los datos para realizar validaciones, más específicamente, el total es de 22 consultas. Éstas consultas no son aleatorias, sino que tienen sentido para la realidad en la que se basó la construcción de la base de datos.
Entonces, para evaluar la corrección de la transformación será necesario realizar estas consultas en la base de datos de origen y destino. Esto es directo para la base de datos de origen en PostgreSQL dado que las consultas disponibles son consultas SQL, pero no es posible utilizarlas para la base de datos de destino en ArangoDB dado que el lenguaje de consultas utilizado es AQL. Existen algunos recursos donde se presentan estas consultas AQL, puntualmente hemos tomado como referencia el trabajo *"On the Performance Evaluation of Big Data Systems"* (Pirzadeh, 2015), pero estas consultas son compatibles con versiones anteriores de ArangoDB y no funcionan en las últimas versiones ya que el lenguaje ha avanzado considerablemente desde sus inicios. Por lo tanto, es necesario traducir estas consultas SQL en consultas AQL manualmente para lograr verificar que los resultados de ambas bases de datos coinciden.
Luego de adquirir conocimientos sobre el lenguaje AQL, se han traducido las primeras tres consultas. Se consideró que este era un número representativo para la validación que se buscaba. En el apéndice B.1 se presentan dichas consultas en ambos lenguajes, la consulta original SQL y la traducción obtenida en AQL.

[4] https://github.com/jfeser/tpch-dbgen

Luego de realizadas las consultas en ambas bases de datos se observa que: PostgreSQL retorna el resultado en un formato tabular y ArangoDB retorna el resultado en formato JSON. Entonces, para poder comparar los resultados primero es necesario unificar la salida a un mismo formato. Para esto, se convirtieron los resultados de PostgreSQL a formato JSON y se utilizó la herramienta JSON Diff[5] para realizar la comparación. Dicha comparación resultó satisfactoria ya que los resultados de las consultas son iguales, con la sutil excepción de diferencias en la parte final decimal de algunos valores. Ésta diferencia puede deberse a distintos factores, como por ejemplo, la diferencia de precisión en los números flotantes utilizados en PostgreSQL y ArangoDB.
Por tanto, se concluye que además de la transformación exitosa, los datos contenidos en la base de datos de grafos obtenida son correctos y pueden ser consultados de manera correcta al igual que en la base de datos relacional de entrada.

[5] https://www.jsondiff.com/

Capítulo 6

Conclusiones y trabajo futuro

En este capítulo se presentarán las conclusiones de este trabajo, evaluando los resultados obtenidos con los objetivos que se plantearon en un principio. También, se presenta una posible línea de trabajo futuro teniendo en cuenta algunos aspectos que fueron despriorizados durante la construcción de la propuesta y que pueden agregarle valor a la misma.

6.1. Conclusiones del proyecto

Durante la primer parte del proyecto se realizó una recolección de información sobre las migraciones de bases de datos relacionales a bases de datos de grafos, obteniendo así tres principales enfoques a evaluar: por un lado la propuesta *"R2PG-DM: A direct mapping from relational databases to property graphs"* (R.-A. Stoica, 2019) enfocada en mapeo directo; continuando con *"Converting Relational to Graph Databases"* (De Virgilio y cols., 2013) que presenta una metodología propia; y finalmente la herramienta de Neo4j ETL Tool.
La evaluación realizada sobre los tres enfoques demostró que los tres contaban con diferentes desventajas, errores y ninguno cumplía exitosamente con todos los aspectos que se buscaban cubrir: implementación e interfaz de usuario disponible; ejecución exitosa en bases de datos de mediano y alto porte; y finalmente corrección y completitud en la transformación de los datos. Si bien la herramienta ETL Tool de Neo4j parecía la más completa dado que esta respaldada por la compañía Neo4j, Inc. de gran prestigio en la industria, la misma no tuvo una ejecución exitosa tanto en bases de datos pequeñas ni tampoco en bases de datos de gran porte como la que utiliza el estándar TPC-H, y además en su interfaz se han encontrado algunos puntos

de mejora como la falta de mensajes informativos sobre el estado del proceso hacia el usuario.

Teniendo en mente los resultados de la investigación y evaluación de las herramientas existentes, la solución propuesta buscó como objetivo principal cumplir estos aspectos que fueron detectados como fallidos en los enfoques evaluados: enfocarse en la corrección y completitud de la transformación; ser robusta frente a bases de datos de cualquier porte; y brindar una mejor experiencia de usuario. El resultado fue una aplicación de escritorio que cumple dichos objetivos, en la Figura 6.1 se puede observar la misma tabla comparativa presentada en la sección 3.5 donde ahora se incluyen los resultados obtenidos para la propuesta de este trabajo.

	R2PG-DM: A direct mapping from relational databases to property graphs	**R2G**: Converting Relational to Graph Databases	**Neo4j ETL Tool**	**Solución Propuesta**
Implementación disponible	**Si**	**No**	**Si**	**Si**
Interfaz disponible	**No**	**No**	**Si**	**Si**
Ejecución en base de datos de pequeño porte	Ejecución exitosa	Ejecución manual fallida	Ejecución termina pero el grafo resultante presenta errores.	Ejecución exitosa
Ejecución en base de datos de gran porte TPC-H	Ejecución falla con errores.	No fue posible evaluarlo dado que no se cuenta con una implementación.	Ejecución no termina.	Ejecución exitosa

miro

Figura 6.1: Tabla comparativa que incluye la solución propuesta.

Para corroborar su correcto funcionamiento, se realizaron pruebas de corrección, utilizando el estándar TPC-H, las cuales fueron exitosas y pruebas de completitud utilizando la base de origen y la base destino generada. Por otro lado, dado que el proceso de transformación puede durar varios minutos se utilizaron WebSockets para obtener una mejor experiencia de usuario con mensajes informativos sobre el estado de la transformación. La propuesta además permite migrar más de un esquema dentro de una misma base de datos y no utiliza tablas intermedias para realizar la transformación, pero si utiliza almacenamiento en memoria.
Teniendo en cuenta que el rendimiento no fue un aspecto priorizado durante la implementación de la misma, el rendimiento de la aplicación a la hora de migrar la base de datos de TPC-H 1 fue relativamente bueno, realizando la transformación en aproximadamente 20 minutos.

6.2. Trabajo a futuro

Si bien la construcción y desempeño de la herramienta fue satisfactoria, hay algunos aspectos que pueden ser mejorados en el futuro.

Primero, el hecho de solo poder elegir PostgreSQL como único manejador de base de datos relacional es bastante limitante, por lo tanto, realizar una adaptación para soportar otros DBMS relacionales sería una mejora considerable. Por otro lado, el algoritmo implementado potencialmente serviría para muchos otros manejadores de bases de datos de grafos, como Neo4J que fue mencionado en este trabajo. Dicho esto, además de soportar distintos DBMS relacionales de entrada, también sería una mejora significativa soportar distintos DBMS de grafos de salida, lo cual permitiría un uso más extensivo de la herramienta.

Luego de finalizada la migración, los nodos creados en la base de datos de salida son etiquetados con el nombre de la tabla relacional a la cual pertenecen. Una posible mejora en este aspecto sería darle la posibilidad al usuario de cambiar las etiquetas de los nodos antes de que sean creados.
En la misma línea, todas las aristas de la base de datos saliente son creadas en la misma colección. Sin embargo, ArangoDB permite crear varias colecciones de aristas y poder elegir qué aristas (y también nodos) conforman el grafo de salida. Por lo tanto, sería interesante poder darle al usuario la posibilidad de agrupar aristas bajo algún criterio que consideren pertinente. Además, se podría llegar a brindar la elección de crear varios grafos en la misma base de datos, ya que ArangoDB lo permite debido a que el concepto de grafo que manejan es una abstracción de los datos existentes en la base de datos.

Apéndice A

Apéndice A

A.1. Consultas SQL sobre PostgreSQL

A.1.1. Tablas que componen la base de datos

Consulta que obtiene el nombre de todas las tablas y columnas que forman parte de la base relacional excluyendo los esquemas $information\_schema$ y $pg\_catalog$. Además se ignoran las vistas bajo la condición de que el valor $table\_type$ sea *BASE TABLE*.

```
select
  ic.table_name as "tableName",
  ic.column_name as "columnName",
  ic.table_schema as "tableSchema"
from information_schema.columns as ic join
      information_schema.tables as it
      on ic.table_name = it.table_name
      and ic.table_schema = it.table_schema
where ic.table_schema <> 'information_schema'
  and  ic.table_schema <> 'pg_catalog'
  and it.table_type = 'BASE TABLE'
```

A.1.2. Claves primarias

Dado el nombre de una tabla relacional, obtiene el nombre de los atributos que componen su clave primaria.

```
select
    ccu.column_name AS "name"
from
    information_schema.table_constraints as tc
```

```
    join information_schema.constraint_column_usage as ccu
    on ccu.constraint_name = tc.constraint_name
    and ccu.table_schema = tc.table_schema
where tc.constraint_type = 'PRIMARY KEY' and tc.table_name=$1;
```

A.1.3. Claves foráneas

Dado el nombre de una tabla relacional, obtiene el nombre de los atributos que componen sus claves foráneas.

```
select
    kcu.column_name as "columnName",
    tc.constraint_name as "constraintName"
from
    information_schema.table_constraints tc
    join information_schema.key_column_usage kcu
    on kcu.constraint_name = tc.constraint_name
where tc.constraint_type = 'FOREIGN KEY' and tc.table_name = $1;
```

A.1.4. Claves únicas

Dado el nombre de una tabla relacional, obtiene el nombre de los atributos que componen su clave única.

```
select
    kcu.column_name as "columnName",
    tc.constraint_name as "constraintName"
from
    information_schema.table_constraints tc
    join information_schema.key_column_usage kcu
    on kcu.constraint_name = tc.constraint_name
where tc.constraint_type = 'UNIQUE' and tc.table_name = $1;
```

A.1.5. Tablas de claves foráneas

Dado un conjunto de claves foráneas, obtiene la información correspondiente a las tablas a las que pertenecen y a las que referencia:

```
select
    kcu.column_name as "columnName",
    ccu.column_name as "foreignColumnName",
    ccu.constraint_name as "constraintName",
    ccu.table_name as "foreignTableName"
from information_schema.constraint_column_usage ccu join
    information_schema.key_column_usage kcu
```

```
    on  kcu.constraint_name = ccu.constraint_name
where ccu.constraint_name = ANY($1)
```

A su vez, para la obtención de la clave primaria de una tabla a la que una clave foránea hace referencia es utilizada la siguiente consulta:

```
select
    ccu.column_name AS "primaryKey",
    tc.table_name as "tableName"
from
    information_schema.table_constraints AS tc
    join information_schema.constraint_column_usage AS ccu
    on ccu.constraint_name = tc.constraint_name
    and ccu.table_schema = tc.table_schema
where tc.constraint_type = 'PRIMARY KEY' and
    tc.table_name=ANY($1)‘
```

Apéndice B

Apéndice B

B.1. Consultas en SQL y AQL

B.1.1. Q1 - Consulta de informe de reporte de precios

Esta consulta provee un informe de los reportes de precio para todos los productos enviados en una fecha dada. Entre la fecha elegida y la máxima fecha existente en la base de datos debe haber un intervalo de entre 60 y 120 días.

Consulta en SQL

```
select
    l_returnflag,
    l_linestatus,
    sum(l_quantity) as sum_qty,
    sum(l_extendedprice) as sum_base_price,
    sum(l_extendedprice * (1-l_discount)) as sum_disc_price,
    sum(l_extendedprice * (1-l_discount) * (1+l_tax)) as sum_charge,
    avg(l_quantity) as avg_qty,
    avg(l_extendedprice) as avg_price,
    avg(l_discount) as avg_disc,
    count(*) as count_order
from
        lineitem
where
    l_shipdate <= '12-01-1998'::date  - interval '90 day'
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus;
```

Consulta en AQL

```
for line in lineitem
filter
    date_iso8601(line.l_shipdate) <= date_add('1998-12-01', -90, 'day')
collect
    returnflag = line.l_returnflag,
    linestatus = line.l_linestatus
aggregate
    sumqty = sum(to_number(line.l_quantity)),
    sum_base_price = sum(to_number(line.l_extendedprice)),
    sum_disc_price = sum(to_number(line.l_extendedprice)
        * (1-to_number(line.l_discount))),
    sum_charge = sum(to_number(line.l_extendedprice)
        * (1-to_number(line.l_discount)) * (1+to_number(line.l_tax))),
    avg_qty = avg(to_number(line.l_quantity)),
    avg_price = avg(to_number(line.l_extendedprice)),
    avg_disc = avg(to_number(line.l_discount)),
    count = length(1)
sort returnflag, linestatus
return {
    returnflag,
    linestatus,
    sumqty,
    sum_base_price,
    sum_disc_price,
    sum_charge,
    avg_qty,
    avg_price,
    avg_disc,
    count
}https://www.jsondiff.com/
```

B.1.2. Consulta del proveedor de menor costo

En ésta consulta se encuentra el proveedor con el menor costo para un producto de determinado tipo y tamaño y en una región específica. Si existen varios vendedores cuyo precio (el mínimo) es el mismo, se devuelven los productos de los vendedores que tengan los 100 balances bancarios más altos. Para cada vendedor, la consulta retorna el balance bancario, nombre, nacionalidad, dirección, teléfono y comentario de información; así como también el número y fabricante del producto.

Consulta en SQL

```
select
    s_acctbal,
    s_name,
    n_name,
    p_partkey,
    p_mfgr,
    s_address,
    s_phone,
    s_comment
from part, supplier, partsupp, nation, region
where
    p_partkey = ps_partkey
    and s_suppkey = ps_suppkey
    and p_size = 15
    and p_type LIKE '%BRASS'
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'EUROPE'
    and ps_supplycost = (
        select
         min(ps_supplycost)
        from
         partsupp,
         supplier,
         nation,
         region
        where
         p_partkey = ps_partkey
         and s_suppkey = ps_suppkey
         and s_nationkey = n_nationkey
         and n_regionkey = r_regionkey
         and r_name = 'EUROPE'
    )
order by s_acctbal DESC, n_name, s_name, p_partkey
limit 100;
```

Consulta en AQL

```
let parts = (
    for p in part
    filter
        p.p_size == 15 and p.p_type LIKE '%BRASS%'
    return p
)
```

```
let partSupps = (
    for p in parts
    for ps in 1..1
        inbound p
        edges
    return {
        _id: ps._id,
        _part_id: p._id,
        p_partkey: p.p_partkey,
        p_mfgr: p.p_mfgr,
        ps_supplycost: to_number(ps.ps_supplycost),
    }
)

let partsBySupplier = (
    for ps in partSupps
    for s in 1..1
        outbound ps
        edges
    filter
        s.s_suppkey != null
    return {
        _id: s._id,
        _part_id: ps._part_id,
        p_partkey: ps.p_partkey,
        p_mfgr: ps.p_mfgr,
        ps_supplycost: ps.ps_supplycost,
        s_name: s.s_name,
        s_acctbal: to_number(s.s_acctbal),
        s_address: s.s_address,
        s_phone: s.s_phone,
        s_comment: s.s_comment
    }
)

let partsByNation = (
    for s in partsBySupplier
    for n in 1..1
        outbound s
        edges
    return {
        _id: n._id,
```

```
        _part_id: s._part_id,
        p_partkey: s.p_partkey,
        p_mfgr: s.p_mfgr,
        ps_supplycost: s.ps_supplycost,
        s_name: s.s_name,
        s_acctbal: s.s_acctbal,
        s_address: s.s_address,
        s_phone: s.s_phone,
        s_comment: s.s_comment,
        n_name: n.n_name
    }
)

let partsByRegion = (
    for n in partsByNation
    for r in 1..1
        outbound n
        edges
    filter
        r.r_name LIKE '%EUROPE%'
    return {
        _id: n._part_id,
        p_partkey: n.p_partkey,
        p_mfgr: n.p_mfgr,
        ps_supplycost: n.ps_supplycost,
        s_name: n.s_name,
        s_acctbal: n.s_acctbal,
        s_address: n.s_address,
        s_phone: n.s_phone,
        s_comment: n.s_comment,
        n_name: n.n_name,
        r_rname: r.r_name
    }
)

let minCostParts = (
    for p in partsByRegion
    collect
        p_partkey = p.p_partkey
    aggregate
        ps_supplycost = min(to_number(p.ps_supplycost))
    return {
        p_partkey,
```

```
        ps_supplycost
    }
)

for mp in minCostParts
for p in partsByRegion
filter
    mp.p_partkey == p.p_partkey and mp.ps_supplycost == p.ps_supplycost
sort
    p.s_acctbal desc, p.n_name, p.s_name, p.p_partkey
limit 100
return {
    s_acctbal: p.s_acctbal,
    s_name: p.s_name,
    n_name: p.n_name,
    p_partkey: p.p_partkey,
    p_mfgr: p.p_mfgr,
    s_address: p.s_address,
    s_phone: p.s_phone,
    s_comment: p.s_comment
}
```

B.1.3. Consulta de la prioridad de envío

En ésta consulta se retorna la prioridad de envío y la potencial ganancia de las órdenes que tengan la mayor recaudación entre las órdenes que no hayan sido enviadas antes de una fecha determinada. Las órdenes se listan en orden decreciente según la ganancia. Si hay más de 10 órdenes que aún no han sido enviadas, se retornan las 10 primeras con mayor ganancia.

Consulta en SQL

```
select
    l_orderkey,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    o_orderdate,
    o_shippriority
from customer, orders, lineitem
where
    c_mktsegment = 'building'
    and c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate < '3-15-1995'::date
    and l_shipdate > '3-15-1995'::date
```

```
group by l_orderkey, o_orderdate, o_shippriority
order by revenue desc,  o_orderdate
limit 10;
```

Consulta en AQL

```
let filteredCustomers = (
    for c in customer
    filter
        c.c_mktsegment LIKE 'BUILDING%'
    return c
)

let filteredOrders = (
    for c in filteredCustomers
    for o in 1..1
        inbound c
        edges
    filter date_iso8601(o.o_orderdate) < date_iso8601('1995-3-15')
    return o
)

for o in filteredOrders
for l in 1..1
    inbound o
    edges
filter date_iso8601(l.l_shipdate) > date_iso8601('1995-3-15')
collect
    o_orderkey = o.o_orderkey,
    o_orderdate = o.o_orderdate,
    o_shippriority = o.o_shippriority
aggregate
    revenue = sum(to_number(l.l_extendedprice) * (1 - to_number(l.l_discount)))
sort revenue desc, o_orderdate
limit 10
return {
    o_orderkey,
    revenue,
    o_orderdate,
    o_shippriority
}
```

Referencias

Angles, R. (2018). The property graph database model. En D. Olteanu y B. Poblete (Eds.), *Proceedings of the 12th alberto mendelzon international workshop on foundations of data management, cali, colombia, may 21-25, 2018* (Vol. 2100). CEUR-WS.org. Descargado de http://ceur-ws.org/Vol-2100/paper26.pdf

ArangoDB. (2020, April). *What is a multi-model database and why use it?* (Inf. Téc.).

Cheng, Y., Ding, P., Wang, T., Lu, W., y Du, X. (2019). Which category is better: benchmarking relational and graph database management systems. *Data Science and Engineering*, *4*, 309–322.

De Virgilio, R., Maccioni, A., y Torlone, R. (2013). Converting relational to graph databases. En *First international workshop on graph data management experiences and systems*. New York, NY, USA: Association for Computing Machinery. Descargado de https://doi.org/10.1145/2484425.2484426 doi: 10.1145/2484425.2484426

Elmasri, R., y Navathe, S. (2010). *Fundamentals of database systems* (6th ed.). USA: Addison-Wesley Publishing Company.

Fernandes, P. (2016). *Introducing the newest rdbms-to-neo4j etl tool.* https://neo4j.com/blog/rdbms-neo4j-etl-tool/.

Pirzadeh, P. (2015). *On the performance evaluation of big data systems* (Tesis Doctoral no publicada). UC Irvine.

Stoica, R., Fletcher, G. H., y Sequeda, J. F. (2019). On directly mapping relational databases to property graphs. En *Amw*. Descargado de https://ceur-ws.org/Vol-2369/short06.pdf

Stoica, R.-A. (2019). *R2pg-dm: a direct mapping from relational databases to property graphs* (Tesis Doctoral, Master's thesis, Eindhoven University of Technology). Descargado de https://pure.tue.nl/ws/portalfiles/portal/139492780/R2PG_DM_Final.R.A.Stoica.pdf