Universidad de La Habana
Facultad de Matemática y Computación

Enfoques Zero-Shot para la Extracción de Conocimiento a partir de Lenguaje Natural

Autor: **Rolando Sánchez Ramos**

Tutor: **Dr. Alejandro Piad Morffis**

Trabajo de Diploma
presentado en opción al título de
Licenciado en Ciencias de la Computación

Noviembre de 2023

github.com/rolysr/nl2ql

Agradecimientos

Opinión del tutor

Dr. Alejandro Piad Morffis
Facultad de Matemática y Computación
Universidad de la Habana
Noviembre, 2023

Resumen

Esta tesis se centra en abordar la complejidad inherente a la consulta de bases de datos en forma de grafo, como Neo4J. Estas bases de datos a menudo requieren un conocimiento especializado en lenguajes de consulta, lo que limita su accesibilidad a un grupo reducido de usuarios con habilidades técnicas avanzadas. Para superar esta limitación, proponemos la aplicación del aprendizaje Zero-Shot, un enfoque innovador en el procesamiento del lenguaje natural. En esta investigación, se lleva a cabo un experimento basado en el modelo <variable>para traducir consultas de lenguaje natural a código *Cypher*. La evaluación se realiza utilizando el conjunto de datos de evaluación <variable>, que abarca una amplia variedad de ejemplos de consultas. Los resultados obtenidos, <variable>, establecen un punto de referencia esencial para el uso de modelos de lenguaje en la traducción de lenguaje natural a código *Cypher*.

Abstract

This thesis focuses on addressing the inherent complexity of querying graph databases like Neo4J. Such databases often require specialized knowledge of query languages, limiting accessibility to a small group of users with advanced technical skills. To overcome this limitation, we propose the application of Zero-Shot learning, an innovative approach in natural language processing. In this research, an experiment is conducted based on the <variable>model to translate natural language queries into *Cypher* code. Evaluation is carried out using the <variable>evaluation dataset, which encompasses a wide variety of query examples. The obtained results, <variable>, establish a crucial benchmark for the use of language models in translating natural language to *Cypher* code.

Índice general

Índice de figuras

Introducción

En la época actual, asistimos a un constante aumento en la producción de información en diversos formatos: visual, auditivo y textual, que abarca todos los ámbitos de la sociedad [4]. De manera particular, resulta sumamente intrigante la información generada a través del ingenio creativo y la investigación humana. Estos tipos de datos se almacenan debido a su relevancia y a la necesidad de acceder a ellos en el futuro, pudiendo optar por una organización estructurada o no. Sorprendentemente, solo alrededor del 20% de la información a nivel mundial se encuentra estructurada [1].

Las bases de conocimiento constituyen un tipo particular de bases de datos diseñadas para la administración del saber. Estas bases brindan los medios para recolectar, organizar y recuperar digitalmente un conjunto de conocimientos, ideas, conceptos o datos [2]. La ventaja fundamental de mantener la información de manera estructurada radica en su facilidad para ser consultada, ampliada y modificada. Debido a su utilidad y prevalencia, la recuperación de información a través de consultas en bases de conocimiento se ha convertido en una tarea esencial.

Es esencial que la información almacenada en bases de conocimiento adopte un formato adecuado para permitir búsquedas ágiles y precisas. Entre los formatos más comunes se encuentran los modelos de Entidad-Relación y el modelo Relacional. A pesar de ser enfoques más antiguos, el modelo Relacional (BDR) sigue siendo el más ampliamente utilizado en la actualidad [3]. No obstante, en ocasiones, las características específicas del problema demandan un formato más expresivo, y es en este punto donde las bases de datos orientadas a grafos (BDOG) [5] entran en juego.

Las BDOG han ganado progresivamente popularidad como una manera efectiva de almacenar información en los últimos años. Estas bases tienen la capacidad de modelar una diversidad de situaciones del mundo real al tiempo que mantienen un alto nivel de simplicidad y legibilidad para

los seres humanos. Las BDOG presentan numerosas ventajas en comparación con las bases de datos relacionales. Esto incluye un mejor rendimiento, permitiendo el manejo más rápido y eficaz de grandes volúmenes de datos relacionados; flexibilidad, ya que la teoría de grafos en la que se basan las BDOG permite abordar diversos problemas y encontrar soluciones óptimas; y escalabilidad, ya que las bases de datos orientadas a grafos permiten una escalabilidad eficaz al facilitar la incorporación de nuevos nodos y relaciones entre ellos. Ejemplo de un sistema de gestión de BDOG es *Neo4J* [**?**], a través del cual es posible construir instancias de este tipo de base de datos e interactuar con las mismas a través del lenguaje de programación *Cypher* [**?**], el cual posee una sintaxis declarativa similar a *SQL* [**?**].

Por otro lado, el avance en la comprensión del lenguaje natural se ha visto potenciado con el surgimiento de los grandes modelos de lenguajes (LLMs) [**?**] como GPT-4 [**?**] o LLaMA-2 [**?**], los cuales presentan una serie de habilidades emergentes como elaboración de resúmenes de textos, generación de código, razonamiento lógico, traducción lingüística entre otras [**?**]. Dichas herramientas constituyen modelos de *Machine Learning* entrenados con un gran volumen de datos, lo cual es posible gracias al número de parámetros con los que estos son configurados [**?**].

Usualmente, para el uso de los LLMs basta con ofrecerles como dato de entrada un texto (*prompt*), el cual describe la tarea que se espera que estos realicen. Además, son muchas las técnicas existentes para elaborar una entrada de calidad, esto con el objetivo de que la respuesta por parte de dicho modelo de lenguage ofrezca resultados alentadores al respecto, lo cual se conoce como *prompt engineering* [**?**]. Una técnica bastánte común es *Zero-Shot Learning* (ZSL) [**?**], la cual consiste en describirle a un LLM un procedimiento a realizar sin ofrecer de antemano ejemplos de cómo resolverlo, como por ejemplo, en tareas relacionadas con la generación de código, donde algunos de estos son capaces de generar algoritmos expresados en un lenguaje de programación formal a partir de una sentencia o consulta en lenguaje natural sin recibir como entrada del usuario algunos ejemplos de código, o especificaciones de cómo funciona el lenguaje objetivo a generar [**?**] [**?**].

En lo que respecta a la comprensión del lenguaje natural y su uso en consultas a bases de conocimiento, existen diversas vías llevadas a cabo y con resultados diversos, donde se hacen análisis sintácticos y semánticos sobre la consulta, muchas veces asistidos por diccionarios o mapas sobre la base de conocimiento en cuestión. Se usan modelos de paráfrasis como técnica de aumento de datos y finalmente Transformers [**?**] o incluso LLMs para llevar de la consulta ya curada al lenguaje de consulta formal o a un

lenguaje intermedio capaz de expresar a esta a alto nivel [**?**] [**?**] [**?**] [**?**].

Por las razones anteriormente expuestas, resulta interesante la investigación sobre la tarea de generación de código de consulta formal a partir de una sentencia en lenguaje natural mediante el uso de LLMs, especialmente el diseño e implementación un experimento capaz de demostrar las capacidades reales de estos para dicho acometido, lo cual designará la importancia de continuar el estudio de dichas herramientas con el objetivo de mejorar los sistemas de extracción de conocimientos en BDOG.

Problemática

Para utilizar el lenguaje de consulta formal *Cypher* se requiere de conocimientos básicos de programación, lo cual consume cierto tiempo y esfuerzo. Esto tiene como consecuencia que, solo aquellas personas con experiencia en el uso de lenguajes de programación puedan hacer uso de la mayoría de los sistemas de almacenamiento de datos desarrollados con esta tecnología y teniendo en cuenta la necesidad de poseer un conocimiento del dominio sobre el cual está construida la base de datos a consultar. Por lo tanto, llevar a cabo una mejora en las herramientas orientadas a democratizar dicho proceso permitiría hacer más rápido y eficiente dicho proceso de consulta en cuanto a tiempo y recursos computacionales. Debido a dicha situación, se propone una experimentación basada en un LLM capaz de traducir una consulta en lenguaje natural a un código en *Cypher*, donde a su vez se verifique la efectividad de este a partir de enfoques basados en ZSL, los cuales intuitivamente pueden ofrecer como resultado una cota inferior para la efectividad de sistemas desarrollados en base a dichos algoritmos de aprendizaje. Además, actualmente la implementación de sistemas de generación de código de consulta formal está principalmente orientada al lenguaje *SQL*, mientras que para el lenguaje *Cypher*, no existen suficiente estudios recientes que avalen la calidad de tales herramientas para dicho caso de uso.

Objetivos

Dadas las ideas anteriores, los objetivos principales del trabajo consistirá en diseñar e implementar una estrategia experimental capaz de evaluar la capacidad mínima de los LLMs para la consulta en lenguaje natural a bases de conocimiento estructuradas con independencia del dominio, para lo cual se empleará un enfoque basado en ZSL.

Para lograr los objetivos generales se trazaron los siguientes objetivos específicos:

1. Estudiar el estado del arte de los modelos de Aprendizaje Automático capaces de hacer predicciones de tipo texto-a-texto.
2. Analizar el trabajo de tesis sobre este tema anteriormente desarrollado en la facultad.
3. Implementar un modelo de Aprendizaje Automático capaz de convertir una consulta en lenguaje natural humano a un lenguaje formal que permita obtener datos a partir de una base de conocimiento.
4. Explorar las capacidades de enfoques Zero-Shot para la traducción de lenguaje natural al lenguaje Cypher.
5. Mejorar el sistema de evaluación de resultados permitiendo que el conjunto de datos de prueba y evaluación sea lo más realista posible y con una mayor complejidad.

Organización de la tesis

[Hablar sobre la estructuracion del documento]

Capítulo 1

Estado del Arte

Con el incremento constante de la cantidad de información generada en todo el mundo, la recuperación de información se ha convertido en un aspecto de creciente relevancia tanto en el ámbito industrial como en el académico. En consecuencia, la reducción del tiempo transcurrido entre el momento en que un usuario desea acceder a la información y el momento en que efectivamente puede hacerlo ha sido objeto de un número creciente de investigaciones científicas en los últimos años. Este capítulo se dedica a la evaluación de diversas estructuras de interfaces entre el ser humano y bases de conocimiento, las cuales tienen como objetivo abordar esta problemática.

Las Interfaces de Lenguaje Natural a Bases de Datos (NLIDB, por sus siglas en inglés) representan un campo de investigación dinámico centrado en facilitar las interacciones entre humanos y computadoras con bases de datos relacionales utilizando consultas en lenguaje natural. A lo largo de las últimas décadas, el desarrollo de NLIDB ha pasado por varias fases transformadoras, impulsadas por avances tecnológicos y metodológicos, así como por una creciente demanda de una mejor accesibilidad a las bases de datos.

Las etapas iniciales del desarrollo de NLIDB se caracterizaron por sistemas específicos de dominio. Estos sistemas fueron diseñados para trabajar dentro de áreas de conocimiento bien definidas, donde se utilizaba el procesamiento de lenguaje natural controlado para garantizar la comprensión de las consultas y la interacción con la base de datos. Por ejemplo, algunos trabajos pioneros [**?**] [**?**] demostraron NLIDBs que se adaptaban a dominios específicos, lo que los hacía altamente efectivos pero limitados en alcance. Del mismo modo, en años posteriores [**?**] [**?**] se continuó la exploración del

uso de interfaces de lenguaje natural controlado dentro de dominios de conocimiento particulares.

Otro enfoque durante esta fase implicó NLIDBs basados en reglas. Algunos sistemas propuestos al respecto [**?**] dependían de reglas predefinidas para traducir consultas en lenguaje natural en declaraciones *SQL* para la recuperación de datos en la base de datos. Si bien estos sistemas ofrecían ciertas ventajas, carecían de versatilidad para manejar una amplia gama de consultas de usuarios en diferentes dominios.

A medida que avanzaba la investigación en NLIDB, hubo un cambio hacia la independencia de dominio y la flexibilidad. Los sistemas recientes han buscado reducir la dependencia del conocimiento específico del dominio y las reglas. Algunos investigadores [**?**] [**?**]) han desarrollado NLIDBs que utilizan técnicas de aprendizaje supervisado, lo que los hace más adaptables a varios dominios y entradas de usuario. Además, un avance significativo se ha producido con la integración de redes neuronales profundas en el desarrollo de NLIDBs, donde varias investigaciones [**?**] [**?**] han demostrado el potencial del aprendizaje profundo en NLIDB, aprovechando vastos repositorios de texto y código para el entrenamiento. Este enfoque ha mejorado significativamente el rendimiento de NLIDB, permitiendo un procesamiento de consultas más natural y contextual.

Para el caso específico de NLIDB con respecto a BDOGs inicialmente se desarrollaron trabajos enfocados en técnicas similares a las empleadas para *SQL* [**?**] [**?**], donde se realizaba un preprocesamiento en la consulta dada, se aprovechaba la información ofrecida por el esquema [**?**] de la base de datos a consultar y finalmente dicho conocimiento era utilizado por un modelo de aprendizaje profundo entrenado sobre un conjunto de pares de lenguaje natural y lenguaje de consulta formal como por ejemplo *Cypher*.

Recientemente, los trabajos orientados a esta área de estudio han estado enfocados en dos metodologías principales que serán tratadas en las secciones 1.2 y 1.3:

1. Enfoques neurosimbólicos basados en representación intermedia de la consulta dada en lenguaje natural.

2. Enfoques basados en técnicas de *prompt engineering* mediante LLMs.

1.1. Preliminares

1.1.1. Bases de Datos *Neo4J*

El sistema *Neo4J* emerge como una plataforma de base de datos de grafos preeminente, distinguiéndose por su capacidad para manejar datos interconectados con una eficiencia y flexibilidad notables [**?**]. Este sistema gestiona las relaciones entre los datos con una estructura nodal y de aristas, lo que permite una representación más natural de las interconexiones inherentes a muchos conjuntos de datos [**?**].

La potencia de *Neo4J* reside en su lenguaje de consulta, *Cypher*, que permite expresar consultas sobre grafos de manera declarativa. *Cypher* es específicamente diseñado para ser intuitivo y potente, proporcionando comandos que facilitan la realización de patrones de búsqueda complejos y análisis de relaciones en una forma compacta y legible [**?**].

En términos de rendimiento, *Neo4J* está diseñado para maximizar la velocidad y eficiencia en la recuperación y manejo de datos relacionales. Utiliza índices basados en árboles B+ y algoritmos de ruta como el de Dijkstra y A* para búsquedas optimizadas en el grafo [**?**]. Además, la integridad de los datos se garantiza mediante propiedades transaccionales ACID [**?**], que son fundamentales en aplicaciones críticas de negocio [**?**].

La escalabilidad de *Neo4J* es una de sus características más destacables, con soporte para configuraciones en clúster que permiten la replicación de datos y la tolerancia a fallos, asegurando la disponibilidad y la escalabilidad horizontal [**?**]. Esto es esencial para aplicaciones que requieren un rendimiento consistente bajo cargas de trabajo de lectura y escritura intensivas.

La versatilidad de integración de Neo4j también merece ser subrayada. Su compatibilidad con API REST, diversos lenguajes de programación y frameworks de desarrollo facilita su adopción en arquitecturas de software existentes [**?**]. Esto se complementa con una comunidad activa y recursos extensos para desarrolladores, lo que promueve una continua innovación y adopción en la industria [**?**].

Figura 1.1: Ejemplo de representación de información en una base de datos *Neo4J*. Nótese la estructura de nodos para representar información de entidades y de aristas que corresponden a relaciones entre estas.

1.1.2. Lenguaje *Cypher*

El lenguaje de consulta utilizado en *Neo4J* es *Cypher*, un lenguaje declarativo centrado en el "qué.[en] lugar del çómoçuando se trata de recuperar datos. Esta naturaleza permite a los usuarios especificar patrones en los grafos sin requerir que ellos describan los algoritmos o pasos lógicos para encontrar esos patrones [**?**]. A continuación se mencionan algunos aspectos relevantes de este lenguaje de consulta formal y cómo facilitan la interacción con bases de datos de grafos como *Neo4J*.

1. **Patrones de coincidencia (Pattern Matching)**: Cypher utiliza una sintaxis que se asemeja a los diagramas de entidad-relación ASCII para patrones de coincidencia, lo que facilita la representación visual de la estructura del grafo en el propio código [**?**]. Su sintaxis intuitiva hace que la comprensión y escritura de consultas sea más accesible, especialmente para usuarios nuevos en el manejo de bases de datos de grafos. Por ejemplo, para encontrar a un usuario y sus amigos en *Neo4J*, podríamos escribir una consulta como:

2. **Filtrado y condiciones**: *Cypher* permite incorporar condiciones dentro de los patrones de coincidencia o en cláusulas `WHERE` para filtrar

```
1 MATCH (user:Person)-[:FRIEND]->(friend) RETURN user, friend
```

Figura 1.2: Ejemplo de código *Cypher*.

resultados. Esto se asemeja al uso de `WHERE` en *SQL* pero está optimizado para trabajar con las conexiones entre nodos [**?**].

3. **Agregación de datos**: *Cypher* proporciona funciones de agregación, como `COUNT`, `SUM`, `AVG`, `MAX` y texttttMIN, que permiten realizar cálculos sobre grupos de datos. Estas funciones son esenciales para resumir información sobre los datos conectados [**?**].

4. **Modificación de grafos**: Además de recuperar datos, *Cypher* puede ser utilizado para crear, actualizar y eliminar nodos y relaciones. Esto incluye la capacidad de manejar transacciones y asegurar la integridad de los datos [**?**].

5. **Optimización de consultas**: *Cypher* está diseñado para optimizar las consultas de grafos de forma automática. El planificador de consultas de *Neo4J* reorganiza y optimiza las operaciones de consulta para una ejecución eficiente, lo que abstrae una capa de complejidad para los desarrolladores [**?**].

6. **Extensibilidad**: *Cypher* es extensible, lo que significa que se pueden crear funciones definidas por el usuario y procedimientos almacenados que se pueden invocar dentro de las consultas. Esto permite personalizar y ampliar la funcionalidad del lenguaje para satisfacer necesidades específicas [**?**].

7. **Interoperabilidad**: A través de su protocolo *Bolt* y la API REST, *Cypher* puede ser utilizado desde una variedad de lenguajes de programación y entornos, permitiendo que sistemas externos interactúen con *Neo4J* [**?**].

1.1.3. Grandes modelos de Lenguaje y Generación de Lenguajes de Consulta Formales

Los Grandes Modelos de Lenguaje (LLMs) son una clase de modelos de procesamiento de lenguaje natural que han revolucionado la forma en que las computadoras comprenden y generan texto. Estos modelos se destacan por su capacidad para generar texto coherente y contextualmente relevante en función del contexto proporcionado. Esta habilidad es esencial cuando se trata de generar código de consulta en lenguajes formales como *Cypher* o textSQL [**?**].

Una de las ventajas clave de los LLMs es su capacidad para traducir preguntas en lenguaje natural en consultas en lenguaje formal de manera automática. Esto significa que pueden tomar preguntas como «¿Cuál es la población de Nueva York? «y convertirlas en consultas *SQL* precisas, como `SELECT population FROM cities WHERE name = 'Nueva York'`. Esta traducción automática simplifica significativamente la interacción entre humanos y sistemas de bases de datos, ya que elimina la necesidad de que los usuarios aprendan el lenguaje formal [**?**].

Otra característica destacada es la versatilidad de los LLMs en términos de lenguajes de consulta. No están limitados a un lenguaje específico; pueden generar consultas en varios lenguajes formales. Esto significa que pueden interactuar con diferentes sistemas de bases de datos que utilizan diferentes lenguajes de consulta, desde *SQL* hasta *Cypher* y *SPARQL*, entre otros [**?**].

Los LLMs también demuestran un fuerte entendimiento de contextos complejos en sus respuestas. Pueden manejar preguntas que involucran múltiples condiciones, cláusulas `JOIN`, filtros y agregaciones en las consultas, lo que los hace aptos para abordar consultas complejas en bases de datos [**?**].

Por último, estos modelos tienen la capacidad de inferir relaciones y estructuras de datos a partir del contexto proporcionado. Esto les permite generar consultas que exploran conexiones y patrones en los datos de una base de datos de manera inteligente, lo que es especialmente valioso en aplicaciones de análisis de datos y minería de información [**?**].

1.2. Extracción de conocimiento mediante enfoques neurosimbólicos basados en representación intermedia (IR, por sus siglas en inglés) de la consulta dada en lenguaje natural

Con respecto a la recuperación de información de una base de conocimiento con este enfoque, mediante consultas hechas en lenguaje natural se han hecho varias investigaciones científicas que se pueden agrupar bajo el patrón de la Figura 1.2.

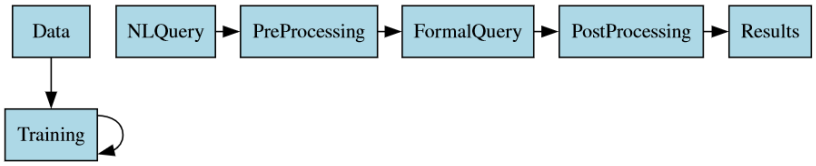

Figura 1.3: Secuencia de flujo representativa del Estado del Arte de traducción de Lenguaje Natural a Lenguaje Formal utilizando el enfoque neurosimbólico basado en (IR).

1.2.1. Conjuntos de entranamiento y evaluación

En cuanto a los datos para el entrenamiento existen dos opciones. Básicamente se puede buscar un conjunto de datos de referencia(*Benchmark*) en los que entrenar y probar el sistema, o se crea uno. La mayoría de las investigaciones existentes elijen la primera opción. Algunos de los *Benchmarks* más populares son:

- **WikiSQL**: WikiSQL [**?**] es el banco de datos más grande y más utilizado, contiene 26531 tablas y 80654 pares de consultas en lenguaje natural y lenguaje *SQL*. Las tablas se extraen de tablas *HTML* de Wikipedia. Luego, cada consulta en *SQL* se genera automáticamente para una determinada tabla bajo la restricción de que la consulta produce un conjunto de resultados no vacío.
- **Spider**: Este *Benchmark* [**?**] es un punto de referencia multidominio a gran escala con 200 bases de datos de 138 dominios diferentes y 10.181 pares de consultas.

- **MetaQA**: El conjunto de datos METAQA-Cypher, originalmente conocido como METAQA [**?**], contiene más de 400000 pares de preguntas y respuestas de múltiples pasos obtenidos de la base de conocimiento WikiMovies [**?**]. Mientras que investigaciones previas se han centrado principalmente en la anotación SPARQL [**?**], nuestra innovación implica reconfigurar METAQA en *Cypher*, estableciéndolo como un valioso punto de referencia para el aprendizaje de pocos ejemplos.

En el caso alternativo, se suelen usar las propias bases de conocimiento objeto de estudio para crear un conjunto de entrenamiento. Una de las técnicas empleadas para esto es Random Walk [**?**], en la que se hace un recorrido aleatorio sobre un subconjunto de las entidades y relaciones de la base de conocimiento, y se elaboran consultas artificiales que respondan a dichas entidades y relaciones [**?**].

1.2.2. Preprocesamiento

En general las investigaciones en el área realizan algún tipo de preprocesamiento a la consulta. Algunos realizan parafraseo para llevar la consulta a representaciones canónicas [**?**], en su lugar otros realizan un análisis morfológico léxico, con toquenización, lematización, eliminación de stopwords, tagueo de partes de la oración (*POS-taguing*), etc. [**?**]. También se encuentran las investigaciones que usan en esta etapa traducciones basadas en diccionarios especializados en el dominio, o de propósito general, al igual que ontologías, para "suavizar"vocablos difíciles de entender por el resto del sistema [**?**]. Además se usan técnicas como vectorización de palabras (*word2vector*), y transformación de la consulta a una representación en grafos [**?**].

1.2.3. Postprocesamiento

En la fase de Post-Procesamiento, se observan diversas enfoques en las investigaciones. La mayoría de los investigadores realizan un análisis semántico que implica la clasificación según tipos de datos, el uso de ontologías y bases de conocimiento externas para realizar mapeos [**?**] [**?**]. Algunos optan por convertir la consulta en una representación canónica, mientras que otros la transforman en un lenguaje intermedio antes de transpilarla al lenguaje objetivo [**?**]. También hay quienes la codifican directamente en forma de grafo, y algunos la convierten en embeddings [**?**].

1.2.4. Consulta

En la fase de construcción de la consulta, existen tres enfoques principales. El primero es un enfoque manual que implica la búsqueda y formateo de palabras clave como *"where"* y *"select"*, además del mapeo de atributos a tablas [**?**]. Otra vía se centra en el uso de modelos, incluyendo decodificadores y en algunos casos redes neuronales convolucionales [**?**]. También se emplea la construcción de la consulta formal mediante un compilador, especialmente cuando se ha utilizado un lenguaje intermedio entre el lenguaje natural y el formal de consulta[**?**].

1.3. Extracción de conocimiento mediante enfoques basados en técnicas de *prompt engineering* mediante LLMs

Con la creciente atención dada a los modelos de lenguaje a gran escala, estos se han convertido en un componente esencial en el procesamiento del lenguaje natural. A medida que aumenta el tamaño de los modelos preentrenados, también está cambiando gradualmente su uso. A diferencia de modelos como BERT [**?**] y T5 [**?**], que requieren un proceso de entrenamiento con una pequeña cantidad de datos, modelos como GPT-3 [**?**] requieren un diseño de un texto de entrada para generar resultados deseados. El reciente modelo de *ChatGPT* [**?**], que emplea Aprendizaje por Reforzamiento para la Retroalimentación Humana (RLHF) [**?**], simplifica el diseño de textos de entrada de calidad, lo que permite una mejor utilización de la capacidad de ZSL de modelos preentrenados a gran escala de manera conversacional. Debido a la sólida capacidad de dichos en la generación de código [**?**] y al hecho de que los modelos de generación de código suelen requerir una gran cantidad de datos anotados para producir buenos resultados [**?**], un modelo de generación de código de ZSL se considera fundamental.

Para la tarea específica de generar código de un lenguaje de consulta formal como *SQL* y *Cypher* se han utilizado distintos enfoques basados en varias de las principales técnicas de *prompt engineering*.

1.3.1. Zero-Shot Learning

Esta técnica se enfoca en la capacidad de un modelo para comprender y generar código en un lenguaje de consulta sin requerir ejemplos específicos de entrenamiento en ese lenguaje en particular. En otras palabras, el

modelo puede realizar esta tarea "desde cero", sin conocimiento previo del lenguaje. Algunos estudio interesantes se han realizado principalmente en la tarea de traducir lenguaje natural a lenguaje *SQL* [**?**] [**?**], los cuales permiten inferir la calidad mínima que estos modelos pueden alcanzar en la realización de dicha tarea [**?**].

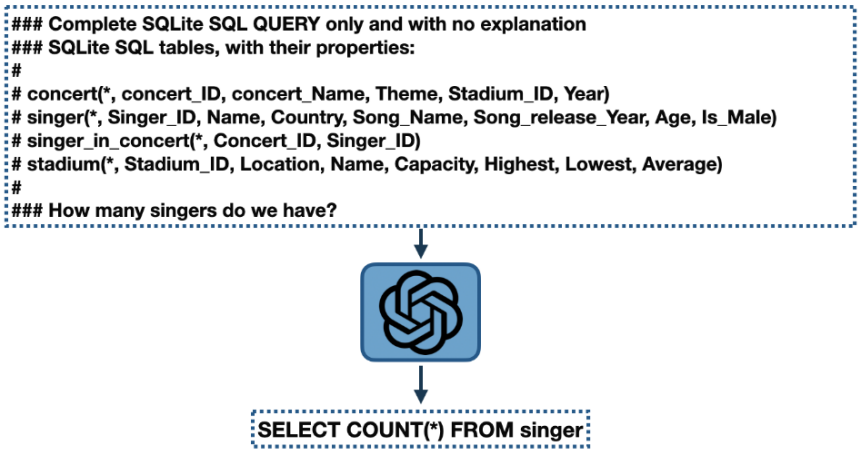

Figura 1.4: Ejemplo del flujo de trabajo de experimentos basados en ZSL para la traducción de lenguaje natural a código en *SQL*.

1.3.2. Few-Shot Learning

En contraste, el enfoque de Few-Shot Learning (FSL) se basa en la idea de que el modelo tiene acceso a un pequeño número de ejemplos (pocos ejemplos) en el lenguaje de consulta deseado para mejorar su capacidad de generar código en ese lenguaje. Esto puede ser especialmente útil cuando se necesita una adaptación rápida a un nuevo lenguaje o contexto. Tal y como muestran algunos resultados experimentales, este enfoque puede tener resultados superiores a varios modelos basados en *fine-tuning* [**?**].

Un ejemplo notable de esta aplicación es el modelo Codex [**?**], que ha demostrado ser un fuerte baseline en el *benchmark* Spider sin ninguna fase de entrenamiento. Además, se ha observado que proporcionar un pequeño número de ejemplos en el dominio en el prompt permite a Codex superar a los modelos de estado del arte cuyos parámetros han sido ajustados con pocos ejemplos de pocos dominios [**?**].

Además, se ha propuesto un marco de selección de demostraciones ODIS [**?**] que utiliza tanto ejemplos fuera de dominio como ejemplos ge-

nerados sintéticamente en el dominio para construir demostraciones. Este enfoque ha demostrado ser efectivo en comparación con los métodos de línea de base que se basan en una única fuente de datos [**?**].

En cuanto a los conjuntos de datos, existen varios conjuntos de datos de texto a SQL que han sido propuestos para evaluar el rendimiento de los modelos LLM. Algunos de estos conjuntos de datos incluyen CoSQL, TableQA, DuSQL, CHASE, y BIRD-SQL [**?**] [**?**] [**?**] [**?**] [**?**] [**?**].

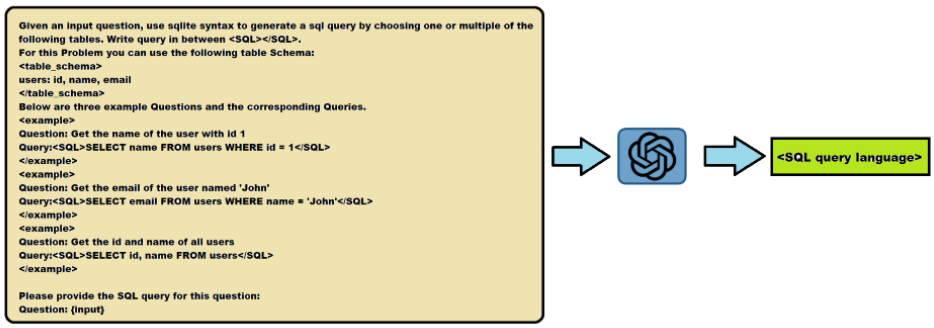

Figura 1.5: Ejemplo del flujo de trabajo de experimentos basados en FSL para la traducción de lenguaje natural a código en *SQL*.

1.3.3. Chain-of-Thought Prompting

El enfoque de la cadena de pensamiento (*Chain of Thought, CoT*) [**?**] en la conversión de texto a *SQL* ha demostrado ser una estrategia prometedora para mejorar la capacidad de los modelos de lenguaje de gran tamaño (LLMs) para realizar razonamientos complejos. Este enfoque se ha utilizado en varias investigaciones recientes para mejorar la capacidad de los LLMs para realizar tareas de razonamiento complejo, como la conversión de texto a SQL [**?**].

Un estudio propuso un nuevo paradigma para la generación de consultas *SQL* a partir de texto, llamado *Divide-and-Prompt*, que divide la tarea en subtareas y luego aborda cada subtarea a través de la cadena de pensamiento. Se presentaron tres métodos basados en la indicación para mejorar la capacidad de los LLMs para generar consultas *SQL* a partir de texto. Los experimentos mostraron que estas indicaciones guían a los LLMs para generar consultas *SQL* a partir de texto con mayor precisión de ejecución [**?**].

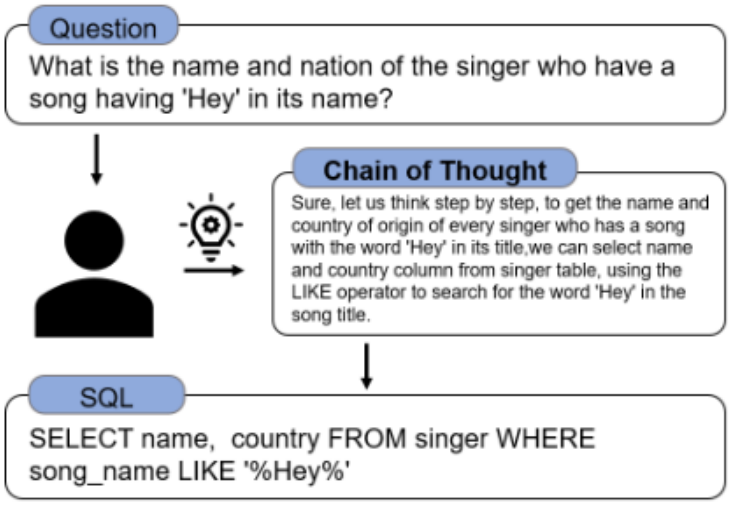

Figura 1.6: Ejemplo del flujo de trabajo de experimentos basados en CoT para la traducción de lenguaje natural a código en *SQL*.

1.3.4. *Fine-Tuning*

El entrenamiento (*fine-tuning*) de un modelo es un proceso que se utiliza para mejorar el rendimiento de un modelo de aprendizaje automático en una tarea específica. En el contexto de la conversión de texto a *SQL*, el ajuste fino puede ser utilizado para mejorar la precisión y la eficacia de los modelos de lenguaje de gran tamaño (LLMs) en la generación de consultas *SQL* a partir de texto. Se han realizado estudios [**?**]ropuso un método de *fine-tuning* para los modelos LLMs utilizando un conjunto de datos de texto a *SQL*, donde el método principal consistía en entrenar el modelo en el conjunto de datos de entrenamiento y luego ajustar el modelo en un conjunto de datos que contenía consultas *SQL* generadas por humanos. Este método demostró ser efectivo para mejorar la precisión de los modelos LLMs en la generación de consultas *SQL* a partir de texto [**?**].

Este enfoque constituye una poderosa herramienta en la conversión de lenguaje natural a *SQL*. Sin embargo, no es una solución mágica, ya que pocas organizaciones tienen conjuntos de datos de entrenamiento *NL-2-SQL* disponibles de manera inmediata. Algunos expertos consideran que las mejores arquitecturas se podrían lograr combinando modelos ajustados con agentes RAG (*Retrieval Augmented Generation*) [**?**].

1.4. Consideraciones generales

A pesar del uso comprobado de dichos enfoques, todavía existe una escasez de estudios orientados a la traducción de lenguaje natural a código de consulta a BDOG como por ejemplo *Cypher* [**?**], por lo tanto, es visible la necesidad de elaborar experimentos enfocados en dicha tarea, tomando como bases las ideas anteriormente expuestas. Debido a esto, resulta imprescindible desarrollar las primeras experimentaciones de la tarea en cuestión utilizando el enfoque ZSL, el cual en promedio permite obtener una cota inferior de qué tan efectivos pueden ser los LLMs con respecto a una tarea específica [].

Capítulo 2

Propuesta de Solución

En el presente capítulo se abordará la metodología seguida para diseñar el experimento propuesto en este trabajo **??**. Primeramente, se expondrá un marco teórico que formaliza la definición del problema a tratar, esto con el objetivo de presentar los conocimientos base tenidos en cuenta para los enfoques probados. Luego, se detallan los primeros acercamientos desechados **??**, argumentando las deficiencias de estos a la hora de arrojar resultados consistentes para la tarea que se desea desarrollar. Finalmente, se detalla la metodología definitiva a implementar, teniendo en cuenta la experiencia obtenida de las anteriores y mostrando su robustez para el análisis experimental [].

De forma general, el componente común para cada vía de solución constituye la presencia de un Gran Modelo de Lenguaje, pues representan los modelos más recientes utilizados para la tarea en cuestión; además, ofrecen resultados alentadores para el caso de traducción a lenguaje *SQL* según lo visto en la sección 1.3. Por lo tanto, tiene sentido probar su eficacia para traducir a código en *Cypher*, ya que ambos presentan similitudes como lenguajes formales declarativos para consultar bases de datos. Dicho modelo será analizado como una "caja negra" capaz de hacer tareas de traducción de lenguaje natural a una consulta semánticamente equivalente en el lenguaje *Cypher*.

Para cada vía de solución se deberá considerar el despliegue de un sistema de gestión de bases de datos para alguna BDOG, ya que es en este componente donde se almacenará la información a extraer por consultas en un lenguaje orientado a este tipo de almacenamiento. En el caso particular de este trabajo, se considerará el uso de *Neo4J*, con el cual se puede interactuar a partir del lenguaje *Cypher* ya mencionado. Por esto, es importante

considerar la implementación de un módulo intermedio para interactuar con una instancia del sistema de gestión *Neo4J*.

El enfoque de *prompt engineering* a utilizar será ZSL, por lo tanto, los textos de entrada que se le darán al modelo para la generación de código *Cypher* no contendrán ejemplos de pares de lenguaje natural con su correspondiente traducción al lenguaje de consulta objetivo. Por lo tanto, se tomarán algunas ideas experimentadas en el estado del arte para *SQL* vistas en la sección **??**.

2.1. Definición del problema *Text-to-Cypher*

Dentro del ámbito de las bases de datos y las consultas que las acceden, existe una tarea particularmente compleja: traducir preguntas formuladas en lenguaje natural a un lenguaje de consulta estructurado como *Cypher*. Dada una pregunta Q, formulada en lenguaje cotidiano, y un esquema de base de datos S, el cual se compone a partir del tuplo N, A, R, donde encontramos múltiples nodos N (que representan instancias de una entidad en la base de datos), atributos C (tanto en nodos como en relaciones) y relaciones entre pares de nodos R. La problemática subyacente en el proceso de convertir *Text-to-Cypher* se centra en generar una consulta en lenguaje *Cypher* Y que sea equivalente y responda adecuadamente a la pregunta inicial Q realizada por un usuario humano.

2.1.1. LLM para resolver *Text-to-Cypher*

Con el auge de la inteligencia artificial y el aprendizaje automático, la tarea de convertir texto en lenguaje natural a código en *Cypher* ha sido recientemente abordada a través de técnicas modernas. En trabajos más recientes, algunos investigadores, **[?]** **[?]**, han abogado por formular esta tarea como un desafío de generación. Utilizando lo que se conoce como "*prompts*.º indicaciones P, es posible dirigir y guiar a un gran modelo de lenguaje M en esta labor. Este modelo, una vez entrenado, puede estimar una distribución de probabilidad sobre posibles consultas *Cypher* Y. De esta forma, el modelo es capaz de generar, paso a paso y token por token, una consulta apropiada.

La fórmula subyacente para generar la consulta Y se estructura como:

$$P_M(Y|P,S,Q) = \prod_{i=1}^{|Y|} P_M(Y_i|P,S,Q,Y<i)$$

Para simplificar, $Y < i$ se refiere al fragmento inicial o prefijo de la consulta *Cypher* que se está construyendo. Mientras que $P_M(Yi|*)$ denota la probabilidad condicional asociada con la generación del i-ésimo token, considerando factores como el prefijo existente $Y < i$, la indicación P, el esquema S y la pregunta original Q.

Uno de los hallazgos más reveladores en el campo reciente es el concepto de aprendizaje en contexto (ICL, por sus siglas en inglés) [**?**], en el cual, grandes modelos de lenguaje pueden adaptarse y aprender de unos pocos ejemplos presentados en un contexto específico. Esta estrategia, defendida por varios investigadores [**?**] [**?**] [**?**], ha mostrado que los LLMs pueden abordar y dominar una amplia variedad de tareas complejas con una cantidad limitada de datos. Sin embargo, hay un equilibrio que mantener: agregar más ejemplos conlleva un aumento en los costos, tanto en términos de mano de obra para preparar esos ejemplos como en términos de costos de procesamiento y tokens al interactuar con APIs avanzadas como la de OpenAI [**?**]. En este estudio, el foco recae en trabajar eficientemente con indicaciones al modelo de lenguaje sin requerir ejemplos adicionales.

2.2. Enfoques considerados

El desarrollo de un sistema con un LLM (*Large Language Model*) para hacer inferencias de *Zero-Shot* y generar *Cypher* implica varios enfoques y desafíos. A continuación, se detallan los enfoques considerados y las dificultades encontradas.

Primero, se exploraron modelos de código abierto. El primer modelo probado fue `Alpaca Lora 7-B` [**?**]. Sin embargo, la calidad de las inferencias no fue la adecuada. En algunos casos, las respuestas eran *Cypher* no compilable con errores de sintaxis. En otros casos, el modelo generaba otro lenguaje de consulta que no era *Cypher*, como por ejemplo *SQL*. Posteriormente, se probó con `GPT4All` [**?**] y con `Vicuna-7B` [**?**], obteniendo resultados similares. Estos modelos, aunque útiles, no proporcionaban la precisión y la generación de *Cypher* que se requería para el sistema.

Además, se consideró la posibilidad de entrenar un propio modelo de lenguaje grande con muchos parámetros para tareas específicas. Este modelo podría ser capaz de generar *Cypher* sin haber sido entrenado específicamente para eso. Sin embargo, esta opción fue descartada debido a la inviabilidad total. Entrenar un modelo de lenguaje grande requiere una gran cantidad de recursos de computación [**?**]. Además, la tarea de entrenar un

modelo para generar *Cypher* sin haber sido entrenado específicamente para eso es extremadamente desafiante. Aunque técnicamente posible, requeriría una inversión significativa de tiempo y recursos, así como un conocimiento profundo de los modelos de lenguaje y el aprendizaje profundo.

En resumen, el desarrollo de un sistema con un LLM para hacer inferencias de *Zero-Shot* y generar *Cypher* implica una serie de desafíos. Estos incluyen la elección del modelo correcto, la necesidad de entrenar el modelo para generar *Cypher* específicamente, y la inversión significativa de recursos y tiempo. A pesar de estos desafíos, el uso de LLMs ofrece la promesa de generar *Cypher* de manera eficiente y precisa, lo que puede ser de gran utilidad en una variedad de aplicaciones.

2.3. Propuesta de solución diseñada

En el desarrollo de sistemas de bases de datos, la interacción eficiente y la gestión de esquemas son fundamentales para la manipulación y el mantenimiento de datos. En este contexto, se ha desarrollado una arquitectura de software compuesta por varios componentes interconectados diseñados para interactuar con una base de datos de grafos, con especial atención en *Neo4J*, un sistema de manejo de bases de datos basado en grafos.

Uno de los componentes clave de esta arquitectura es el `GraphContractor`, un módulo diseñado para facilitar la interacción con la base de datos. Este componente actúa como intermediario entre la aplicación y la base de datos, manejando la lógica necesaria para establecer conexiones, ejecutar consultas y manejar los resultados. La modularidad del `GraphContractor` permite su reutilización y fácil mantenimiento, además de proporcionar una capa de abstracción que simplifica las operaciones de la base de datos para los desarrolladores.

Para la generación de esquemas de la base de datos, se ha creado el `SchemaMaker`, una herramienta automatizada que se encarga de construir los esquemas necesarios para *Neo4J*. Esta herramienta juega un rol crucial en la estructuración de la base de datos, ya que define la organización de nodos, relaciones, propiedades y restricciones. El `SchemaMaker` garantiza que la base de datos esté correctamente configurada para cumplir con los requisitos del dominio y las necesidades de la aplicación, asegurando así la integridad y coherencia de los datos.

En la interfaz entre el lenguaje natural y la base de datos, se ha integrado el modelo de lenguaje `GPT-4`, utilizado como una "caja negra" para la traducción de consultas en lenguaje natural a *Cypher*, el lenguaje de con-

sulta para *Neo4J*. La capacidad de `GPT-4` para comprender y generar texto hace posible que los usuarios realicen consultas complejas sin necesidad de conocer la sintaxis específica de *Cypher*. Para mejorar la precisión y relevancia de las traducciones, se ha elaborado una plantilla de prompt que se nutre de la salida del `SchemaMaker`. Esta plantilla guía al modelo de lenguaje proporcionando contexto y estructura, lo que permite que `GPT-4` genere consultas *Cypher* más precisas y eficientes.

Finalmente, se ha desarrollado el `DBSeeder`, un componente encargado de poblar la base de datos de *Neo4J* con datos iniciales o de prueba. Utilizando consultas *Cypher* generadas por el modelo de lenguaje `GPT-4`, el `DBSeeder` trabaja en conjunto con el `GraphContractor` para insertar nodos, atributos y relaciones en la base de datos. Esta funcionalidad es especialmente valiosa en las etapas de desarrollo y prueba, donde se requiere de una base de datos poblada para validar el diseño y la lógica de la aplicación. Además, el `DBSeeder` está diseñado para operar dentro de un contenedor de *Docker*, lo que ofrece ventajas significativas en términos de portabilidad, escalabilidad y aislamiento del entorno de desarrollo.

Cada uno de estos componentes representa un eslabón en la cadena de herramientas que permitirán interactuar con la base de datos de grafos de manera más intuitiva y automatizada. La integración de tecnologías avanzadas como `GPT-4` en el proceso no solo mejora la accesibilidad para los usuarios finales sino que también agiliza el ciclo de desarrollo, ofreciendo un enfoque moderno y eficiente en la gestión de bases de datos de grafos como *Neo4J*.

2.3.1. Interacción con una base de datos *Neo4J*

El componente `GraphContractor` actúa como un facilitador o intermediario entre el usuario y la base de datos *Neo4J*. Su propósito es simplificar las tareas de conexión y ejecución de consultas contra la base de datos, manejando internamente los detalles de la comunicación y posibles excepciones.

Al instanciar `GraphContractor`, se le proporciona una URL, junto con un nombre de usuario y contraseña para la autenticación. Luego, esta herramienta intenta establecer una conexión con la instancia de la base de datos *Neo4J* en la URL especificada. Si la conexión es exitosa, el `GraphContractor` estará listo para ejecutar consultas. Si la conexión falla, por ejemplo, debido a problemas con la red o las credenciales de acceso, se informará al usuario al respecto con un mensaje informativo.

Una vez que el `GraphContractor` está conectado a la base de datos, es

posible realizar consultas a una base de datos objetivo. Para dicha tarea, este acepta una cadena de texto que representa la consulta *Cypher* a ejecutar. Al efectuar dicha funcionalidad con una consulta de *Cypher* válida, `GraphContractor` la ejecutará en la base de datos y devolverá los resultados. Si ocurre algún error durante la ejecución de la consulta, como una sintaxis incorrecta de *Cypher* o un problema de conexión, el error se capturará y se presentará al usuario, ofreciendo retroalimentación inmediata.

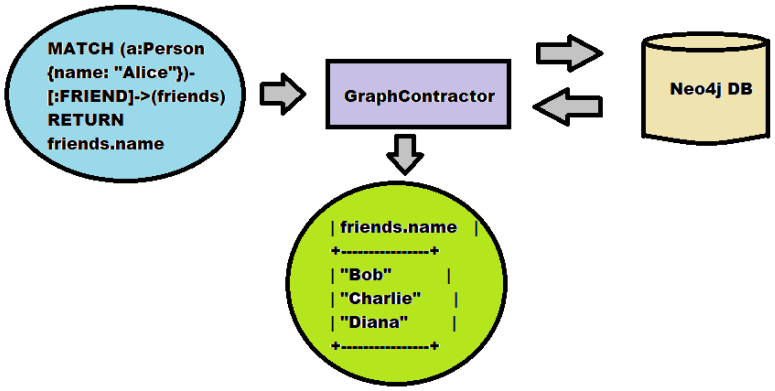

Figura 2.1: Flujo de funcionamiento del componente `GraphContractor`.

En resumen, `GraphContractor` encapsula la complejidad de la gestión de la conexión a la base de datos y la ejecución de consultas, proporcionando una interfaz simplificada para interactuar con *Neo4J*. Esto permite a los usuarios centrarse en la lógica de sus consultas y manejo de datos, en lugar de en los detalles subyacentes de la implementación de la base de datos.

2.3.2. Extracción de información de una base de datos *Neo4J*

En el ámbito de la extracción de información de bases de datos orientadas a grafos, como *Neo4J*, se ha desarrollado un componente denominado `KnowledgeBase`. Este componente actúa como una interfaz avanzada para la interacción con dichas bases de datos, aprovechando las capacidades del componente `GraphContractor`. La funcionalidad principal de `KnowledgeBase` radica en su habilidad para encapsular consultas predefinidas en el lenguaje *Cypher*, el cual es específico para bases de datos *Neo4J*.

El diseño de `KnowledgeBase` tiene como objetivo principal facilitar la extracción de información de manera eficiente y transparente. Para lograr

esto, utiliza consultas en *Cypher* que están integradas dentro de sus funcionalidades. Estas consultas son ejecutadas a través de una instancia interna de `GraphContractor`, proporcionando una capa de abstracción que simplifica las interacciones del usuario con la base de datos.

Sus funcionalidades principales implementadas fueron:

- **Inicialización y Configuración**: El proceso de inicialización establece una conexión vital con la base de datos y prepara el componente para operaciones de consulta. Esto implica la integración con un sistema central que maneja las interacciones con la base de datos.
- **Verificación de Existencia de Entidades**: Una funcionalidad central permite verificar la presencia de entidades específicas en la base de datos. Se emplean etiquetas y propiedades para formular consultas que determinan la existencia de la entidad.
- **Comprobación de Atributos en Entidades**: Otra capacidad importante es determinar si una entidad posee un atributo específico. Esta función es clave para validar la integridad y completitud de los datos.
- **Extracción y Análisis de Entidades y Atributos**: La extracción de etiquetas de entidades provee una visión general de los tipos de datos almacenados. Adicionalmente, se realiza un análisis detallado de los atributos asociados con cada entidad y relación, incluyendo el tipo de dato y los rangos de valores.
- **Inferencia del Tipo de Dato**: La habilidad para identificar el tipo de dato de un valor proporcionado es esencial para el manejo adecuado de los datos, permitiendo realizar conversiones y validaciones de tipo cuando sea necesario.
- **Identificación de Claves y Atributos**: Se realizan operaciones para extraer claves de entidades y relaciones. Esto proporciona información detallada sobre los campos disponibles en diferentes tipos de nodos y enlaces.
- **Análisis de Relaciones entre Entidades**: Identificar y catalogar las relaciones entre distintas entidades es crucial para comprender las interacciones y conexiones dentro de la base de datos.

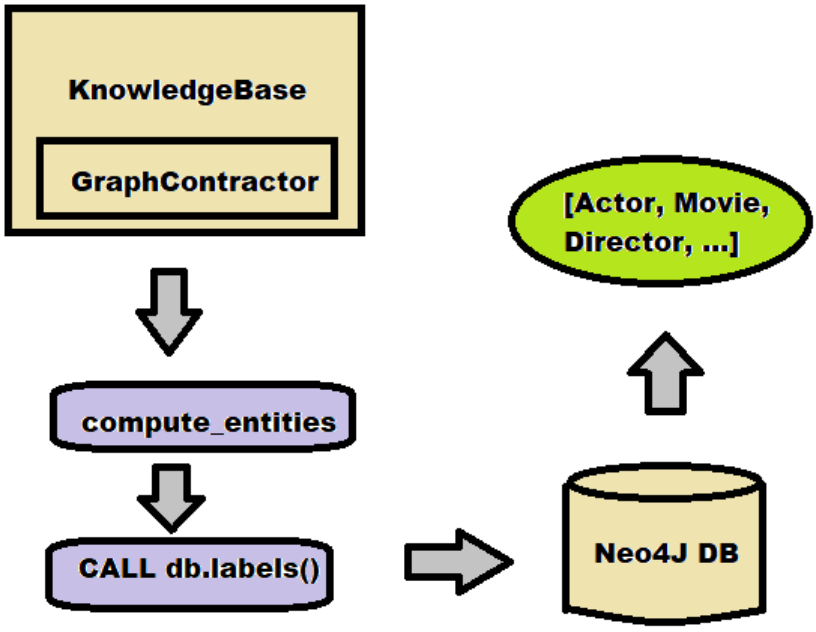

Figura 2.2: Flujo de funcionamiento del componente `KnowledgeBase`.

2.3.3. Almacenamiento de información en una base de datos *Neo4J*

El componente `DBSeeder` es una herramienta diseñada para cargar datos en una base de datos en forma de grafo. Su propósito es automatizar el proceso de toma de datos estructurados y su inserción en la base de datos, creando nodos y relaciones entre ellos según se define en los datos de entrada. La interacción de este con una instancia de una base de datos *Neo4J* es a través de una instancia del componente `KnowledgeBase` visto en la sección 2.3.2.

Al inicializar esta herramienta, se le proporcionan dos piezas de información esenciales: un conjunto de conocimientos que describe la estructura de la base de datos y una ruta a un archivo que contiene los datos a ser sembrados en la base de datos. Estos datos de entrada son esenciales para guiar el proceso de sembrado.

Una vez configurada, la herramienta tiene la capacidad de procesar los datos de entrada. Esto se realiza leyendo cada línea del archivo de datos, donde cada línea representa un conjunto de información que debe ser transformada en elementos dentro de la base de datos. La herramienta ana-

liza cada línea para comprender y aislar las partes que corresponden a entidades y las relaciones entre ellas.

Para cada conjunto de datos, la herramienta verifica si los elementos ya existen en la base de datos. Si no es así, procede a crear nuevos nodos que representan entidades y luego establece relaciones entre estos nodos, basándose en la relación especificada en los datos.

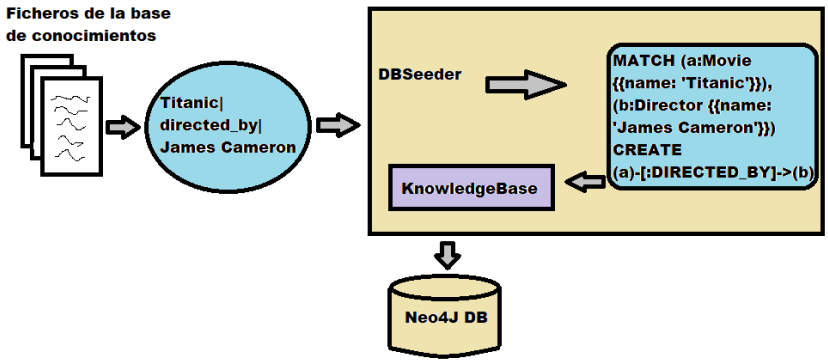

Figura 2.3: Flujo de funcionamiento del componente `DBSeeder`.

El proceso de llenado en la base de datos asegura que no se introduzcan duplicados y que los datos se estructuren correctamente esta la base de datos de acuerdo con sus reglas y definiciones. Al finalizar, la base de datos debería reflejar una red de nodos interconectados que representan tanto las entidades como las relaciones definidas en el archivo de datos original. Este proceso es fundamental para preparar la base de datos para su uso en aplicaciones que requieren acceso a datos relacionales y estructurados en forma de grafo.

2.3.4. Selección del modelo

La selección de `GPT-4` para la generación de código a partir de lenguaje natural utilizando aprendizaje *Zero-Shot* (ZSL) está justificada por varias razones fundamentadas en investigaciones y comparaciones técnicas recientes. `GPT-4` es un avance significativo respecto a sus predecesores, construido sobre la arquitectura de `GPT-3` pero alcanzando nuevos niveles de

rendimiento y escala []. Este modelo mejora en la corrección factual de las respuestas y reduce las .ªlucinaciones", donde el modelo comete errores de hecho o razonamiento, obteniendo un 40% más de precisión que `GPT-3.5` en las pruebas de rendimiento factual internas de OpenAI [].

El modelo `GPT-4` se basa en la arquitectura Transformer, que utiliza mecanismos de atención para procesar texto, y se ha mejorado con una mezcla de expertos (MoE) para lograr un modelo con aproximadamente 1,76 billones de parámetros, un orden de magnitud mayor que `GPT-3` []. Además, estudios recientes han mostrado que `GPT-4` supera a `GPT-3.5` en aprendizaje zero-shot en casi todas las tareas evaluadas, lo que incluye una variedad de dominios de razonamiento como deductivo, inductivo, abductivo, analógico, causal y multi-salto, a través de tareas de preguntas y respuestas [].

Además, el modelo `GPT-4` emplea técnicas de *fine-tuning* y Aprendizaje por Refuerzo con Retroalimentación Humana (RLHF), lo que le permite ser un modelo multimodal robusto capaz de procesar entradas textuales y visuales y generar salidas basadas en texto []. Este enfoque ha demostrado ser eficaz en la mejora de las capacidades de razonamiento de los modelos de lenguaje grandes (LLMs), lo que lo hace especialmente adecuado para tareas complejas que requieren razonamiento, como la traducción de lenguaje natural a lenguaje de consulta formal [].

Cuadro 2.1 Rendimiento de GPT-4 en referencias académicas. []

	GPT-4	**GPT-3.5**	**LM SOTA**	**SOTA**
MMLU	86.4%	70.0%	70.7%	75.2%
HellaSwag	95.3%	85.5%	84.2%	85.6%
AI2 Reasoning Challenge (ARC)	96.3%	85.2%	85.2%	86.5%
WinoGrande	87.5%	81.6%	85.1%	85.1%
HumanEval	67.0%	48.1%	26.2%	65.8%
DROP	80.9%	64.1%	70.8%	88.4%
GSM-8K	92.0%	57.1%	58.8%	87.3%

En comparación con otros modelos como `PALM, Chinchilla, LaMDA, LLaMA` y `Gopher`, que también han evaluado sus habilidades de razonamiento, `GPT-4` se destaca por su capacidad mejorada de aprendizaje zero-shot y por el uso de estrategias de prompting refinadas para mejorar aún más su rendimiento en tareas de razonamiento, lo que lo convierte en una elección

prometedora para la generación de código [].

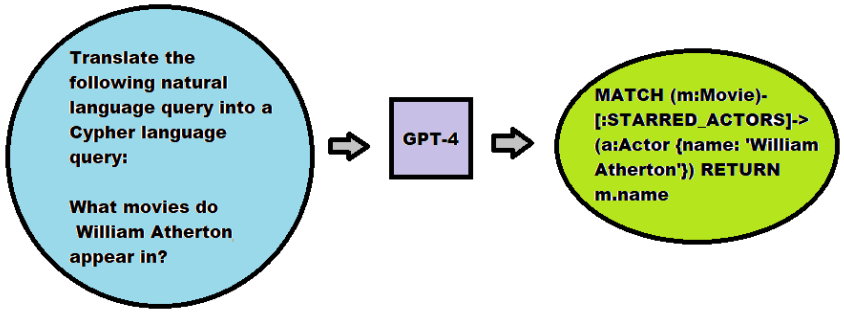

Figura 2.4: Ejemplo básico sobré como utilizar `GPT-4` para traducir lenguaje natural en lenguaje de consulta *Cypher*.

2.3.5. Diseño de la información entrada al LLM

Tal y como se mostró en la sección anterior, el uso de `GPT-4` para la tarea de traducción no resulta complicado, pues este es utilizado como una gran "caja negra" capaz de realizar operaciones que se le indiquen en un texto (*prompt*) de entrada. Debido a esto, también es importante mencionar que no cualquier entrada es efectiva o suficiente para obtener el resultado esperado [**?**]. Al proceso de diseñar una entrada de calidad para que el modelo realice una tarea específica exitosamente se denomida *prompt engineering* [**?**].

Dado que la hipótesis central de este trabajo se basa en el uso del aprendizaje *Zero-Shot* (ZSL), el texto de entrada al modelo `GPT-4` no puede contener ejemplos de cómo traducir una consulta en lenguaje humano a lenguaje *Cypher*, es decir, no deberá reflejar contenido demostrativo de la tarea a realizar, lo cual se justifica por la misma definición del enfoque ZSL **??**. Además, como parte de la información de entrada al modelo para este tipo de tareas, es común añadir una descripción de la estructura de la base de datos a consultar [**?**] [**?**], lo cual se conoce como esquema de la base de datos [**?**].

Por lo mencionado anteriormente, el texto de entrada al modelo deberá contener:

- **La tarea a ejecutar**: Se le describirá al modelo la tarea a realizar, mencionando los datos que recibirá de entrada y el formato en que se desea obtener la respuesta.

- **Esquema de la base de datos**: Se especificará el contenido de la base de datos objetivo en forma de grafo, mencionando las entidades, relaciones (especificando si son en una sola o ambas direcciones entre un par de entidades) y atributos presentes en la misma (correspondientes tanto a las entidades como a las relaciones entre estas).
- **Consulta en lenguaje natural**: En este caso, se añadirá la consulta en lenguaje natural humano a traducir.

Para la obtención del esquema de la base de datos de tipo *Neo4J* se diseñó el componente `SchemaMaker`. Con el fin de elaborar la descripción de la estructura de la fuente de datos objetivo, este recibe los nombres de las entidades, las relaciones y los atributos presentes en la misma. Dicha información la obtiene auxiliándose del componente `KnowlegdeBase` analizado en la sección 2.3.2, mediante el cual se realizan las consultas en lenguaje *Cypher* pertinentes a la instancia de la fuente de datos *Neo4J* en cuestión. A continuación se muestra un ejemplo del funcionamiento de la herramienta `SchemaMaker`:

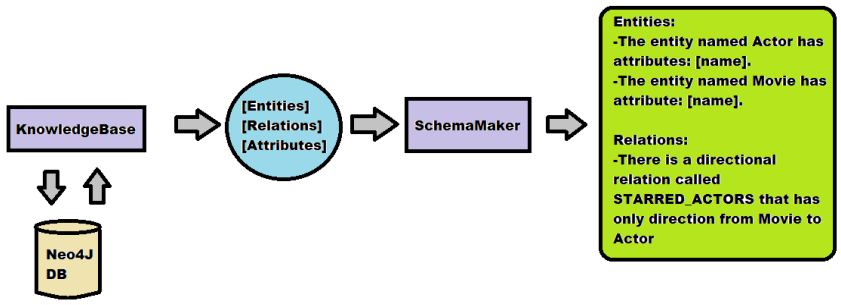

Figura 2.5: Ejemplo del proceso que se realiza para obtener un formato verbalizado de una base de datos *Neo4J* con el uso del `SchemaMaker`.

Finalmente, el texto de entrada para el modelo a utilizar mencionado en la sección 2.3.4 se integraría de la siguiente manera al proceso de traducción de lenguaje natural a lenguaje *Cypher*:

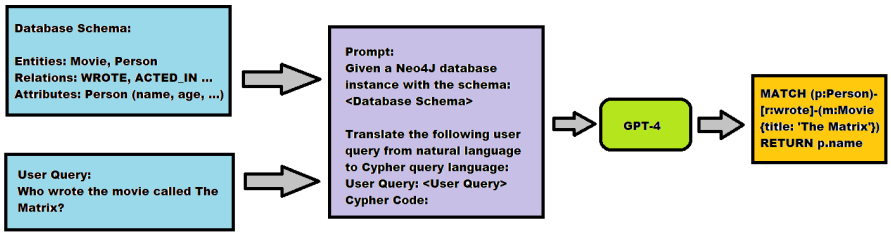

Figura 2.6: Flujo de entrada y salida en el proceso de traducción de lenguaje natural a código en *Cypher*.

Tal y como se muestra en la imagen anterior, junto con la entrada de una consulta en lenguaje natural se elabora un texto de entrada al modelo utilizando además, el esquema de la base de datos *Neo4J* a consultar, la cual como ya se mencionó en esta subsección, es producida por el `SchemaMaker`.

2.3.6. Caso de estudio

A partir de los contenidos abordados en esta sección, resulta importante mostrar la arquitectura general del sistema diseñado, así como el flujo de funcionamiento de la misma.

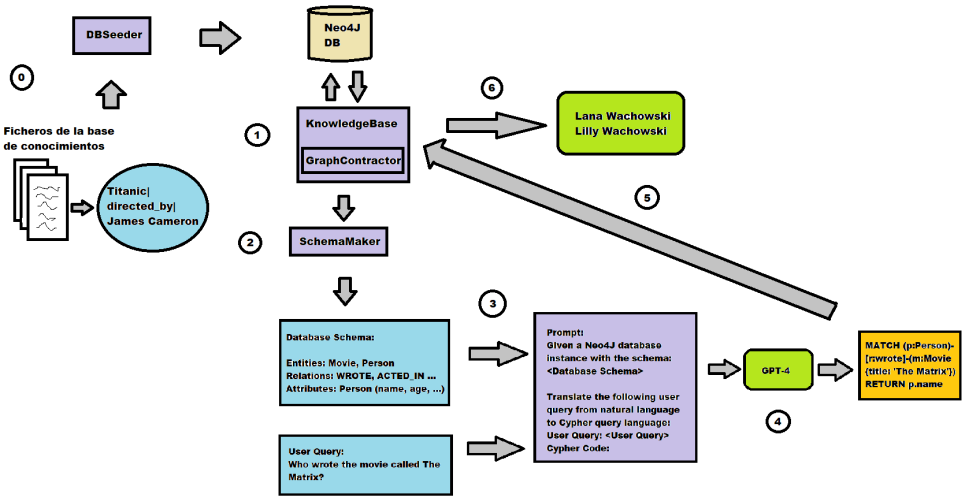

Figura 2.7: Arquitectura y funcionamiento de la propuesta de solución.

Para una mejor comprensión de la figura 2.3.6 se añadieron un conjunto de íncides que resaltan las distintas fases por las que pasa el sistema implementado. A continuación se enumeran y explican en qué consisten cada una de estas etapas:

1. **Paso 0**: Este representa el proceso mediante el cual se construye una instancia de una base de datos *Neo4J* a partir de un conjunto de ficheros de texto. Esta tarea se puede llevar a cabo mediante el componente `DBSeeder`, el cual puede programarse con funcionalidades específicas de acuerdo al formato de los ficheros iniciales y los datos que estos contienen. Como se mencionó en la sección 2.3.3, el componente `DBSeeder` se apoya del componete `GraphContractor` internamente para realizar peticiones a la base de datos con el objetivo de añadir nuevos registros de información, traduciendo la creación de entidades, relaciones y atributos en instrucciones de *Cypher* ejecutadas sobre una base de datos *Neo4J* objetivo.

2. **Paso 1**: En esta fase, el componente `KnowledgeBase` extrae la información referente a las entidades, relaciones y atributos de una base de datos *Neo4J* existente. Para ello, hace uso internamente de una instancia de un `GraphContractor` 2.3.1.

3. **Paso 2**: En este paso, se utiliza la herramienta `SchemaMaker` 2.3.5, el

cual recibe la información de la base de datos procedente del componente `KnowledgeBase` y produce una descripción verbalizada y legible en lenguaje natural sobre la estructura de la instancia de *Neo4J* objetivo.

4. **Paso 3**: En este momento del proceso general, se recibe una consulta descrita en lenguaje natural humano sobre una solicitud de datos de la base de datos objetivo. Luego esta es utilizada junto con el esquema de la base de datos obtenido del `SchemaMaker` para conformar un texto de entrada al modelo `GPT-4` con las características mencionadas en la sección 2.3.5.

5. **Paso 4**: Una vez obtenido el texto de entrada para realizar una inferencia con el modelo `GPT-4`, se procede a hacer una ejecución de este, produciendo así como salida un texto que representa un código en lenguaje *Cypher*.

6. **Paso 5**: En esta etapa, se utiliza la salida del modelo `GPT-4` para ser ejecutada sobre la base de datos *Neo4J* objetivo mediante la herramienta `KnowledgeBase`.

7. **Paso 6**: Finalmente, se obtiene la respuesta procedente de la base de datos *Neo4J* con la información solicitada.

   Con lo anteriormente explicado se expone un ejemplo de caso de uso donde se tienen como entradas al sistema una instancia de una base de datos *Neo4J* y una consulta en lenguaje humano para obtener como salida final el conjunto de datos extraídos de dicha base de datos objetivo correspondientes a la solicitud textual dada sobre estos.

Capítulo 3

Detalles de Implementación

En este capítulo se exponen los detalles sobre la implementación de los componentes que intervinieron en el proyecto, resaltando los principales aspectos de programación tenidos en cuenta.

El sistema implementado fue desarrollado con el *IDE VSCode* [**?**], el cual fue configurado para utilizar el lenguaje *Python* (versión 3.9) [**?**] y contar con las facilidades que esta herramienta ofrece para detectar errores de sintaxis y de dependencias en tiempo de compilación. Por otro lado, el sistema operativo utilizado para implementar el proyecto fue *Ubuntu-22.04*, con el cual se facilitó el proceso de instalación de bibliotecas para el lenguaje de programación utilizado.

En general, no se optó por utilizar una arquitectura de software específica, ya que el objetivo del proyecto se centra fundamentalmente en utilizar el sistema resultante para evaluar los experimentos orientados a responder la hipótesis de estudio y no en desarrollar una aplicación a ser llevada a producción para emplearse por usuarios humanos.

3.1. Despliegue de una instancia de una base de datos *Neo4J*

Para desplegar una instancia de una base de datos en forma de grafo de tipo *Neo4J* nos apoyamos en *Docker* [**?**], con el cual se desplegó un contenedor que pudiese ejecutar el sistema de gestión de *Neo4J* y ser accesible ante peticiones de una aplicación mediante el protocolo *Bolt* [**?**].

Una vez intalado *Docker* se utilizó el siguiente comando:

este comando ejecuta un contenedor Docker con la imagen de *Neo4J*,

```
1  docker run \
2 --name testneo4j \
3 -p 7474:7474 -p 7687:7687 \
4 -d \
5 -v $HOME/neo4j/data:/data \
6 -v $HOME/neo4j/logs:/logs \
7 -e NEO4J_AUTH=neo4j/testpassword \
8 neo4j
```

Figura 3.1: Comando utilizado para desplegar una base de datos *Neo4J*.

expone los puertos 7474 y 7687, guarda los datos y los registros en el *host*, y establece las credenciales de autenticación para la interfaz de usuario de *Neo4J*:

- **–name testneo4j**: Esto asigna el nombre testneo4j al contenedor que se está.
- **-p 7474:7474 -p 7687:7687**: Estas opciones mapean los puertos 7474 y 7687 del contenedor a los mismos puertos del *host*. Esto permite acceder a los servicios que se ejecutan en estos puertos dentro del contenedor desde el *host*.
- **-d**: Esta opción hace que el contenedor se ejecute en segundo plano (modo *detached*).
- **-v** $HOME/neo4j/data : /data - v$**HOME/neo4j/logs:/logs**: Estas opciones montan los directorios $HOME/neo4j/data y$HOME/neo4j/logs del *host* en los directorios */data* y */logs* del contenedor, respectivamente. Esto permite que los datos y los registros generados por el contenedor se guarden en el *host*.
- **-e NEO4J_AUTH=neo4j/testpassword**: Esta opción establece la variable de entorno `NEO4J_AUTH` en el contenedor con el valor *neo4j/testpassword*. Esto se utiliza para configurar la autenticación en *Neo4J*.
- **neo4j**: Esta es la imagen de la que se está creando el contenedor. En este caso, se está utilizando la imagen oficial de *Neo4J*.

3.2. GraphContractor

Para la implementación del componente `GraphContractor` se diseñó una clase para interactuar con una instancia de base de datos *Neo4J*. La clase hereda de la clase `Graph` del módulo `py2neo`, que proporciona una interfaz de alto nivel para interactuar con bases de datos *Neo4J*.

```
1  class GraphContractor(Graph):
2      """
3          Graph Contractor class for interacting with a Neo4J DB
               instance
4      """
5
6      def __init__(self, url, name, password):
7          try:
8              self.graph = Graph(url, auth=(name, password))
9
10         except Exception as e:
11             print(e)
12             print('Error connecting to the database(Remember VPN)')
13
14     def make_query(self, query: str):
15         try:
16             return self.graph.run(query).data()
17         except BaseException as e:
18             print(e)
19             return str(e)
```

Figura 3.2: Implementación de la clase `GraphContractor`.

La clase GraphContractor tiene un método `__init__` que se utiliza para inicializar una nueva instancia de la clase. Este método toma tres argumentos: `url`, `name` y `password`, que se utilizan para establecer una conexión con la base de datos *Neo4J*. Esta clase también tiene un método `make_query` que se utiliza para ejecutar consultas en la base de datos *Neo4J*. Este método toma una consulta en formato de cadena y la ejecuta en la base de datos.

3.3. KnowledgeBase

El componente KnowledgeBase fue implementado en una clase cuyo constructor recibe una instancia de una entidad de tipo GraphContractor:

```
1  class KnowledgeBase:
2     def __init__(self, graph: GraphContractor) -> None:
3         self.graph = graph
4
5      def entity_exists(self, label, property_name, property_value):
6        ...
7     def entity_has_attribute(self, label, property_name,
          entity_name):
8        ...
9
10    def compute_entities(self):
11        ent = self.graph.graph.run('CALL db.labels()')
12        entities = QueryUtils._unfold_graph_resp(ent)
13        return entities
14
15    def compute_attributes(self, entities, relations):
16       ...
17    def compute_relations(self, entities):
18       ...
19    def _infer_data_type(self, value):
20       ...
21    def get_type_min_max_entity_attribute(self, entity,
          attribute_name):
22       ...
23    def get_type_min_max_relation_attribute(self, relation,
          attribute_name):
24       ...
25    def get_keys_of_label(self, label):
26       ...
27    def get_keys_of_relation(self, relation):
28       ...
```

Figura 3.3: Implementación de la clase KnowledgeBase.

Esta herramienta presenta un conjunto de métodos auxiliares para di-

versas tareas que impliquen la extracción de información de una base de datos, donde cada uno internamente utiliza la instancia del `GraphContractor` proporcionado. Por ejemplo, tal como se muestra en la figura 3.3, para obtener el conjunto de entidades de la base de datos se ejecuta un código de *Cypher* correspondiente sobre la instancia de *Neo4J* a consultar.

3.4. DBSeeder

La clase `DBSeeder` en el código proporcionado en la figura 3.4 se utiliza para poblar (*seed*) una base de datos con información de una base de datos base y una base de conocimientos de tipo *Neo4J*.

```
1 class DBSeeder:
2   def __init__(self, kb: KnowledgeBase, db_base_file_path: str) ->
        None:
3       self.kb = kb
4       self.db_base_file_path = db_base_file_path
5
6    def seed_db(self):
7      ...
8   def create_entities_relations_attributes(self, entity1,
        relation_type, entity2):
9      ...
```

Figura 3.4: Implementación de la clase `DBSeeder`.

El método constructor `__init__` recibe dos argumentos: `kb`, que es una instancia de `KnowledgeBase`, y `db_base_file_path`, que es la ruta del archivo de la base de datos base. Por otro lado, el método `seed_db` se encarga de almacenar información en la base de datos objetivo. Este método abre el el conjunto de ficheros que contienen información de la base de datos para su lectura y luego itera sobre cada línea del archivo, llamando al método `create_entities_relations_attributes`, el cual verifica si las entidades `entity1` y `entity2` existen en la base de datos. Si no existen, se crean. Luego, se crea una relación entre el par de entidades dadas y la relación especificada.

3.5. SchemaMaker

En la figura 3.6 se muestra la implementación de la clase `SchemaMaker`, la cual tiene el método estático `compute_schema_description`.

```
class SchemaMaker:
    @staticmethod
    def compute_schema_description(entities, relations, attributes):
        schema_description = ""
        schema_description += f"Entities: {entities}\n"
        for entity in entities:
            entity_attrs = attributes[entity]
            if len(entity_attrs) > 0:
                schema_description += f"The entity named {entity} has
                    the attributes: {[attr[0] for attr in
                    entity_attrs]}\n"
        for relation in relations:
            for ent1, ent2, is_double_sense in relations[relation]:
                if is_double_sense:
                    schema_description += f"There is a relation
                        called {relation} between the entitites
                        {ent1} and {ent2}. The relation {relation}
                        can be used in both senses.\n"
                    continue
                schema_description += f"There is a directional
                    relation called {relation} that has only
                    direction from {ent1} to {ent2}.\n"
            if len(attributes[relation]) > 0:
                schema_description += f"The relation {relation} has
                    attributes {attributes[relation]}\n"
        return schema_description
```

Figura 3.5: Implementación de la clase `Schema Maker`.

Dicho método toma tres argumentos: `entities`, `relations` y `attributes`, los cuales utiliza para generar una descripción del esquema de una base de datos. Este comienza inicializando una cadena vacía `schema_description` y luego agrega información sobre las entidades y relaciones. Primero, agrega una línea que indica las entidades presentes en el esquema. Luego, para

cada entidad, si tiene atributos, agrega una línea que indica los atributos de esa entidad. Después de manejar las entidades, el método pasa a las relaciones. Para cada relación, si es de doble sentido, agrega una línea que indica que existe una relación bidireccional entre las dos entidades involucradas. Si no es de doble sentido, agrega una línea que indica que existe una relación unidireccional desde la primera entidad a la segunda. Finalmente, si la relación tiene atributos, agrega una línea que indica los atributos de la relación.

3.6. `GPT-4`

Para implementar una estructura capaz de realizar inferencias a partir del model `GPT-4` se utilizó la biblioteca *Langchain* y se definió una función `get_model` que se utiliza para inicializar un modelo de lenguaje basado en el tipo de modelo especificado. Fue necesario además, el uso de una `API_KEY` de *OpenAI* con el objetivo de acceder a los modelos disponibles [**?**].

La función `get_model` toma dos argumentos: `model_type` y `model_name`. La variable `model_type` puede ser chat o cualquier otro tipo de modelo, y `model_name` es el nombre del modelo específico que se va a utilizar. Para el caso específico de este trabajo el tipo de modelo fue chat y el nombre utilizado fue gpt-4.

Primero, la función inicializa una plantilla de *prompt* utilizando de acuerdo a si el modelo a utilizar es de tipo chat o generativo. La plantilla de *prompt* se inicializa con un texto predefinido que describe la tarea del modelo y los marcadores de posición para el lenguaje de consulta, el tipo de base de datos, el esquema y la consulta. Luego, la función inicializa un modelo de lenguaje basado en `model_type`. Ambos modelos se inicializan con un nombre de modelo y una temperatura. Finalmente, la función inicializa una instancia de un modelo capaz de hacer inferencias a partir de un texto de entrada predefinido con variables.

```python
from langchain.prompts import PromptTemplate, ChatPromptTemplate
from langchain.chat_models import ChatOpenAI
from langchain import LLMChain
from langchain import OpenAI

# template for the model
template = """
You are an agent capable of transforming natural language queries
    to queries in the query language {query_language}. Your task
    is: Given a database schema of type {database_type} and a query
    written in human natural language, return only the code to
    answer that query in the query language {query_language} and
    respect the relations directions.

The database schema is: {schema}

The natural language query is: {query}

The code in the query language {query_language} is:

""".strip()

def get_model(model_type, model_name):
    # Init prompt template
    prompt = ChatPromptTemplate.from_template(template=template) if
        model_type == "chat" else PromptTemplate(template=template,
        input_variables=[
        "query_language", "database_type", "schema", "query"])

    # Init llm
    llm = ChatOpenAI(model=model_name, temperature=0.7) if
        model_type == "chat" else OpenAI(temperature=0.7)

    # Init chain
    llm_chain = LLMChain(prompt=prompt, llm=llm)

    return llm_chain
```

Figura 3.6: Implementación para utilizar el modelo `GPT-4`.

Capítulo 4

Análisis Experimental

4.1. Enfoques considerados en la evaluación

4.1.1. Elaboración manual de consultas de prueba

4.1.2. Generación de consultas de prueba sintéticas

4.1.3. *Benchmark* orientado a *Cypher*

Conclusiones y Recomendaciones

Bibliografía

[1] Amazon. What is structured data? https://aws.amazon.com/es/what-is/structured-data/, 2023. (Citado en la página 11).

[2] Web Archive. Creación de una base de conocimiento. https://web.archive.org/web/20180315025341/http://es.ccm.net/faq/2158-organizacion-crear-una-base-de-conocimientos, 2023. (Citado en la página 11).

[3] SAP insights. ¿qué es el modelado de datos? https://www.sap.com/latinamerica/products/technology-platform/datasphere/what-is-data-modeling.html, 2023. (Citado en la página 11).

[4] Fundación MAPFRE. ¿cuánta información se genera y almacena en el mundo? https://www.fundacionmapfre.org/blog/cuanta-informacion-se-genera-y-almacena-en-el-mundo/, 2023. (Citado en la página 11).

[5] Wikipedia. Graph database. https://en.wikipedia.org/wiki/Graph_database, 2023. (Citado en la página 11).