

# **Unidad 5 - Almacenamiento y Representación del Conocimiento**

**UNR - TUIA - Procesamiento de Lenguaje Natural**

Docente teoría: Juan Pablo Manson - [jpmanson@gmail.com](mailto:jpmanson@gmail.com) - [LinkedIN](#)

## **1. Búsqueda semántica**

La búsqueda semántica denota una búsqueda con significado, a diferencia de la búsqueda léxica donde el motor de búsqueda busca coincidencias literales de las palabras de la consulta o variantes de ellas, sin comprender el significado general de la consulta. La búsqueda semántica busca mejorar la precisión de la búsqueda al comprender la intención del buscador y el significado contextual de los términos tal como aparecen en el espacio de datos buscable (searchable en inglés), ya sea en la Web o dentro de un sistema cerrado, para generar resultados más relevantes. El contenido que se posiciona bien en la búsqueda semántica está bien escrito en una voz natural, se centra en la intención del usuario y considera temas relacionados que el usuario pueda buscar en el futuro.

Va más allá del simple emparejamiento de palabras y busca comprender la intención del usuario y el contexto semántico de la consulta para proporcionar

resultados más relevantes y precisos. Estos son algunos puntos clave sobre la búsqueda semántica:

1. **Comprendión del Contexto:** La búsqueda semántica se esfuerza por entender el contexto en el que se usan las palabras. Por ejemplo, la palabra "banco" puede referirse a una institución financiera o a un lugar para sentarse en un parque. Un motor de búsqueda semántica intentará determinar cuál de estos significados es el más relevante según el contexto de la consulta.
2. **Intención del Usuario:** La búsqueda semántica también intenta comprender la intención detrás de una consulta. Por ejemplo, si alguien busca "cómo atar una corbata", es probable que esté buscando instrucciones o un video tutorial, y no solo páginas que mencionen esa frase.
3. **Uso de Ontologías y Grafos de Conocimiento:** Muchos sistemas de búsqueda semántica utilizan ontologías o grafos de conocimiento, que son representaciones estructuradas de información y relaciones entre conceptos. Estas estructuras ayudan a los sistemas a comprender las relaciones semánticas entre diferentes palabras y conceptos.
4. **Mejora de la Relevancia:** Al comprender mejor la intención y el contexto, la búsqueda semántica puede proporcionar resultados que son más relevantes para el usuario, lo que mejora la experiencia del usuario y la eficiencia de la búsqueda.
5. **Tecnologías de Procesamiento del Lenguaje Natural (NLP):** La búsqueda semántica a menudo se basa en tecnologías de procesamiento del lenguaje natural para analizar y comprender el lenguaje humano, lo que permite una mejor interpretación de las consultas.
6. **Evolución de los Motores de Búsqueda:** Con el tiempo, los motores de búsqueda han evolucionado para incorporar capacidades semánticas. Por ejemplo, Google introdujo el "Knowledge Graph" para mejorar la comprensión de las relaciones entre diferentes conceptos y proporcionar respuestas más directas a las consultas de los usuarios.

## Antes del uso del sentido

### Búsqueda por Palabras Clave

Antes de la búsqueda semántica, los sistemas normalmente creaban un índice basado en palabras clave para ayudar a encontrar datos. Con herramientas de

vectorización TF-IDF y rankeo usando BM25, es posible construir índices de búsqueda sobre textos. Apache Lucene y Elasticsearch son implementaciones de búsqueda de palabras clave a gran escala y de nivel empresarial.

En forma general, estos métodos tokenizan o dividen el texto en tokens y calculan métricas sobre la importancia de esos términos. Para mejorar la eficacia de la búsquedas, estas técnicas realizan tareas como normalizar las diferencias entre mayúsculas y minúsculas, lematización o stemming, o se eliminan stopwords, entre otras cosas. La búsqueda de palabras clave se ha mantenido muy bien a lo largo del tiempo y todavía tiene vigencia.

Veamos un ejemplo de búsqueda de palabras clave. El siguiente código crea un índice BM25 sobre una lista de elementos de texto y ejecuta una serie de búsquedas. Aquí utilizaremos la librería `txtai()`, que posee varias herramientas de búsqueda y NLP:

```
# !pip install txtai

from txtai.scoring import ScoringFactory

data = ["EE.UU. supera los 5 millones de casos confirmados de virus",
        "El último estante de hielo completamente intacto de Canadá ha colapsado",
        "Pekín moviliza embarcaciones de invasión a lo largo de la costa debido a",
        "Hombre de Maine gana $1M con un boleto de lotería de $25",
        "Obtén grandes ganancias sin trabajar, gana hasta $100,000 al día"]

# Crear un índice BM25
scoring = ScoringFactory.create({"method": "bm25", "terms": True})
scoring.index((x, texto, None) for x, texto in enumerate(data))

print("%-20s %s" % ("Consulta", "Mejor coincidencia"))
print("-" * 50)

for consulta in ("ganador de lotería", "iceberg canadiense", "número de casos",
                 "tensiones crecientes"):
    # Obtener el índice de la sección que mejor coincide con la consulta
    resultados = scoring.search(consulta, 1)
    coincidencia = data[resultados[0][0]] if resultados else "Sin resultados"
```

```
print("%-20s %s" % (consulta, coincidencia))
```

Y el resultado obtenido será:

Consulta	Mejor coincidencia
ganador de lotería	Hombre de Maine gana \$1M con un boleto de lotería de \$25
iceberg canadiense	El último estante de hielo completamente intacto de Canadá ha colapsado
número de casos	EE.UU. supera los 5 millones de casos confirmados de virus
tensiones crecientes	Pekín moviliza embarcaciones de invasión a lo largo de la costa

Observemos que cada consulta tiene un término que se encuentra en el resultado de coincidencia. Por ejemplo, "lotería" está en "Hombre de Maine gana \$1M con un boleto de lotería de \$25". También notemos que no todos los términos de la consulta son necesarios en los resultados. En general, la búsqueda por palabras clave hace un buen trabajo, ya que las métricas de tokens calculadas ayudan a producir el mejor resultado de coincidencia.

¿Pero qué pasa si queremos encontrar coincidencias conceptuales donde el término de la consulta no forma parte del resultado? Veamos a continuación:

```
from txtai.scoring import ScoringFactory

data = ["EE.UU. supera los 5 millones de casos confirmados de virus",
        "El último estante de hielo completamente intacto de Canadá ha colapsado",
        "Pekín moviliza embarcaciones de invasión a lo largo de la costa debido a las tensiones crecientes",
        "Hombre de Maine gana $1M con un boleto de lotería de $25",
        "Obtén grandes ganancias sin trabajar, gana hasta $100,000 al día"]

# Crear un índice BM25
scoring = ScoringFactory.create({"method": "bm25", "terms": True})
scoring.index((x, texto, None) for x, texto in enumerate(data))

print("%-20s %s" % ("Consulta", "Mejor coincidencia"))
print("-" * 50)

for consulta in ("historia conmovedora", "cambio climático", "salud pública",
                 "guerra", "vida silvestre", "asia", "afortunado", "ofertas engañosas")
```

```

# Obtener el índice de la sección que mejor coincide con la consulta
resultados = scoring.search(consulta, 1)
coincidencia = data[resultados[0][0]] if resultados else "Sin resultados"

print("%-20s %s" % (consulta, coincidencia))

```

Y en este caso los resultados serán:

Consulta	Mejor coincidencia
----------	--------------------

historia conmovedora	Sin resultados
cambio climático	Sin resultados
salud pública	Sin resultados
guerra	Sin resultados
vida silvestre	Sin resultados
asia	Sin resultados
afortunado	Sin resultados
ofertas engañosas	Sin resultados

Como podemos ver, nuestra búsqueda no arroja ningún resultado. Esto es, porque los textos de búsqueda no se encuentran en ninguna parte de nuestras frases. A continuación, veamos como la búsqueda semántica nos puede ayudar.

## Utilizando el sentido

### Búsqueda semántica

Las aplicaciones de búsqueda semántica comprenden el lenguaje natural e identifican resultados que tienen el mismo significado, no necesariamente las mismas palabras clave. El diagrama ilustra cómo funciona la búsqueda semántica:



Usualmente, el primer paso es utilizar un modelo de lenguaje grande (LLM) para vectorizar el contenido de entrada. La vectorización transforma las entradas en arreglos de números. Conceptos similares tendrán valores similares.

A continuación, los vectores deben almacenarse en algún lugar. Las bases de datos de vectores son sistemas que se especializan en almacenar estos arreglos numéricos y encontrar coincidencias. Por lo general, están respaldados por índices de búsqueda del vecino más cercano aproximado (ANN, por sus siglas en inglés, que ya veremos luego).

Una vez creados los índices de vectores, el último paso es la búsqueda. La búsqueda de vectores transforma una consulta de entrada en un vector y luego ejecuta una consulta para encontrar el mejor resultado, coincidiendo con la mejor coincidencia conceptual.

El siguiente código es casi idéntico al ejemplo anterior, excepto que utiliza una instancia de embeddings generada por `sentence-transformers` :

```
# !pip install txtai transformers sentence-transformers
from sentence_transformers import SentenceTransformer, util
from txtai.embeddings import Embeddings

# Datos
data = ["EE.UU. supera los 5 millones de casos confirmados de virus",
        "El último estante de hielo intacto de Canadá se ha colapsado repentinamente",
        "Pekín moviliza embarcaciones de invasión a lo largo de la costa ante el avance de las tropas ucranianas",
        "Hombre de Maine gana $1M con un billete de lotería de $25",
        "Obtén enormes ganancias sin trabajar, gana hasta $100,000 al día"]

# Cargar modelo de sentence-transformers adecuado para español. Puedes consultar la documentación para obtener más información.
model_name = "hiiamsid/sentence_similarity_spanish_es"
model = SentenceTransformer(model_name)

# Crear embeddings con txtai
embeddings = Embeddings({"method": "transformers", "path": model_name})
embeddings.index(((x, text, None) for x, text in enumerate(data)))

print("%-20s %s" % ("Consulta", "Mejor Coincidencia"))
print("-" * 50)
```

```

for consulta in ("de no creer", "cambio climático", "aumentan los contagios",
    "conflicto en asia", "afortunado", "ofertas engañosas"):

# Obtener índice de la sección que mejor coincide con la consulta
uid = embeddings.search(consulta, 1)[0][0]

print("%-20s %s" % (consulta, data[uid]))

```

Cuyo resultado al ejecutar es:

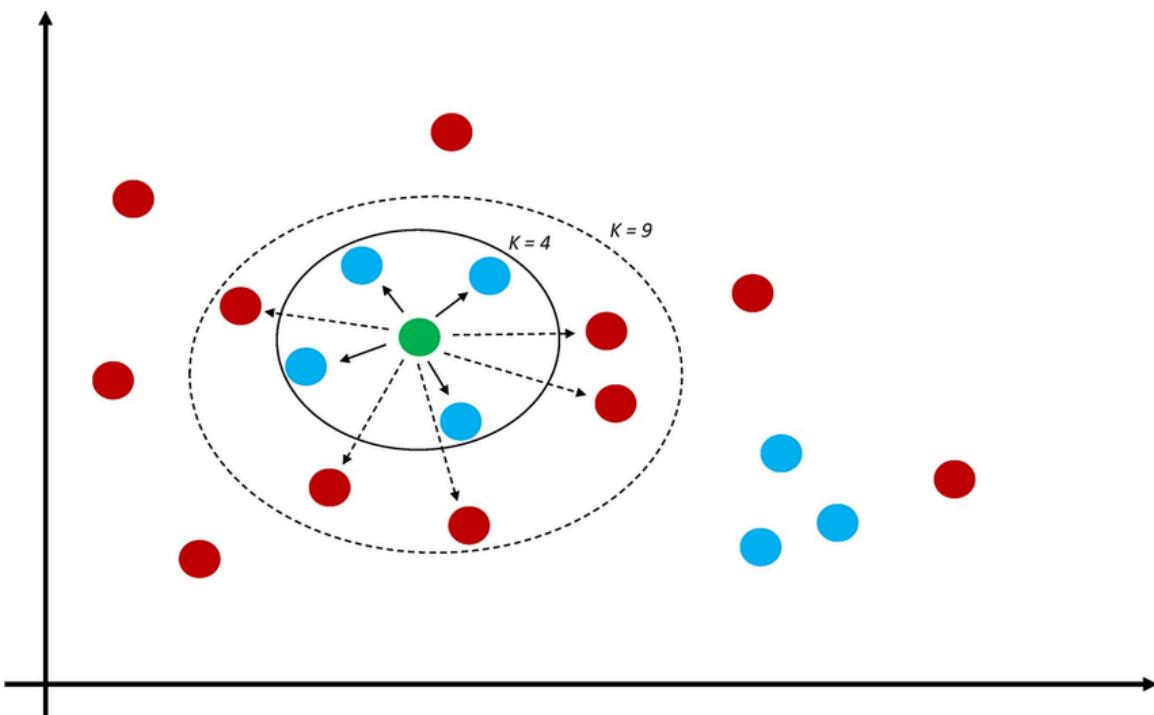
Consulta	Mejor Coincidencia
de no creer	Hombre de Maine gana \$1M con un billete de lotería de \$25
cambio climático	El último estante de hielo intacto de Canadá se ha colapsa
aumentan los contagios	EE.UU. supera los 5 millones de casos confirmados d
conflicto en asia	Pekín moviliza embarcaciones de invasión a lo largo de la c
afortunado	Hombre de Maine gana \$1M con un billete de lotería de \$25
ofertas engañosas	Obtén enormes ganancias sin trabajar, gana hasta \$100,0

Observemos las coincidencias aquí. El ejemplo anterior muestra que para casi todas las consultas, el texto real no se almacena en la lista de secciones de texto. Éste es el verdadero poder de la búsqueda semántica sobre la búsqueda basada en palabras clave.

## Estrategias de optimización de búsqueda (algoritmos)

### k-NN (k-Nearest Neighbors)

KNN, que significa K-Vectores más Cercanos en español, es un algoritmo utilizado tanto para clasificación como para regresión. Su funcionamiento se basa en la idea de que las muestras similares tienden a estar cerca unas de otras en un espacio vectorial. El siguiente gráfico, nos muestra como podemos buscar los K vectores más cercanos al vector verde, calculando las distancias entre ellos:



k-NN tiene muchas aplicaciones. En el contexto de las búsquedas semánticas, el proceso es el siguiente:

- Se calculan las distancias a todos los vectores en un conjunto de entrenamiento y se guardan.
- Se ordenan las distancias calculadas.
- Se almacenan los K vectores más cercanos.
- Se calcula la clase más frecuente mostrada por los K vectores más cercanos.



Imaginemos que tenemos un conjunto de datos muy grande. Podría ser un impedimento utilizar el método, tanto por el almacenamiento de las distancias, como también por el costo computacional de calcular y ordenar todos los valores.

Veamos un ejemplo usando [NearestNeighbors](#), que es una implementación del algoritmo k-Nearest Neighbors (k-NN) de la biblioteca scikit-learn. Además usaremos el modelo Universal Sentence Encoder desde TensorFlow Hub. *USE* es un modelo que convierte oraciones en vectores (embeddings) de alta dimensión que capturan su significado semántico:

```

import tensorflow_text
import tensorflow as tf
import tensorflow_hub as hub
from sklearn.neighbors import NearestNeighbors

# Cargar Universal Sentence Encoder
embed = hub.load("https://tfhub.dev/google/universal-sentence-encoder-mul

# Lista de oraciones
data = [
    "Hola, ¿cómo estás?",
    "Buenos días, ¿todo bien?",
    "Hola, ¿qué tal?",
    "Me gusta el NLP.",
    "El procesamiento de lenguaje natural es fascinante.",
    "KNN puede encontrar vecinos cercanos en datos."
]

# Convertir oraciones en embeddings usando USE
sentence_embeddings = embed(data).numpy()

# Entrenar el modelo k-NN
n_neighbors = 2
knn = NearestNeighbors(n_neighbors=n_neighbors, metric='cosine', algorithm='brute')
knn.fit(sentence_embeddings)

# Consulta
queries = ["Hola, ¿cómo va tu día?", "Aprendiendo procesamiento de lenguaje"]

# Convertimos las consultas en embeddings
vectorized_query = embed(queries).numpy()

# Realizamos la búsqueda de los vecinos más cercanos
distances, indices = knn.kneighbors(vectorized_query)

# Imprimimos los vecinos más cercanos y sus distancias
for i in range(len(queries)):
    print(f"Consulta: {queries[i]}")

```

```

for j in range(n_neighbors):
    print(f"Vecino {j + 1}: {data[indices[i][j]]} - Distancia: {distances[i][j]:.4f}")
    print("-" * 40)

```

Obtenemos como resultado:

```

Consulta: Hola, ¿cómo va tu día?
Vecino 1: Hola, ¿cómo estás? - Distancia: 0.1648
Vecino 2: Hola, ¿qué tal? - Distancia: 0.2543
-----
Consulta: Aprendiendo procesamiento de lenguaje natural
Vecino 1: El procesamiento de lenguaje natural es fascinante. - Distancia: 0.35
Vecino 2: Me gusta el NLP. - Distancia: 0.7226
-----
Consulta: Buenos días, ¿todo bien?
Vecino 1: Buenos días, ¿todo bien? - Distancia: 0.0000
Vecino 2: Hola, ¿cómo estás? - Distancia: 0.4263
-----
```

Se crea una instancia del modelo k-NN usando `NearestNeighbors`

- `n_neighbors=2`: Busca los 2 vecinos más cercanos.
- `metric='cosine'`: Usa la métrica del coseno para calcular la distancia (que es una buena métrica para comparar embeddings).
- `algorithm='brute'`: Usa la búsqueda exhaustiva para encontrar los vecinos más cercanos.

Cuando utilizamos la métrica `'cosine'` con `NearestNeighbors` en `scikit-learn`, la distancia que se devuelve es efectivamente la distancia del coseno. Sin embargo, es importante aclarar lo que significa esta distancia.

La similitud coseno mide el coseno del ángulo entre dos vectores. La similitud varía entre -1 y 1, donde 1 indica que los vectores son idénticos, 0 indica que son ortogonales (es decir, no relacionados) y -1 indica que apuntan en direcciones completamente opuestas.

La distancia del coseno, por otro lado, es simplemente 1 menos la similitud del coseno. Por lo tanto, varía entre 0 y 2, donde 0 indica que los vectores son idénticos y 2 indica que apuntan en direcciones completamente opuestas.

Entonces, cuando imprimimos la distancia usando `metric='cosine'`, estamos obteniendo la distancia del coseno. Una distancia de 0 indica que las dos entradas son idénticas (en términos de dirección), y valores más altos indican que son menos similares (con 2 siendo completamente disímiles).

## ANN (Approximate Nearest Neighbors)

Vecinos Más Cercanos Aproximados (ANN) se refiere a un conjunto de algoritmos que buscan encontrar puntos en un espacio que son "cercanos" a un punto dado, pero no necesariamente son los más cercanos absolutos. La palabra "aproximado" aquí es clave: a cambio de permitir una pequeña imprecisión en los resultados, estos algoritmos pueden operar mucho más rápido que las soluciones exactas.

En muchos escenarios, como sistemas de recomendación, motores de búsqueda y otros, tener una respuesta exacta no es tan crítico como tener una respuesta rápidamente. Además, en altas dimensiones, buscar el vecino más cercano exacto puede ser muy costoso en tiempo y recursos. Por lo tanto, los algoritmos ANN ofrecen un compromiso entre precisión y velocidad.

El uso más común de ANN es en sistemas de recomendación y motores de búsqueda. Por ejemplo, si tienes un vector que representa las preferencias de un usuario y quieras encontrar productos similares, puedes usar un algoritmo ANN para encontrar rápidamente productos que estén cerca en el espacio de características.

Otro uso común es en la visión por computadora, donde los algoritmos ANN pueden ayudar a identificar imágenes o partes de imágenes que son similares a una imagen dada.



Aunque los algoritmos ANN son muy rápidos, hay que tener en cuenta que son aproximados. Esto significa que, en ocasiones, pueden devolver resultados que no son exactamente los más cercanos. Sin embargo, en muchas aplicaciones prácticas, esta pequeña imprecisión es aceptable dada la gran mejora en la velocidad.

Hay una serie de técnicas y algoritmos que se han desarrollado para realizar este tipo de búsquedas. En este repositorio, se encuentra un benchmark de los

algoritmos y librerías más populares, con sus respectivas comparativas:

Con el paso del tiempo, han aparecido varios algoritmos de ANN, cada uno proporcionando un método distintivo para una búsqueda de similitud efectiva:

- **Random Projection Trees**: Estas estructuras utilizan proyecciones aleatorias para dividir los datos en una jerarquía de divisiones. Funcionan mejor en áreas con entre tres y varias dimensiones.
- **Locality-Sensitive Hashing (LSH)**: LSH hace uso de técnicas de hashing para agrupar lógicamente objetos que probablemente sean similares. Ofrece un equilibrio entre la velocidad de búsqueda y la precisión y es adecuado para datos de alta dimensión.
- **Hierarchical Navigable Small World (HNSW)**: Este enfoque más reciente crea enlaces entre puntos que están cerca en el espacio de incrustación, creando una red de puntos de datos. Se realizan compensaciones efectivas entre eficiencia y precisión.

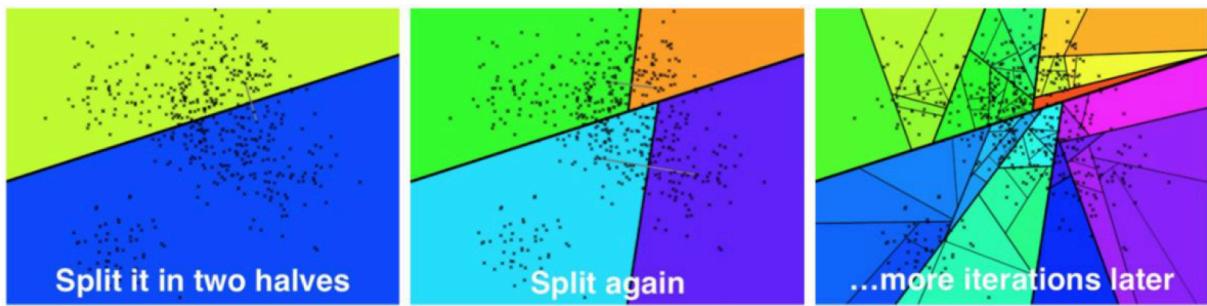
Más información sobre algoritmos ANN:

<https://towardsdatascience.com/comprehensive-guide-to-approximate-nearest-neighbors-algorithms-8b94f057d6b6>

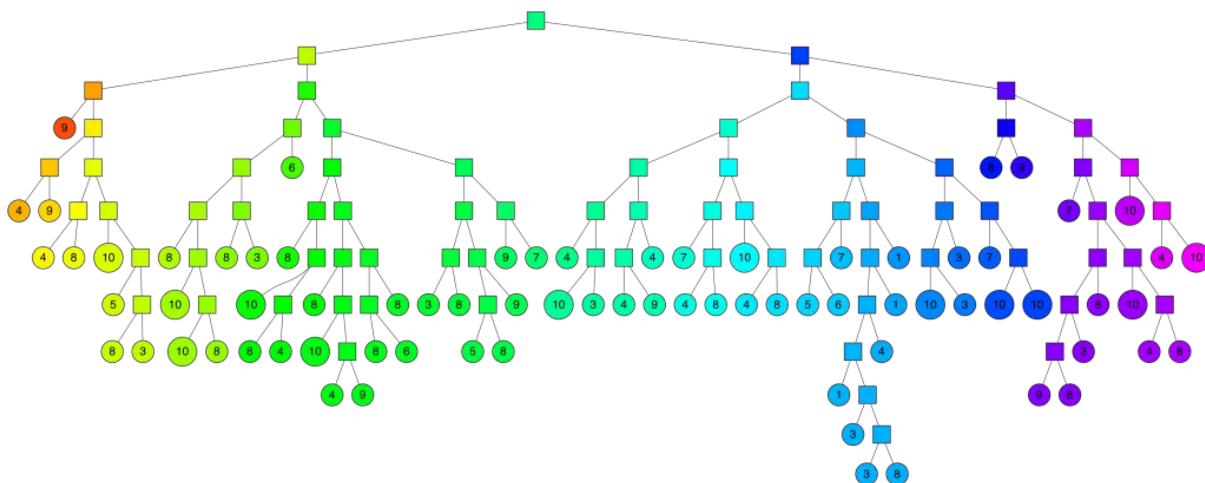
## **ANNOY (Approximate Nearest Neighbors Oh Yeah) []**

ANNOY es una librería en C++ con acceso desde Python, para buscar puntos en un espacio que están cerca de un punto de consulta dado, por ejemplo, un vector que represente una frase o palabra.

También crea grandes estructuras de datos que se mapean en la memoria para que muchos procesos puedan compartir los mismos datos. La clave del funcionamiento de ANNOY es que construye múltiples árboles binarios balanceados, en lugar de un solo árbol. Durante la construcción del árbol, ANNOY selecciona dos puntos al azar y traza un hiperplano entre ellos para dividir los puntos restantes en dos grupos. Este proceso se repite de manera recursiva. Fue desarrollado por Erik Bernhardsson en 2015, trabajando en ese momento en Spotify. ANNOY está diseñado para buscar en conjuntos de datos de 100 a 1000 dimensiones densas. Para calcular los vecinos más cercanos, divide el conjunto de puntos a la mitad y lo hace recursivamente hasta que cada conjunto tenga k elementos:



ANNOY utiliza Proyección Aleatoria ([Random Projections](#)) para construir los árboles binarios. Según este método, la probabilidad de que dos vectores estén en el mismo lado del hiperplano aleatorio, es proporcional a la distancia del coseno entre ellos.



Fuente: <https://erikbern.com/2015/10/01/nearest-neighbors-and-vector-models-part-2-how-to-search-in-high-dimensional-spaces.html>

Un algoritmo de búsqueda de vecinos más cercanos aproximados tiene como objetivo identificar puntos cuya distancia con respecto a un punto de consulta es, a lo sumo,  $c$  veces mayor que la distancia del punto de consulta a sus verdaderos vecinos más cercanos.

La principal ventaja de este enfoque radica en que, en una amplia variedad de casos, un vecino aproximado puede ser prácticamente tan útil o relevante como un vecino exacto. Esto es particularmente cierto cuando la métrica de distancia utilizada refleja adecuadamente la percepción de calidad o similitud que tiene el usuario. En tal escenario, pequeñas variaciones en la distancia resultan siendo insignificantes para el propósito final de la búsqueda.

En el contexto de la clasificación mediante algoritmos basados en vecinos más cercanos (como ANNOY), la "clase" se refiere a la asignación de una

categoría o clase a un punto en función de sus vecinos. Concretamente, un punto se categoriza en función de la clase que predomina entre sus k vecinos más cercanos. Aquí, el valor de k representa el número de vecinos considerados y es un número entero positivo que, por lo general, no es muy grande.

```
# pip install annoy

import tensorflow_text
import tensorflow as tf
import tensorflow_hub as hub
from annoy import AnnoyIndex

# Cargar Universal Sentence Encoder
embed = hub.load("https://tfhub.dev/google/universal-sentence-encoder-mul

# Lista de oraciones
sentences = [
    "Hola, ¿cómo estás?",
    "Buenos días, ¿todo bien?",
    "Hola, ¿qué tal?",
    "Me gusta el aprendizaje automático.",
    "El procesamiento de lenguaje natural es fascinante.",
    "Annoy ayuda a encontrar vecinos cercanos."
]

# Convertir oraciones en embeddings usando USE
sentence_embeddings = embed(sentences).numpy()

# Construir el índice ANNOY
dimension = sentence_embeddings.shape[1] # Asegurarse de que la dimensió
index = AnnoyIndex(dimension, 'angular')
for i, emb in enumerate(sentence_embeddings):
    index.add_item(i, emb)

arboles = 10 # Construir índice con 10 árboles
index.build(arboles )
index.save('sentences_USE.ann')
```

```

# Consulta
query = "Hola, ¿cómo va tu día?"
query_embedding = embed([query]).numpy()[0]

# Buscar las 2 oraciones más similares
nns = index.get_nns_by_vector(query_embedding, 2)
for idx in nns:
    print(sentences[idx])

# Imprime:
# Hola, ¿cómo estás?
# Hola, ¿qué tal?

```

El parámetro de `árboles` para ANNOY se refiere a la cantidad de árboles aleatorios que se construyen para indexar los puntos en el espacio. El uso de múltiples árboles mejora la precisión de la búsqueda de vecinos más cercanos aproximados a expensas de consumir más memoria. Aumentar el número de árboles también incrementa el tiempo requerido para construir el índice. No hay un número "mágico" o universalmente óptimo para todos los casos. La elección del número de árboles depende del tamaño del conjunto de datos, las dimensiones de los vectores, la memoria disponible y los requisitos específicos de precisión y velocidad de la aplicación.

## Búsqueda

Cuando hacemos la búsqueda con `get_nns_by_vector`, Annoy busca y retorna los "n" vecinos más cercanos para el vector especificado. No se limita a buscar en un sólo clúster. El objetivo de Annoy es encontrar aproximaciones rápidas de los vecinos más cercanos, independientemente de en qué clúster se encuentren. Cuando Annoy construye sus árboles (10 en nuestro caso), utiliza proyecciones aleatorias (Random Projections) para dividir el espacio y organizar los puntos en nodos. Sin embargo, cuando realizamos una consulta con `get_nns_by_vector`, el algoritmo busca en varios árboles y agrupa los resultados. Por lo tanto, incluso si los puntos están en diferentes nodos o clústeres en diferentes árboles, todavía tienen la oportunidad de ser devueltos como vecinos cercanos.

## FAISS (Facebook AI Similarity Search) []

FAISS (Facebook AI Similarity Search) es una biblioteca desarrollada por Facebook AI Research (FAIR) que ofrece soluciones eficientes para la búsqueda de similitud y la agrupación de vectores densos. Está optimizado para manejar grandes cantidades de datos y es particularmente útil para casos donde se necesitan búsquedas rápidas en grandes conjuntos de vectores.

### ¿Cómo funciona FAISS?

FAISS se explica detalladamente en el [paper "Billion-scale similarity search with GPUs"](#). Los aspectos clave son:

1. **Quantization:** FAISS utiliza la cuantificación, que implica la partición del conjunto de vectores en subconjuntos (o "celdas") y luego representa estos subconjuntos con solo un vector (o unos pocos vectores). Esto permite una reducción significativa en la cantidad de datos con los que se debe trabajar durante la búsqueda.
2. **Indexación:** FAISS permite crear [diferentes tipos de índices](#) para adaptarse a diferentes necesidades. Algunos índices son más adecuados para búsquedas exactas, mientras que otros son mejores para búsquedas aproximadas. Por ejemplo, `IndexFlatL2` es un índice simple que realiza búsquedas exactas basadas en la distancia L2, mientras que `IndexIVFFlat` es un índice que primero cuantifica los vectores y luego realiza una búsqueda aproximada. `IndexHNSWFlat` implementa el algoritmo HNSW que es particularmente efectivo para búsquedas en grandes conjuntos de datos en alta dimensión, y ofrece un buen equilibrio entre precisión y velocidad.
3. **Búsqueda:** Una vez que se ha construido el índice con los vectores de datos, se puede realizar una búsqueda para encontrar los k vecinos más cercanos de un vector de consulta. Dependiendo del tipo de índice utilizado, esta búsqueda puede ser exacta o aproximada.
4. **GPU Acceleration:** FAISS también ofrece soporte para GPUs, lo que puede acelerar aún más las operaciones de indexación y búsqueda.

### Componentes clave de FAISS:

1. **Vectores:** Son la unidad fundamental de datos en FAISS. En contextos típicos, un vector podría representar un documento, una imagen o cualquier otro tipo de dato en un espacio vectorial.

2. **Distancia:** FAISS generalmente utiliza la distancia L2 (euclíadiana) para medir similitudes, pero también admite la distancia del coseno y otras métricas.
3. **Índices:** Los índices en FAISS determinan cómo se almacenan y se buscan los vectores. FAISS tiene una variedad de tipos de índices diseñados para diferentes casos de uso.

### Ventajas de FAISS:

- **Rapidez:** FAISS está optimizado para ser extremadamente rápido, especialmente en GPUs.
- **Flexibilidad:** Ofrece una variedad de métodos de indexación, lo que permite a los usuarios elegir el compromiso adecuado entre precisión y velocidad para su aplicación particular.
- **Escalabilidad:** Puede manejar conjuntos de datos extremadamente grandes, haciendo que la búsqueda de similitud a gran escala sea práctica.

Veamos un ejemplo de como utilizar la librería `faiss` :

```
# !pip install faiss-gpu # Versión para GPU
# !pip install faiss-cpu # Versión para CPU

import tensorflow_text
import numpy as np
import tensorflow as tf
import tensorflow_hub as hub
import faiss

# Cargar Universal Sentence Encoder
embed = hub.load("https://tfhub.dev/google/universal-sentence-encoder-mul

# Lista de oraciones
sentences = [
    "Hola, ¿cómo estás?", 
    "Buenos días, ¿todo bien?", 
    "Hola, ¿qué tal?", 
    "Me gusta el aprendizaje automático.", 
    "El procesamiento de lenguaje natural es fascinante.", 
    "KNN puede encontrar vecinos cercanos en datos."
```

```

]

# Convertir oraciones en embeddings usando USE
sentence_embeddings = embed(sentences).numpy()

# Construcción del índice con FAISS
dimension = sentence_embeddings.shape[1]
index = faiss.IndexFlatL2(dimension)
index.add(sentence_embeddings)

# Consulta
queries = ["Hola, ¿cómo va tu día?", "Podemos usar FAISS para realizar búsquedas"]
query_embeddings = embed(queries).numpy()

# Búsqueda con FAISS
k = 2 # número de vecinos más cercanos que queremos
distances, indices = index.search(query_embeddings, k)

# Imprimimos los resultados
for i, query in enumerate(queries):
    print(f"Consulta: {query}")
    for j in range(k):
        print(f"{j+1}. {sentences[indices[i][j]]} - Distancia: {distances[i][j]:.4f}")
    print("-" * 40)

```

Y el resultado será:

```

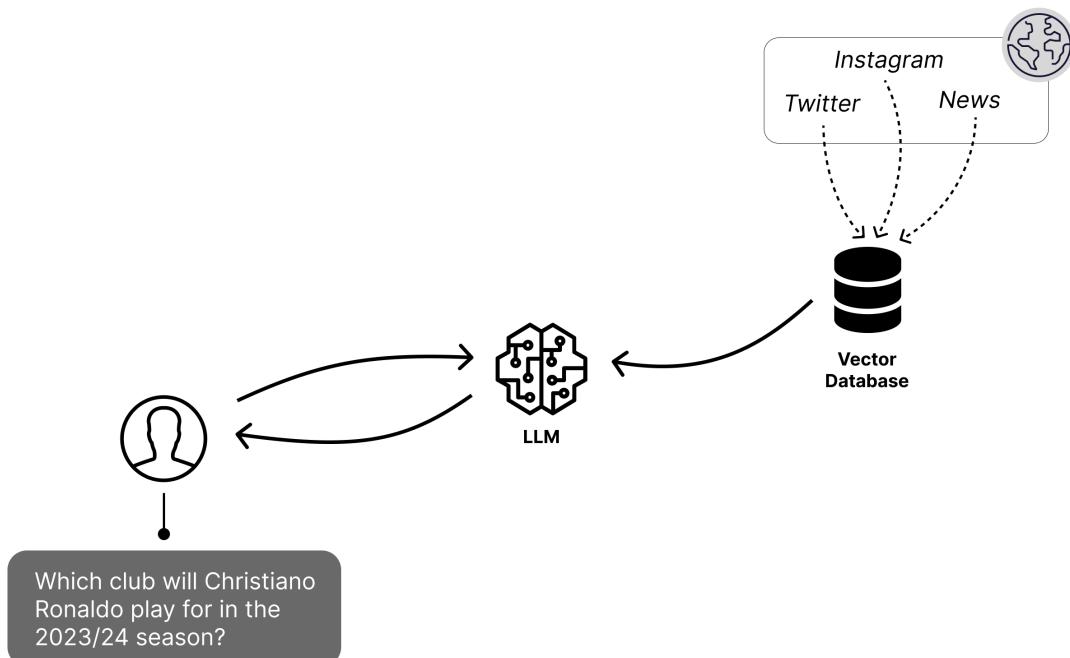
Consulta: Hola, ¿cómo va tu día?
1. Hola, ¿cómo estás? - Distancia: 0.3296
2. Hola, ¿qué tal? - Distancia: 0.5086
-----
Consulta: Podemos usar FAISS para realizar búsquedas
1. KNN puede encontrar vecinos cercanos en datos. - Distancia: 0.9976
2. Me gusta el aprendizaje automático. - Distancia: 1.1297
-----
```

## Representación

## 2. Bases de datos vectoriales

Las bases de datos de vectores hacen posible buscar y comparar rápidamente grandes colecciones de vectores. Esto es tan interesante porque los modelos de embeddings más actuales son altamente capaces de entender la semántica/significado detrás de las palabras y traducirlas en vectores. Esto nos permite comparar eficientemente las oraciones entre sí.

Para la mayoría de las aplicaciones de Modelos de Lenguaje de Gran Tamaño (LLM, por sus siglas en inglés), confiamos en esa capacidad ya que nuestro LLM nunca puede saberlo todo. Solo ve como una versión congelada del mundo, que depende del conjunto de entrenamiento con el que se entrenó el LLM.



Por lo tanto, necesitamos alimentar nuestro modelo con datos adicionales, información que el LLM no puede conocer por sí mismo. Y todo esto debe suceder durante el tiempo de ejecución de nuestra aplicación. Por lo tanto, debemos tener un proceso en marcha que decida lo más rápidamente posible con qué datos adicionales queremos alimentar nuestro modelo.

Con la búsqueda tradicional por palabras clave, nos encontramos con limitaciones, principalmente debido a dos problemas:

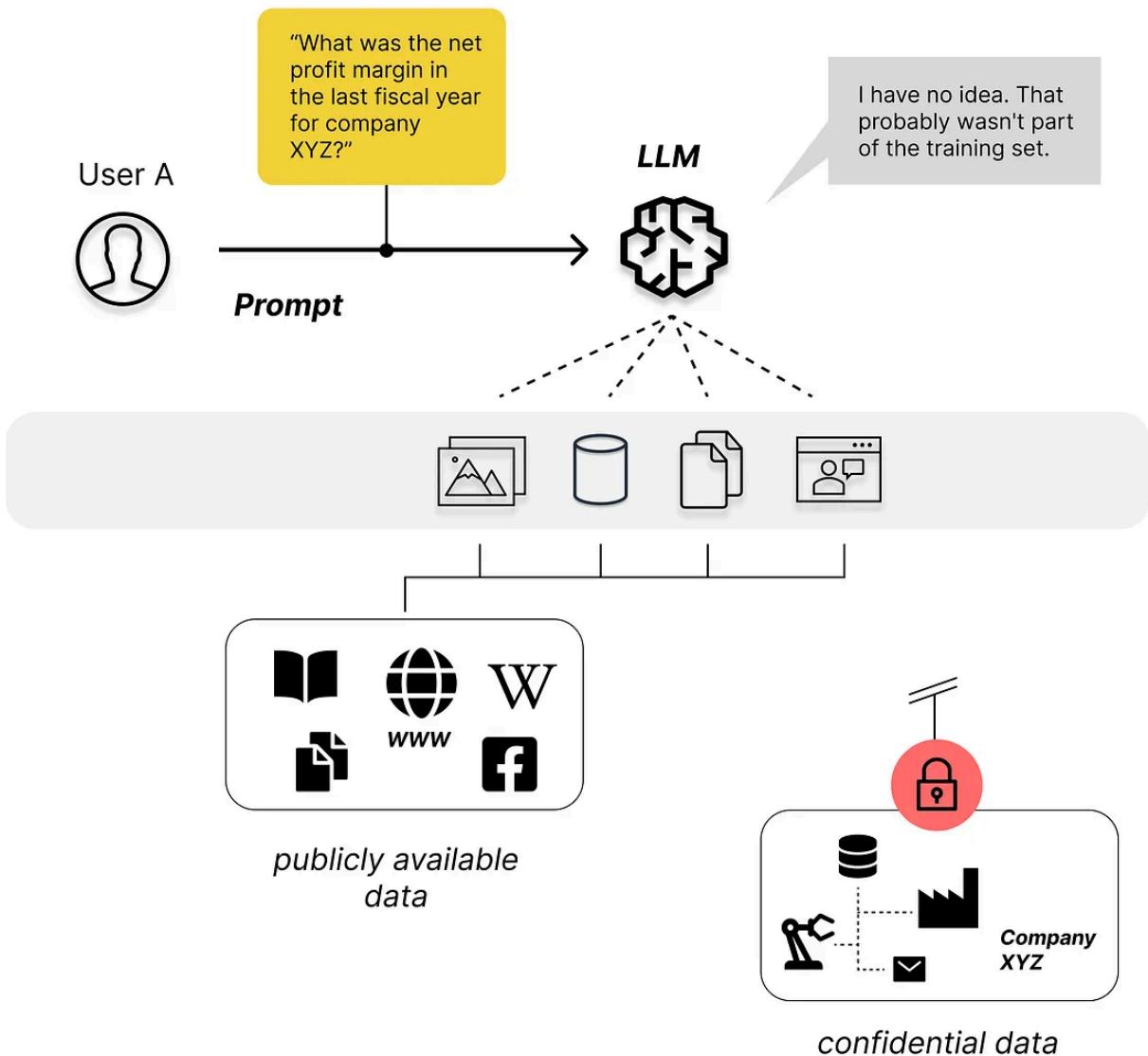
- Los idiomas son complejos. En la mayoría de los idiomas, podemos hacer más o menos la misma pregunta de 20 maneras diferentes. A menudo no es suficiente simplemente buscar palabras clave en nuestros datos. Necesitamos una forma de mapear el significado detrás de las palabras y frases para encontrar contenido relacionado con la pregunta.
- También necesitamos asegurarnos de que esta búsqueda se realice en milisegundos, no en segundos o minutos. Por lo tanto, necesitamos un paso que nos permita buscar en la colección de vectores de la manera más eficiente posible.

### **¿Cómo ayuda eso a nuestra aplicación LLM?**

Utilizamos este enfoque en muchas de nuestras aplicaciones LLM cuando los mismos LLM alcanzan los límites de su conocimiento:

Cosas que los LLM no saben de antemano:

- **Datos que son demasiado nuevos:** Artículos sobre eventos actuales, innovaciones recientes, etc. Cualquier contenido nuevo creado después de la recopilación del conjunto de entrenamiento del LLM.
- **Datos que no son públicos:** datos personales, datos internos de la empresa, datos secretos, etc.

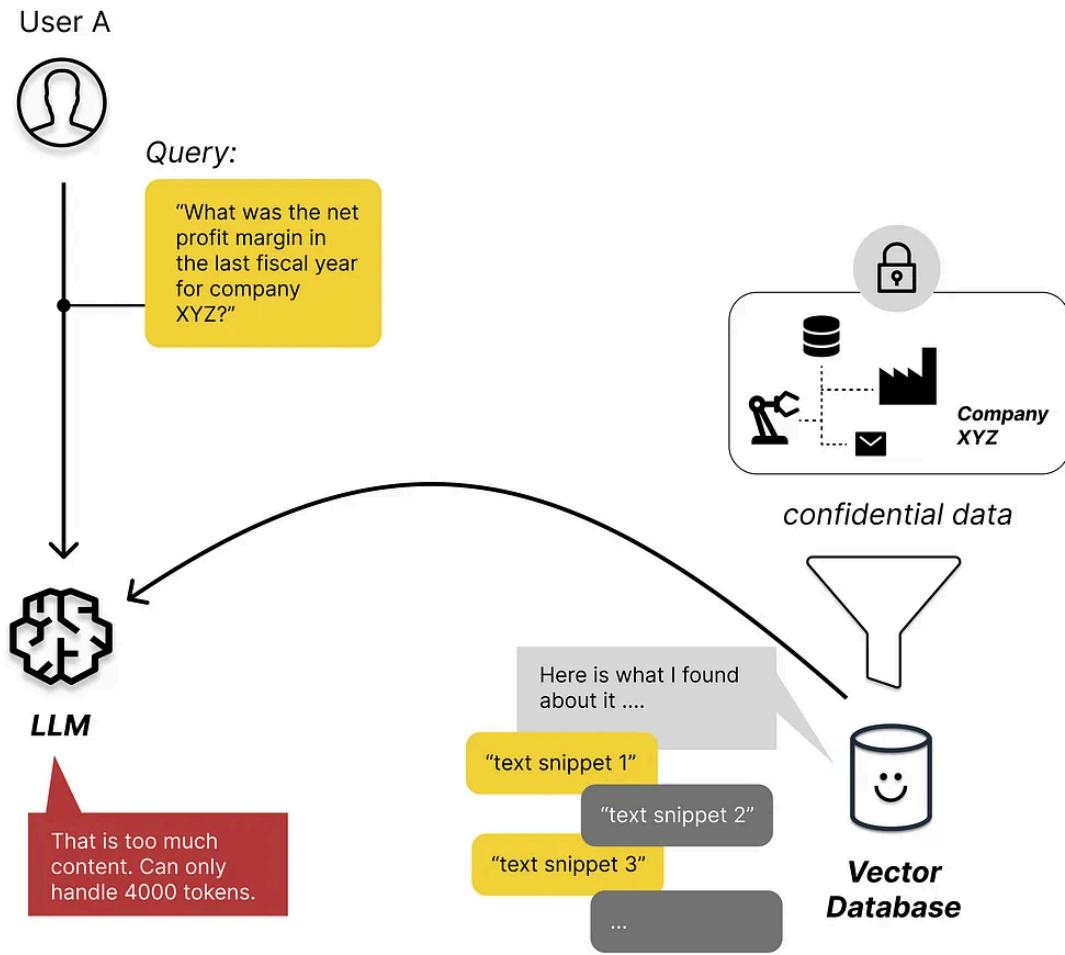


Lo que el LLM sabe y lo que no.

¿Por qué no podemos simplemente darle al modelo todos los datos que tenemos?

*Respuesta corta:* Los modelos tienen un límite, un límite de tokens.

Si no queremos entrenar o afinar el modelo, no tenemos más opción que proporcionar al modelo toda la información necesaria dentro del prompt. Tenemos que respetar los límites de tokens de los modelos.



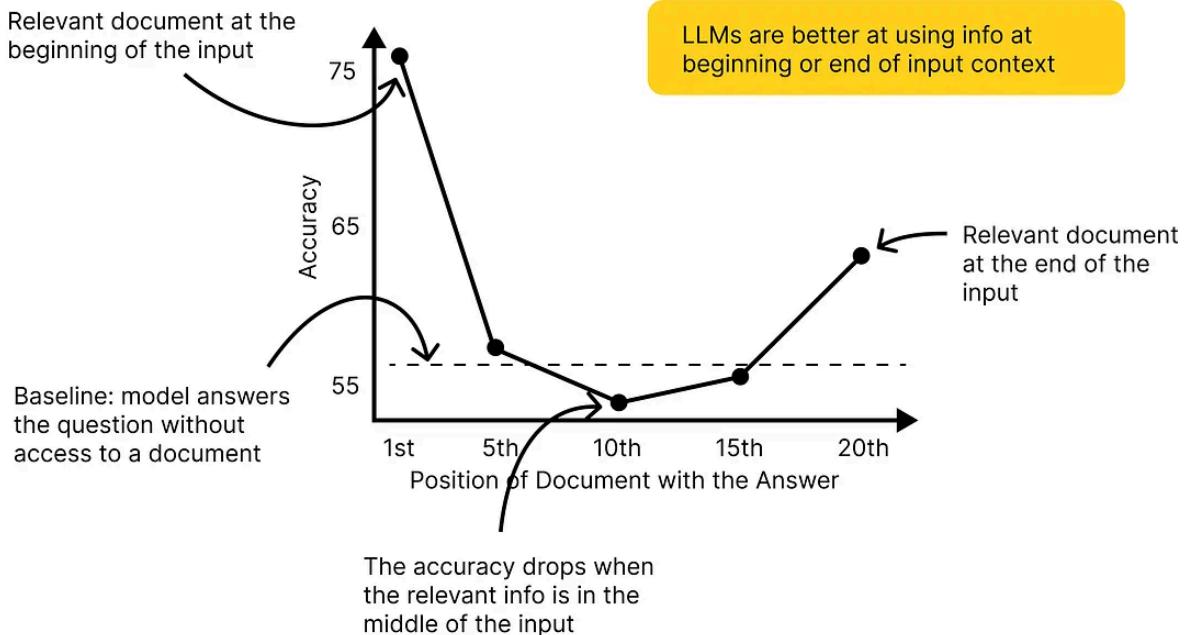
### LLMs y su Límite de Tokens

Los LLMs tienen un límite de tokens por razones prácticas y técnicas. Los modelos más recientes de OpenAI tienen un límite de tokens de alrededor de 4,000-32,000 tokens, mientras que el LLM de código abierto LLama tiene 2,048 tokens (si no se ajusta). Podemos aumentar el número máximo de tokens afinando el modelo, pero más datos no siempre es mejor. Un límite de 32,000 tokens nos permite incluir textos grandes en un prompt de una sola vez. Si esto tiene sentido o no, es otra cuestión. (Lample, 2023; OpenAI, 2023)

La calidad de los datos es más importante que la mera cantidad de datos; datos irrelevantes pueden tener un impacto negativo en el resultado.

Incluso reorganizar la información dentro del prompt puede marcar una gran diferencia en cuán precisamente los LLMs entienden la tarea. Un investigador de la Universidad de Stanford ha descubierto que cuando la información importante se coloca al principio o al final del prompt, la respuesta suele ser más precisa. Si la misma información se encuentra en medio del prompt, la

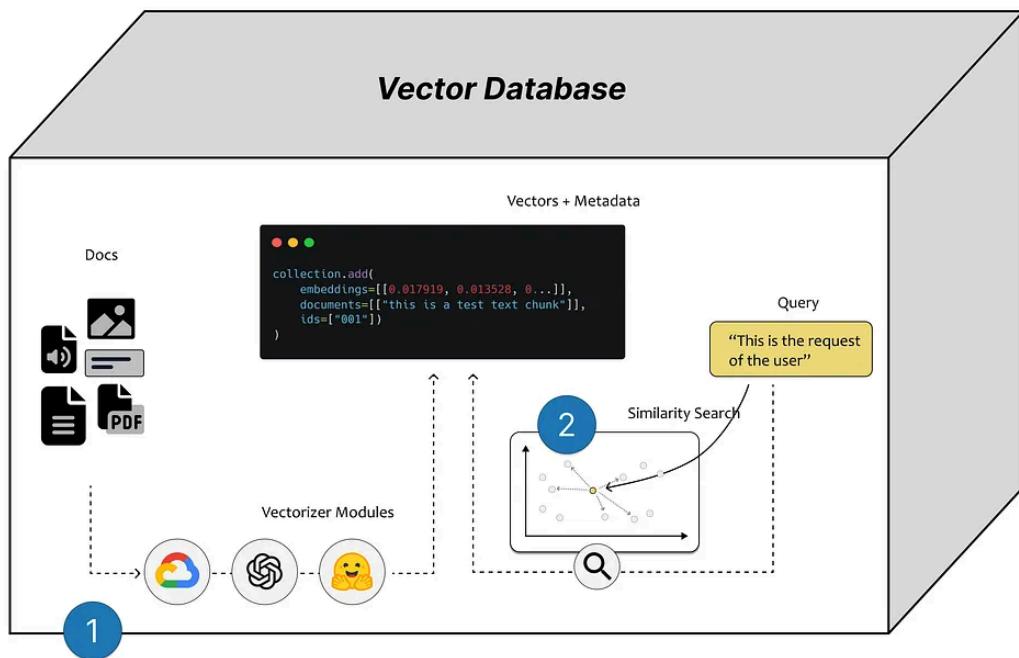
precisión puede disminuir significativamente. Es importante considerar cuidadosamente qué datos estamos proporcionando a nuestro modelo y cómo estructuramos nuestro prompt. (Liu et al., 2023; Raschka, 2023)



### Cómo los LLMs entienden los prompts

La calidad de los datos de entrada es clave para el éxito de nuestra aplicación LLM, por lo que es esencial implementar un proceso que identifique con precisión el contenido relevante y evite añadir demasiados datos innecesarios. Para garantizar esto, debemos usar procesos de búsqueda efectivos para resaltar la información más relevante.

¿Cómo nos ayudan exactamente las bases de datos vectoriales a hacer esto? Las bases de datos vectoriales están compuestas por varias partes que nos ayudan a encontrar rápidamente lo que buscamos. La indexación es la parte más importante y se hace solo una vez, cuando insertamos nuevos datos en nuestro conjunto de datos. Después de eso, las búsquedas son mucho más rápidas, ahorrandonos tiempo y esfuerzo.



1

**Text/Images/Audio to Vectors**  
*using Embeddings Models*

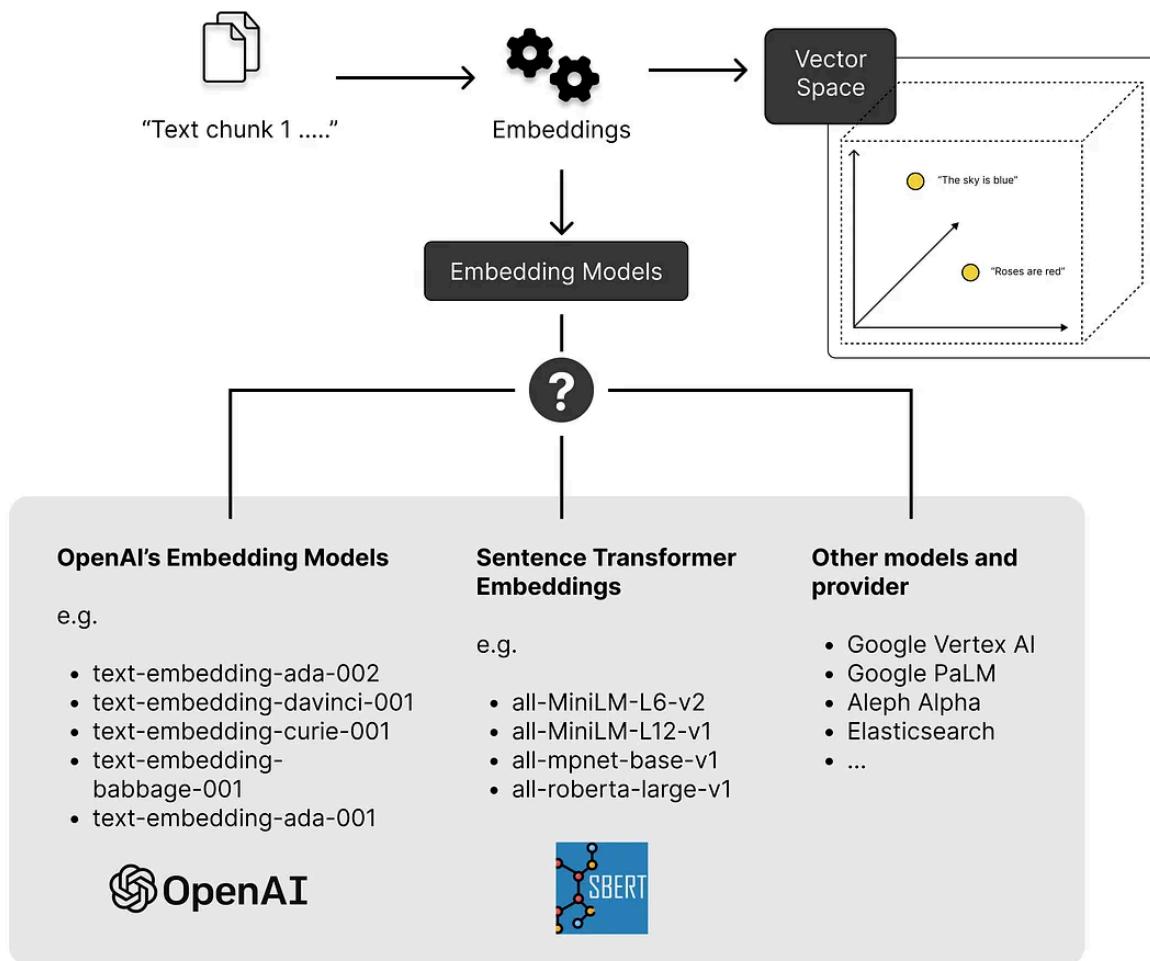
2

**Speed up similarity search**  
*using Approximate Nearest Neighbor Techniques*

Vector Databases and their components

Para alimentar nuestra base de datos vectorial, primero debemos convertir todo nuestro contenido en vectores. Como se describe a continuación, podemos usar modelos de embeddings para eso. Simplemente porque es más conveniente, a menudo usamos uno de los servicios listos para usar de OpenAI, Google y compañía.

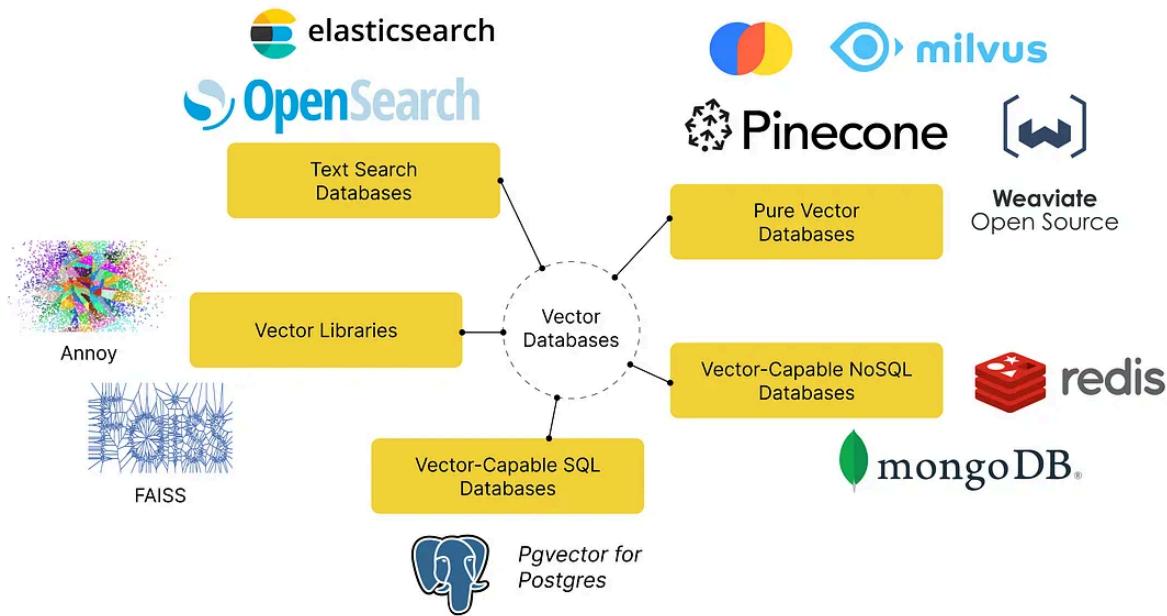
En la imagen de abajo podemos ver algunos modelos de embeddings que podemos elegir:



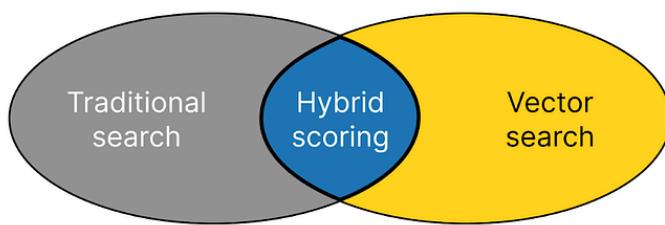
## Tipos de bases de datos

Las bases de datos vectoriales vienen en diferentes formas. Distinguimos entre:

- Bases de datos vectoriales puras
- Capacidades extendidas en bases de datos SQL, NoSQL o de búsqueda de texto
- Librerías vectoriales simples



Las bases de datos de búsqueda de texto pueden buscar en grandes cantidades de texto palabras o frases específicas. Recientemente, algunas de estas bases de datos han comenzado a utilizar la búsqueda vectorial para mejorar aún más su capacidad de encontrar lo que estamos buscando. Por ejemplo, Elasticsearch propone tanto la búsqueda tradicional como la búsqueda vectorial para crear un sistema de 'Puntuación Híbrida', brindándote los mejores resultados de búsqueda posibles.



Elasticsearch: Combinar la búsqueda tradicional con la búsqueda vectorial mejora los resultados de búsqueda.

La búsqueda vectorial también está siendo adoptada gradualmente por más y más bases de datos SQL y NoSQL como Redis, MongoDB o Postgres. Pgvector, por ejemplo, es la búsqueda de similitud vectorial de código abierto para Postgres:

- búsqueda de vecinos más cercanos exactos y aproximados

- distancia L2, producto interno y distancia del coseno

Para proyectos más pequeños, las librerías vectoriales son una excelente opción y proporcionan la mayoría de las características necesarias.

Facebook AI Research lanzó una de las primeras librerías vectoriales, FAISS, en 2017. FAISS es una librería para buscar y agrupar eficientemente vectores densos, y puede manejar conjuntos de vectores de cualquier tamaño, incluso aquellos que no caben en memoria. Está escrito en C++ y viene con un wrapper para Python, lo que facilita su integración en el mundo del Data Science.



Chroma, Pinecone, Weaviate, por otro lado, son bases de datos vectoriales puras que pueden almacenar tus datos vectoriales y ser buscadas como cualquier otra base de datos. Veremos cómo configurar una base de datos vectorial con Chroma y cómo llenarla con datos vectoriales. Si buscamos una solución rápida, bibliotecas vectoriales como FAISS pueden ayudarnos a comenzar fácilmente con todos los métodos de indexación necesarios.

### Características principales

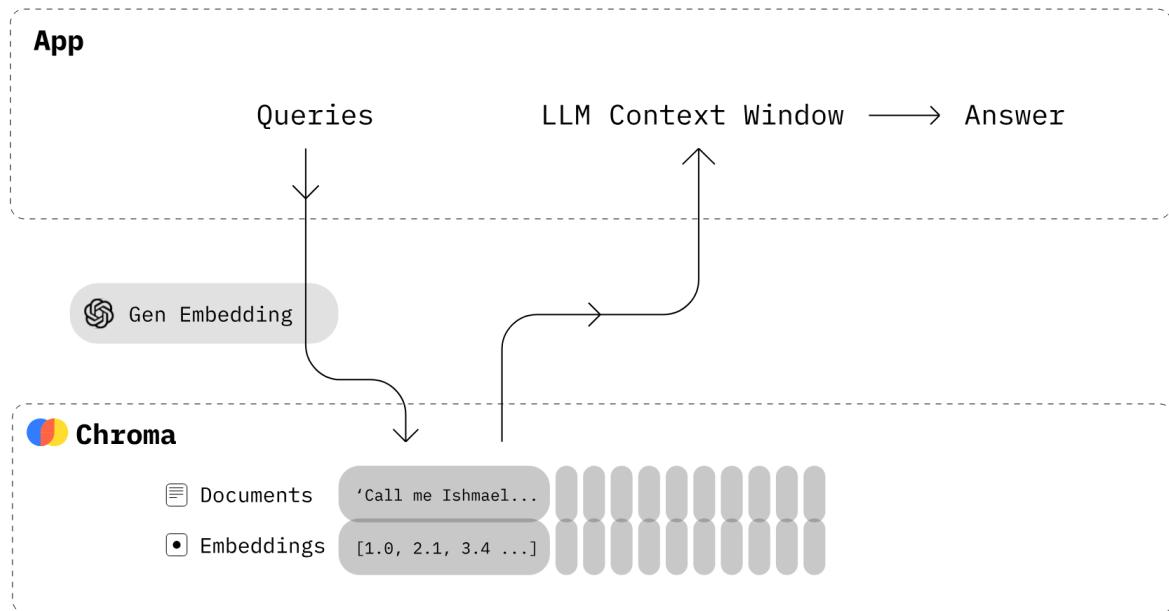
Aquí hay algunas características principales de las bases de datos vectoriales:

- 1. Búsqueda de Vecinos Más Cercanos (NN, por sus siglas en inglés):** Las bases de datos vectoriales permiten buscar los vectores más cercanos a un vector dado en términos de distancia (por ejemplo, distancia euclídea o coseno). Esto es fundamental para encontrar elementos similares en la base de datos. Pueden soportar métodos KNN como ANN.
- 2. Alta Dimensionalidad:** Estas bases de datos están optimizadas para manejar vectores en espacios de alta dimensión, lo cual es común en modelos de incrustación.

3. **Indexación Eficiente:** Para acelerar las búsquedas, estas bases de datos utilizan técnicas especiales de indexación, como árboles de búsqueda de vecinos más cercanos o hash de sensibilidad local.
4. **Escalabilidad:** Son capaces de manejar grandes conjuntos de datos y ofrecer tiempos de respuesta rápidos incluso cuando el volumen de datos es masivo.
5. **Soporte para Distintas Métricas:** Además de la distancia euclíadiana, muchas bases de datos vectoriales soportan otras métricas, como la distancia del coseno, para calcular similitudes.
6. **Integración con Modelos de Embeddings:** Algunas bases de datos vectoriales se integran con plataformas populares de aprendizaje automático, facilitando la conversión y carga de embeddings directamente en la base de datos.
7. **Persistencia y Seguridad:** Al igual que otras bases de datos, las bases de datos vectoriales ofrecen características de persistencia y opciones para respaldar, restaurar y proteger los datos.
8. **Distribución y Paralelización:** Algunas bases de datos vectoriales avanzadas son distribuidas y permiten paralelizar las operaciones de búsqueda para acelerar las consultas en grandes conjuntos de datos.
9. **Flexibilidad:** Muchas de estas bases de datos permiten combinar búsquedas vectoriales con otras operaciones típicas de bases de datos, como filtros, agregaciones y búsquedas basadas en texto.
10. **Aplicaciones Diversas:** Aunque son populares en el procesamiento de lenguaje natural, las bases de datos vectoriales también son útiles en otros campos, como la visión por computadora, la biología computacional y la recomendación de productos.

## ChromaDB

([\)](#) es una base de datos vectorial de código abierto diseñada para almacenar embeddings vectoriales y desarrollar y construir aplicaciones de modelos de lenguaje de gran tamaño. La base de datos facilita el almacenamiento de conocimientos, habilidades y hechos para aplicaciones LLM.



El diagrama anterior muestra el funcionamiento de ChromaDB cuando se integra con cualquier aplicación LLM. ChromaDB nos ofrece una herramienta para realizar las siguientes funciones:

1. Almacenar embeddings y sus metadatos con identificadores.
2. Incrustar documentos y consultas.
3. Buscar embeddings.

ChromaDB es sencilla de usar y configurar con cualquier aplicación impulsada por LLM. Está diseñado para aumentar la productividad del desarrollador, convirtiéndolo en una herramienta amigable para los desarrolladores (<https://docs.trychroma.com/api-reference>).

ChromaDB trabaja con “Colecciones”, que permiten generar un espacio de trabajo para almacenar nuestros datos:

```
# import chromadb and create client
import chromadb

client = chromadb.Client()
collection = client.create_collection("my-collection")
```

En el código anterior, hemos instanciado el objeto cliente para crear la colección "my-collection" en la carpeta del repositorio.

La colección es donde se almacenan los embeddings, documentos y cualquier metadato adicional para consultar más tarde en diversas aplicaciones.

### Agregar documentos a la colección:

```
# agregar los documentos en la base de datos
collection.add(
    documents=["Este es un documento sobre gatos", "Este es un documento sobre coches"],
    metadatas=[{"categoria": "animal"}, {"categoria": "vehículo"}, {"categoria": "otro"}, {"categoria": "mascotas"}],
    ids=["id1", "id2", "id3"]
)
```

Ahora, hemos añadido algunos documentos de muestra junto con metadatos e IDs para almacenarlos de manera estructurada.

ChromaDB almacenará los documentos de texto y manejará la tokenización, vectorización e indexación automáticamente sin ningún comando adicional. Por defecto, usará el modelo [all-MiniLM-L6-v2](#) de 384 dimensiones para vectorizar el texto.

### Consultar la base de datos de la colección

```
# realiza una consulta para buscar datos en la base
results = collection.query(
    query_texts=["vehículo"],
    n_results=1
)
results
```

Nuestra salida será:

```
{'ids': [['id2']],
'distances': [[1.417740821838379]],
'metadatas': [[{'categoria': 'vehículo'}]],
'embeddings': None,
'documents': [['Este es un documento sobre autos']]}
```

En el ejemplo anterior, realizamos la búsqueda utilizando un texto especificado en `query_texts`. Supongamos que quisiéramos realizar búsquedas, donde nuestra query sea un vector de embeddings, creado con un modelo pre-entrenado

ajeno a ChromaDB. Podríamos hacerlo del siguiente modo:

```
import tensorflow_text
import tensorflow_hub as hub
import chromadb

# Cargar Universal Sentence Encoder
embed = hub.load("https://tfhub.dev/google/universal-sentence-encoder-mul

# Configuración inicial de ChromaDB
client = chromadb.Client()
collection = client.get_or_create_collection("all-my-documents")

textos = [
    "Me encanta la inteligencia artificial.",
    "El procesamiento de lenguaje natural es un subcampo de la IA.",
    "ChromaDB permite almacenar y buscar vectores eficientemente.",
    "Los embeddings semánticos representan el significado de un texto."
]

ids_textos = [f"doc{i}" for i in range(1, len(textos)+1)]
fuentes = ["fuente1", "fuente2", "fuente3", "fuente4"] # Ejemplo de metadatos

# Calcular embeddings para los documentos
embeddings = embed(textos).numpy().tolist() # Convertir a lista para que sea
collection.add(
    documents=textos,
    metadatas=[{"source": fuente} for fuente in fuentes],
    ids=ids_textos,
    embeddings=embeddings
)

consulta = "¿Qué es el procesamiento de lenguaje natural?"
embedding_consulta = embed([consulta]).numpy().tolist()

results = collection.query(
```

```

query_embeddings=embedding_consulta, # Aquí pasamos el embedding de la consulta
n_results=3 # Traemos los 3 resultados más cercanos
)

# Imprimir los documentos y sus distancias
for id, doc, distancia in zip(results["ids"], results["documents"], results["distances"]):
    print(f"IDs: {id}")
    print(f"Documento: {doc}")
    print(f"Distancia: {distancia}")
    print('-' * 40)

# Imprime
# IDs: ['doc2', 'doc4', 'doc1']
# Documento: ['El procesamiento de lenguaje natural es un subcampo de la IA']
# Distancia: [0.4458719789981842, 0.7879210114479065, 1.03930485248565]

```

En el caso anterior, especificamos nuestro vector de consulta en `query_embeddings`. Previamente debemos calcular los embeddings de nuestra frase de consulta. Tal como usamos el modelo `universal-sentence-encoder-multilingual/3`, podríamos haber aplicado muchos otros modelos pre-entrenados en diferentes lenguajes o datasets.

## Otras bases de datos vectoriales

**Pinecone** (): <https://www.pinecone.io/learn/vector-database/>

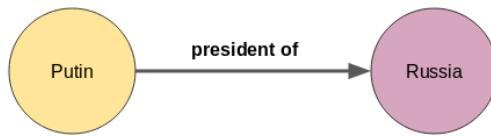
**Qdrant** (): <https://cloud.qdrant.io/>

**Milvus** (): <https://milvus.io/>

## 3. Knowledge graphs (grafos de conocimiento)

### ¿Qué es un grafo de conocimiento?

Para entender mejor los grafos de conocimiento, comenzemos entendiendo su unidad básica, es decir, un "hecho" (fact). Un hecho es la pieza más básica de información que puede ser almacenada en un KG.



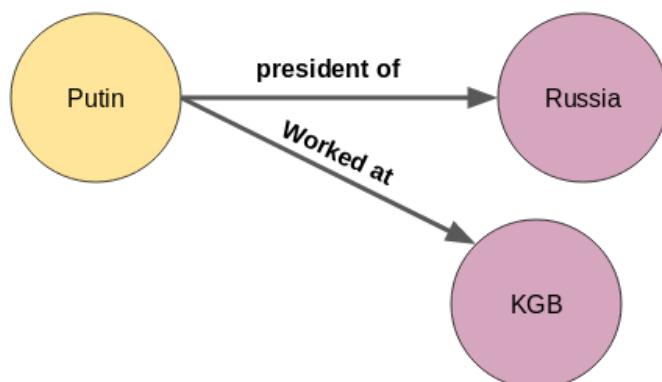
Los hechos pueden ser representados en forma de tríadas de cualquiera de las maneras:

- HRT: <cabeza, relación, cola>
- SPO: <sujeto, predicado, objeto>

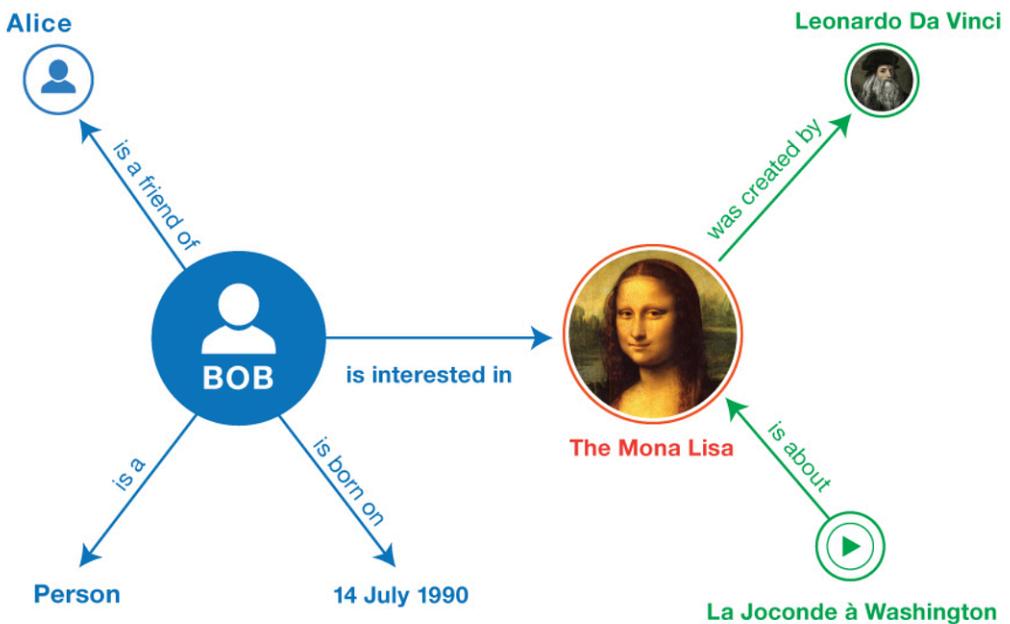
Los hechos contienen 3 elementos (por eso también se les llama tríadas) que pueden ayudar con la representación intuitiva de KG como un grafo:

- Cabeza o cola: estas son entidades que son objetos del mundo real o conceptos abstractos que se representan como nodos
- Relación: estas son las conexiones entre entidades que se representan como aristas.

Los KG, pueden crecer en cantidad de nodos, de modo que podemos fácilmente agregar más hechos o relaciones:



A continuación se muestra un ejemplo de un KG más grande, donde aparecen más sujetos. Un ejemplo de hecho podría ser <BoB, está\_interesado\_en, La\_Mona\_Lisa>. Podemos ver que el KG no es más que una colección de múltiples hechos de este tipo.



No hay limitaciones en el tipo de dato de los hechos almacenados en KG. Como se muestra en el ejemplo anterior, tenemos personas (Bob, Alice, etc.), pinturas (Mona Lisa), fechas, etc., representadas como nodos en el KG.

### ¿Por qué un grafo de conocimiento?

Esta es la primer pregunta, y un interrogante válido, que cualquiera haría al ser introducido a KG. Trataremos de repasar algunos puntos en los que comparamos KG con grafos normales e incluso otras formas de almacenar información. El objetivo es resaltar las principales ventajas de usar KG.

### Comparado con Grafos Normales

- Datos heterogéneos: admite diferentes tipos de entidades (persona, fecha, pintura, etc.) y relaciones (gusta, nacido en, etc.).
- Modelar información del mundo real: más cercano al modelo mental de nuestro cerebro del mundo (representa información como lo hace cualquier humano).
- Realizar razonamiento lógico: recorrer los grafos en un camino para hacer conexiones lógicas (el padre de A es B y el padre de B es C, por lo tanto, C es el abuelo de A).

### Comparado con otros tipos de almacenamiento

- **Representación estructurada:** muy distante de representaciones no estructuradas como datos de texto.

- **Elimina redundancia:** en comparación con datos tabulares, no es necesario agregar columnas o filas mayormente vacías para añadir nuevos datos.
- **Consultar información compleja:** mejor que SQL para datos donde la relación importa más que los puntos de datos individuales (por ejemplo, en caso de que tengas que hacer muchas declaraciones JOIN en una consulta SQL, lo cual es inherentemente lento).

### **¿Cómo usar KG?**

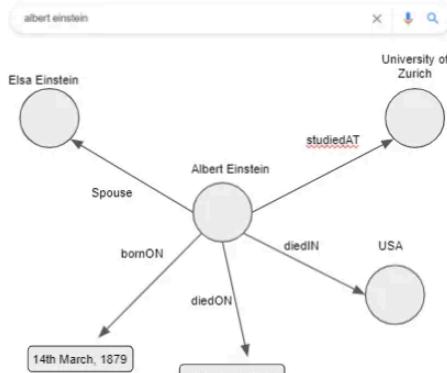
Los grafos de conocimiento pueden ser utilizados para un gran número de tareas, ya sea para razonamiento lógico, recomendaciones explicables, análisis complejos o simplemente como una mejor forma de almacenar información. Hay dos ejemplos muy interesantes que discutiremos brevemente.

### **Panel de Conocimiento de Google (Google Knowledge Panel)**

Al consultar a Google sobre una persona famosa, ubicación o concepto, devuelve un panel de conocimiento a la derecha. El panel contiene una amplia variedad de información (descripción, educación, nacimiento, fallecimiento, citas, etc.) y, curiosamente, en diferentes formatos: (texto, imagen, fechas, números, etc.).

Toda esta información puede ser almacenada en un KG y a continuación se muestra un ejemplo de ello. Esto demuestra lo fácil que es almacenar información y también destaca lo intuitivo que es simplemente leer y comprender el hecho desde un KG.

### Google Knowledge Panel



**Albert Einstein**  
Theoretical physicist

Albert Einstein was a German-born theoretical physicist, widely acknowledged to be one of the greatest physicists of all time. Einstein is known widely for developing the theory of relativity, but he also made important contributions to the development of the theory of quantum mechanics. [Wikipedia](#)

**Born:** 14 March 1879, Ulm, Germany  
**Died:** 18 April 1955, Penn Medicine Princeton Medical Center, New Jersey, United States  
**Spouse:** Elsa Einstein (m. 1919–1936), Mileva Marić (m. 1903–1919)  
**Education:** University of Zürich (1905), ETH Zürich (1896–1900), [MORE](#)

**Books**  
Relativity : the special a... 1916 | The World As I See It 1950 | Out of My Later Years 1950 | The Evolution of Physics 1938 | View 35+ more

**Quotes**  
Imagination is more important than knowledge.  
If you can't explain it simply, you don't understand it well enough.  
Life is like riding a bicycle. To keep your balance you must keep moving.

**People also search for**  
Eduard Einstein | Isaac Newton | Elsa Einstein | Stephen Hawking | View 15+ more

Ejemplo de panel de conocimiento basado en un grafo de conocimiento utilizado por Google. [Derecha] el panel real mostrado por Google cuando buscas a Einstein. [Izquierda] recreación de cómo podríamos almacenar información similar en un KG.

## Sistema de recomendación

Los algoritmos clásicos consideraban las interacciones usuario-ítem para generar recomendaciones. Con el tiempo, los algoritmos más recientes, comenzaron a considerar información adicional sobre el usuario y los ítems para mejorar las recomendaciones.

A continuación, podemos ver un KG (Knowledge Graph) de películas, que no solo contiene conexiones usuario-ítem (aquí persona-películas) sino también interacciones usuario-usUARIO y atributos de ítems. La idea es que, proporcionando toda esta información adicional, podemos hacer sugerencias mucho más precisas e informadas. Sin entrar en el algoritmo exacto, razonalicemos qué recomendaciones podrían generarse.

"Avatar" podría ser recomendada a,

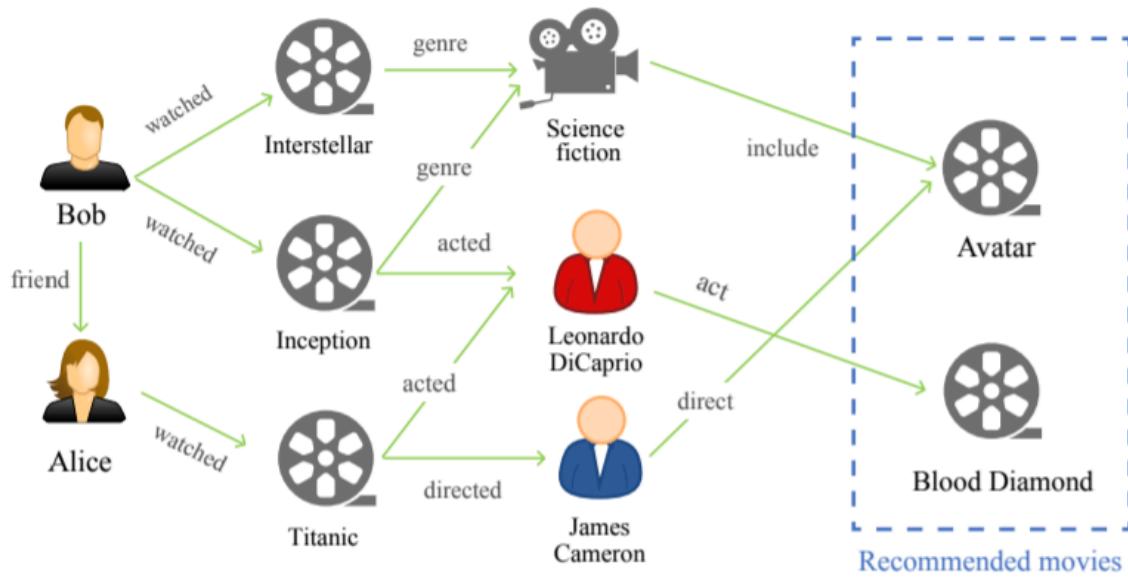
- Bob: ya que pertenece al género de ciencia ficción, al igual que Interstellar e Inception (que Bob ya ha visto)

- Alice: ya que está dirigida por James Cameron (Titanic)

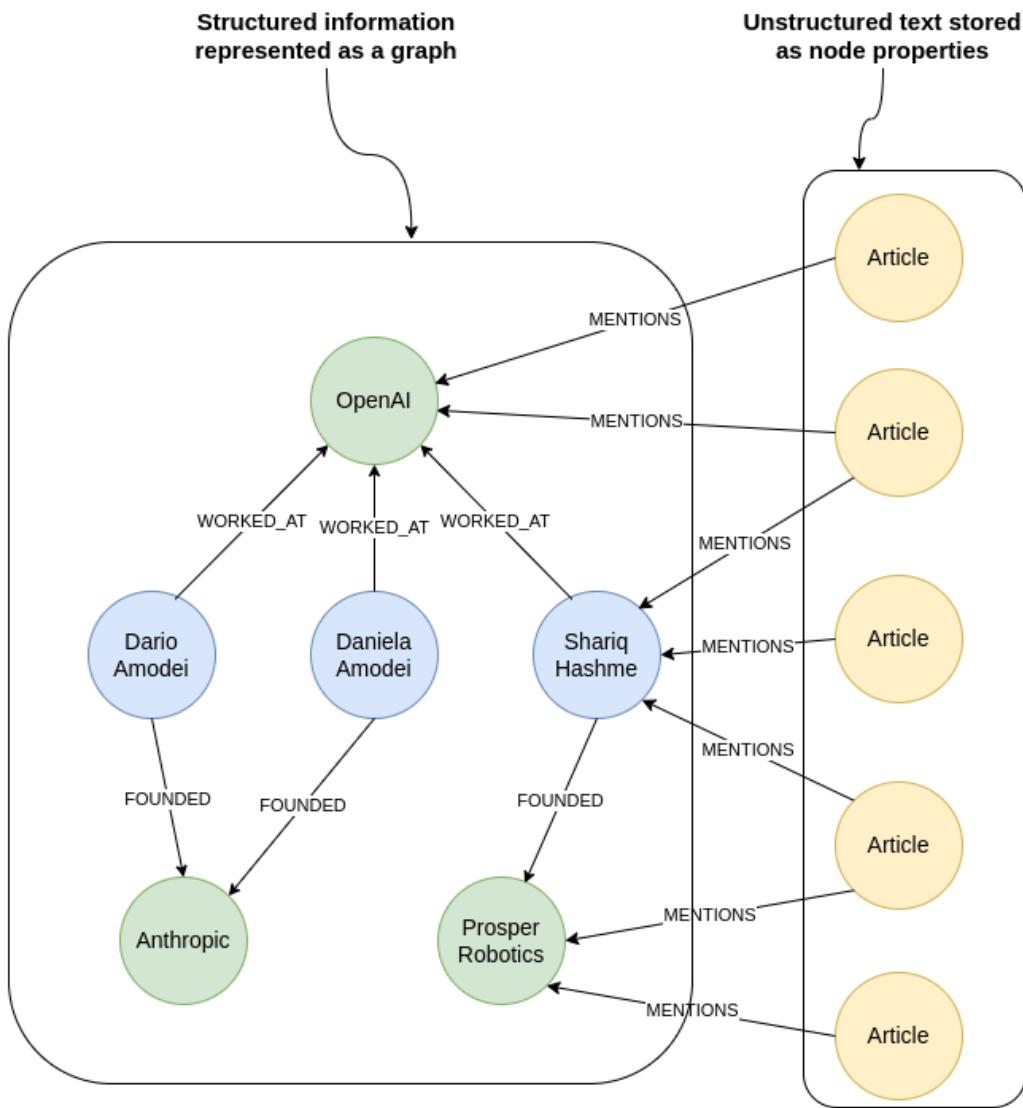
"Diamante de Sangre" podría ser recomendada a,

- Bob: ya que DiCaprio actuó en Inception también

Este simple ejercicio mental debería mostrar cómo muchas interacciones del mundo real pueden ser fácilmente representadas en forma de hechos usando KG. Y luego podemos aprovechar los algoritmos basados en KG para un caso de uso específico, como generar recomendaciones.:



Las bases de datos de conocimientos, si bien representan información estructurada, puede combinarse también con datos no estructurados. En la figura siguiente se representa este caso:



Un nodo del grafo, podría incluir información no estructurada, como un paper o artículo, o incluso un nodo podría contener un puntero o una etiqueta para obtener información de alguna fuente externa. Los nodos nos aportan esta flexibilidad, aunque requiere del apoyo de bases de datos de grafos que permitan esas capacidades.

## Grafo de Conocimiento en la práctica

Repasaremos algunos KG de código abierto y fácilmente disponibles en la web. En algunos casos, es posible que incluso queramos crear nuestro propio KG, así que también discutiremos algunos puntos con respecto a ello. Luego entenderemos rápidamente algunas reglas y formas en que un KG puede ser estructurado al discutir la ontología de KG. Finalmente, discutiremos las bases

de datos de alojamiento de KG y aprenderemos cómo podemos consultar (obtener hechos de) un KG.

### Grafos de Conocimiento de código abierto

Aunque existen varios KG de tamaño pequeño y específicos de dominio, por otro lado, también tenemos muchos KG de gran tamaño y agnósticos al dominio que contienen hechos de todos tipos y formas. Algunos de los grafos de conocimiento de código abierto más famosos son:

- **DBpedia**: es un esfuerzo comunitario y colaborativo para extraer contenido estructurado de la información presente en varios proyectos de Wikimedia.
- **Freebase**: una base de datos masiva, editada colaborativamente de datos interconectados. Promocionada como “una base de datos compartida abiertamente del conocimiento mundial”. Fue comprada por Google y utilizada para alimentar su propio KG. En 2015, finalmente se discontinuó.
- **OpenCyc**: es una puerta de entrada a Cyc, una de las bases de conocimiento general y motores de razonamiento de sentido común más completos del mundo.
- **Wikidata**: es una base de datos gratuita, colaborativa y multilingüe, que recopila datos estructurados para brindar soporte a proyectos de Wikimedia.
- **YAGO**: una base de conocimiento semántica enorme, derivada de Wikipedia, WordNet y GeoNames.

	DBpedia	Freebase	OpenCyc	Wikidata	YAGO
Number of triples	411 885 960	3 124 791 156	2 412 520	748 530 833	1 001 461 792
Number of classes	736	53 092	116 822	302 280	569 751
Number of relations	2819	70 902	18 028	1874	106
No. of unique predicates	60 231	784 977	165	4839	88 736
Number of entities	4 298 433	49 947 799	41 029	18 697 897	5 130 031
Number of instances	20 764 283	115 880 761	242 383	142 213 806	12 291 250
Avg. number of entities per class	5840.3	940.8	0.35	61.9	9.0
No. of unique subjects	31 391 413	125 144 313	261 097	142 278 154	331 806 927
No. of unique non-literals in object position	83 284 634	189 466 866	423 432	101 745 685	17 438 196
No. of unique literals in object position	161 398 382	1 782 723 759	1 081 818	308 144 682	682 313 508

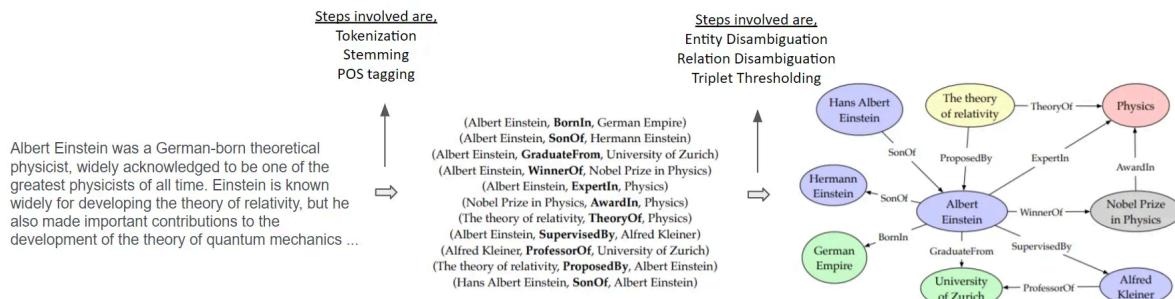
Linked Data Quality of DBpedia, Freebase, OpenCyc, Wikidata, and YAGO  
(<https://www.semantic-web-journal.net/system/files/swj1366.pdf>)

### Creación de un grafo de conocimiento personalizado

A pesar de tener varios KG de código abierto, es posible que tengamos la necesidad de crear un KG específico para nuestro caso de uso. En ese

escenario, nuestros datos base (desde los cuales queremos crear el KG) podrían ser de varios tipos: tabulares, gráficos o bloques de texto. Cubriremos algunos pasos sobre cómo crear un KG a partir de datos no estructurados como el texto, ya que es relativamente más fácil convertir datos estructurados en KG utilizando conocimientos de dominio mínimos y programación. El proceso completo se puede dividir en dos pasos,

- **Creación de hechos:** este es el primer paso donde analizamos el texto (oración por oración) y extraemos hechos en formato de tríada como <H, R, T> (Head, Relation, Tail). Dado que estamos procesando texto, podemos aprovechar pasos de preprocesamiento como tokenización, derivación o lematización, etc., para limpiar el texto. A continuación, queremos extraer las entidades y relaciones (hechos) del texto. Para las entidades, podemos usar algoritmos de reconocimiento de entidades nombradas (NER, por sus siglas en inglés). Para la relación, podemos usar técnicas de análisis de dependencia de oraciones para encontrar la relación entre cualquier par de entidades. Ejemplo de artículo con código.
- **Selección de hechos:** Una vez que hemos extraído varios hechos, los siguientes pasos obvios podrían ser eliminar duplicados e identificar hechos relevantes que podrían agregarse a un KG. Para identificar duplicados, podemos usar técnicas de desambiguación de entidades y relaciones. La idea es consolidar los mismos hechos o elementos de un hecho, en caso de repeticiones. Por ejemplo, "Albert Einstein" también puede escribirse como "Albert E." o "A. Einstein" en el texto, pero en realidad, todos se refieren a la misma entidad. Finalmente, podemos tener un sistema basado en reglas exhaustivo que decida qué tríada debería agregarse al KG o cuál podría omitirse según factores como información redundante ( $A \rightarrow$  hermano de  $\rightarrow B$  está presente, por lo tanto  $B \rightarrow$  hermano de  $\rightarrow A$  es redundante) o información no relevante.



Pasos involucrados en la creación automática de grafos de conocimiento

## Ontología del grafo de conocimiento

En ciencias de la computación y ciencias de la comunicación, una ontología es una definición formal de tipos, propiedades, y relaciones entre entidades que realmente o fundamentalmente existen para un dominio de discurso en particular.

Una ontología cataloga las variables requeridas para algún conjunto de computación y establece las relaciones entre ellos. En los campos de la inteligencia artificial y la Web Semántica, se crean ontologías para limitar la complejidad y para organizar la información. La ontología puede entonces ser aplicada para resolver problemas. Se utiliza para crear un modelo del mundo (en la práctica solo un subconjunto), que enumera los tipos de entidades, las relaciones que las conectan y las restricciones sobre las formas en que las entidades y las relaciones pueden combinarse. De cierta manera, una ontología define las reglas sobre cómo se conectan las entidades dentro del mundo.

El

Framework de Descripción de Recursos (RDF, por sus siglas en inglés) y el Lenguaje de Ontología Web (OWL, por sus siglas en inglés) son algunos de los marcos de vocabulario utilizados para modelar la ontología. Proporcionan un marco común para expresar esta información de modo que pueda ser intercambiada entre aplicaciones sin pérdida de significado.

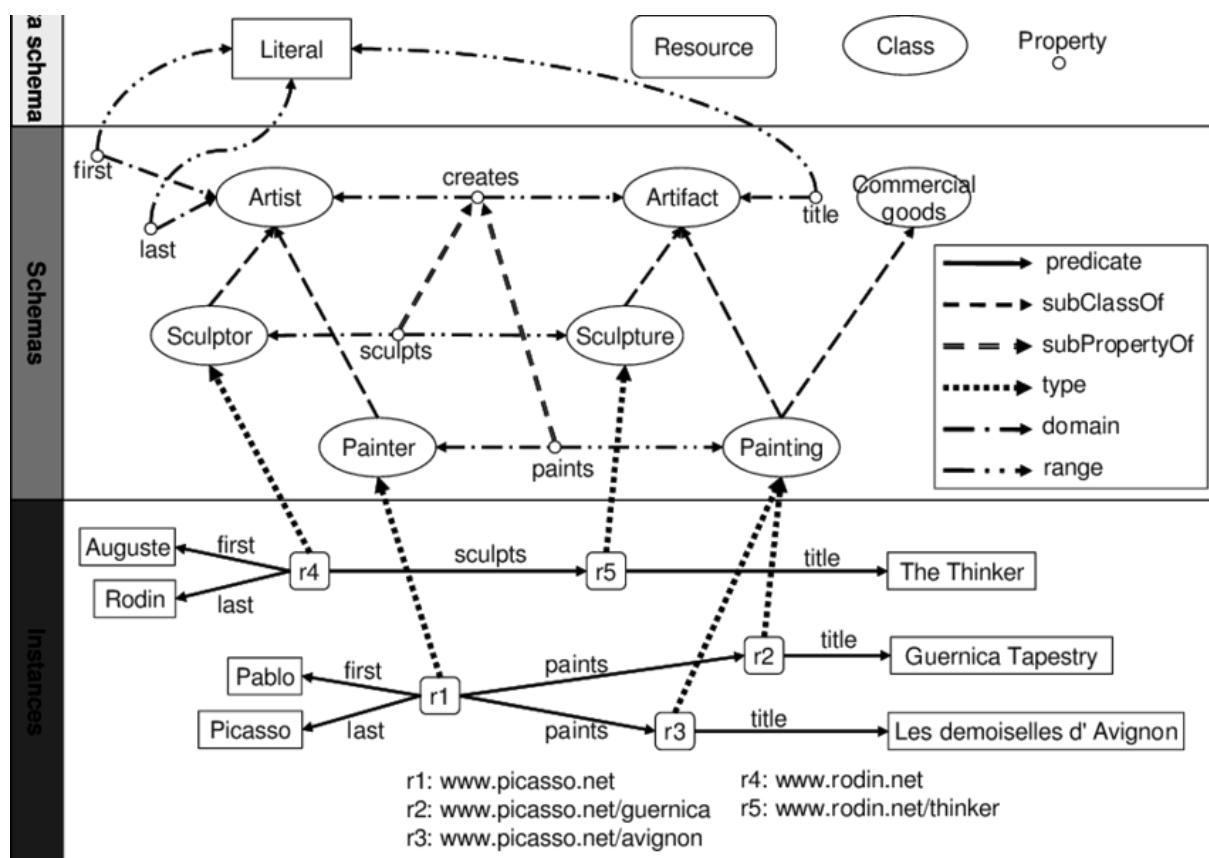
Los componentes claves de una ontología son:

- **Clases/Conceptos:** Los protagonistas principales, por ejemplo "Paciente", "Doctor" y "Enfermedad". Son los bloques de construcción que dan estructura a nuestro conjunto.
- **Individuos/Instancias:** Son los sujetos específicos, como "Dr. Strange" — son los personajes o cosas reales. Se ajustan a las Clases/Conceptos que hemos establecido.
- **Propiedades:** Estas son como relaciones entre personajes, mostrando cómo están vinculados. Por ejemplo, la propiedad "tieneSíntoma" enlaza "Enfermedad" y "Síntoma", al igual que las pistas en una historia de detectives.
- **Axiomas y Restricciones:** Estas son como las reglas de nuestro mundo. Los axiomas son como notas de detective — declaraciones lógicas que desentrañan relaciones. Y las restricciones son como indicaciones escénicas, guiando las acciones de los personajes.

- **Jerarquías y Taxonomías:** Imaginemos organizar personajes en un árbol genealógico — las grandes categorías son como los mayores, y las específicas son como sus hijos. Es como dar orden a nuestro elenco.
- **Inferencia:** Las acciones de los personajes conducen a conclusiones. Es como predecir un giro en la trama basado en las pistas que hemos descubierto.

## RDF (Resource Description Framework)

RDF es un estándar y modelo para la representación de información en la web y nos permite crear *ontologías* de una manera formal, de modo que pueda ser procesada por sistemas de computación. Es una de las principales herramientas utilizadas en la creación y gestión de la web semántica.



Este marco permite describir recursos y sus relaciones de una manera estructurada y semántica. Estos "recursos" pueden ser cualquier cosa, desde personas y lugares hasta conceptos abstractos, y se identifican mediante URIs (Identificadores Uniformes de Recursos).

- **Estructura:** La información en RDF se representa en forma de "triples", que consisten en un sujeto, un predicado y un objeto.
  - **Sujeto:** Es el recurso que se está describiendo.
  - **Predicado:** Define la relación o propiedad del sujeto.
  - **Objeto:** Es el valor o recurso al que se refiere el predicado.

Por ejemplo, este triple indica que "Juan tiene una edad de 30 años":

```
<Juan> <tieneEdad> <30>
```

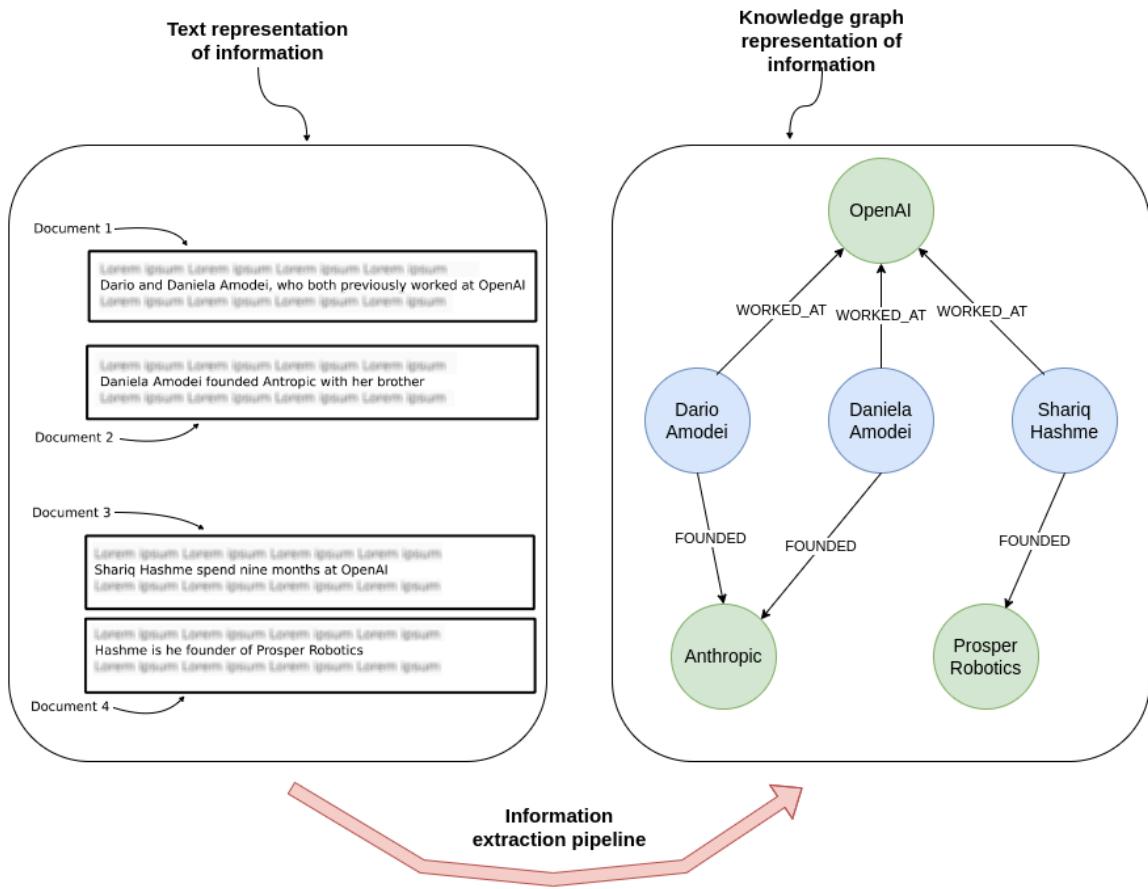
- **Uso:** RDF permite crear una "web de datos" que puede ser navegada y consultada. Esto facilita la conexión de datos entre diferentes fuentes y permite que las máquinas "entiendan" y procesen la información de manera más efectiva.
- **Formatos de Serialización:** Aunque RDF define la estructura y semántica de los datos, hay diferentes formatos para escribir o "serializar" estos datos, como RDF/XML, Turtle, N-Triples y JSON-LD, entre otros.
- **Aplicación:** RDF es la base para muchos otros estándares y herramientas en la web semántica, incluyendo SPARQL (un lenguaje de consulta para RDF) y OWL (un lenguaje de ontología basado en RDF). Es utilizado en una variedad de aplicaciones, desde la gestión de bibliotecas y museos hasta la integración de datos empresariales y la construcción de motores de búsqueda semánticos.

## Generación de ontología a partir de texto

En el siguiente ejemplo, exploraremos cómo generar una ontología básica a partir de una serie de frases. Al hacerlo, obtendremos una visión de cómo se pueden identificar y extraer entidades y relaciones a partir de texto y cómo se pueden representar semánticamente para su posterior uso en aplicaciones de NLP.

### Análisis de texto con POS Tagging

Las técnicas de etiquetado POS y NER, nos permiten conocer la estructura sintáctica de las frases y las entidades como personas, lugares, empresas, etc. Con esa información es posible construir grafos de conocimiento:



Al comprender cómo se relacionan las palabras entre sí en una oración, podemos extraer relaciones significativas entre ellas. Por ejemplo, en la frase "Juan lee un libro", el etiquetado POS nos permite identificar "Juan" como sujeto, "lee" como verbo y "libro" como objeto. Un grafo de conocimiento es una representación gráfica que conecta entidades (como personas, lugares o cosas) con relaciones específicas. En nuestro ejemplo anterior, "Juan" y "libro" serían las entidades, y "lee" sería la relación.

```
# !pip install spacy networkx
# !python -m spacy download es_core_news_md

import spacy
from spacy import displacy
from IPython.core.display import display, HTML

nlp = spacy.load("es_core_news_md")

sentences = [
    "María es amiga de Sofía.",
```

```

"Juan completó la tarea."
]

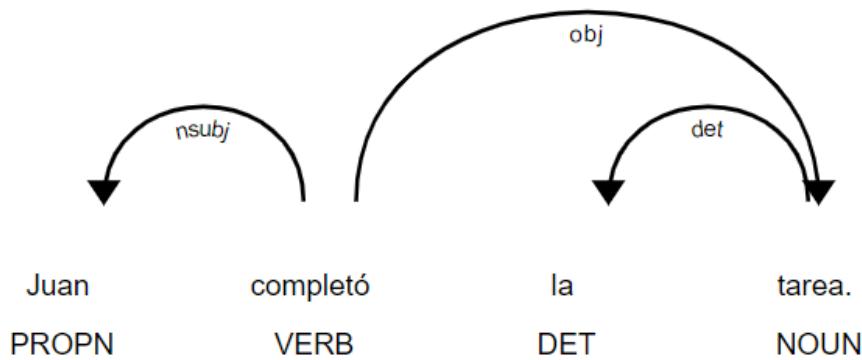
for sentence in sentences:
    doc = nlp(sentence)

    # Visualizar token.pos_ y token.dep_
    display(HTML(f"<strong>Frase:</strong> {sentence}"))
    displacy.render(doc, style='dep', jupyter=True, options={'distance': 120})

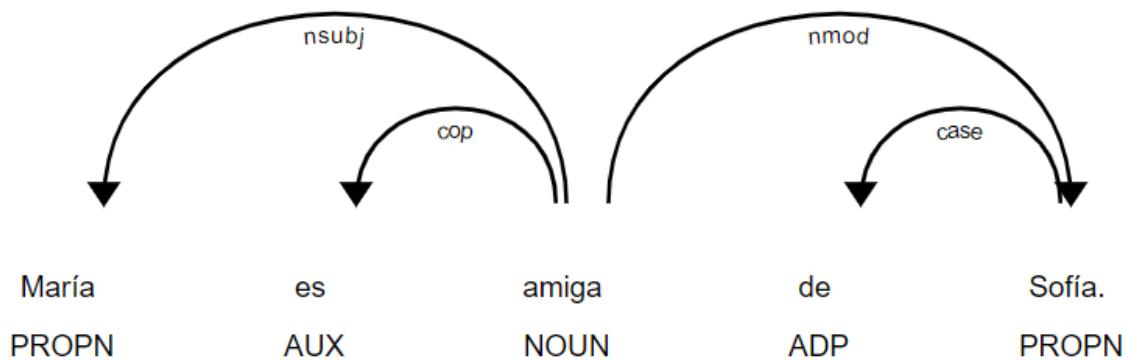
```

El resultado será:

**Frase:** Juan completó la tarea.



**Frase:** María es amiga de Sofía.



Visualizando la estructura de las frases, podríamos construir algoritmos que permitan extraer los hechos y transformarlos en grafos de conocimiento.

Veamos otro ejemplo con una implementación sencilla:

```

# !pip install spacy networkx
# !python -m spacy download es_core_news_md

# Importar las bibliotecas necesarias
import spacy
import networkx as nx
import matplotlib.pyplot as plt

# Cargar el modelo en español para tokenización, etiquetado POS, análisis y NLP
nlp = spacy.load("es_core_news_md")

# Función para extraer entidades y relaciones de un documento
def extract_entities_relations(doc):
    subject = ""
    obj = ""
    relation = ""
    prep_obj = "" # Objeto de una preposición

    for token in doc:
        # Detectar el sujeto
        if "subj" in token.dep_:
            subject = token.text
        # Detectar el objeto directo
        if "obj" in token.dep_:
            obj = token.text
        # Detectar el ROOT o relación principal
        if token.dep_ == "ROOT":
            relation = token.text
        # Buscar hijos del ROOT que sean modificadores nominales (nmod)
        for child in token.children:
            if child.dep_ == "nmod":
                prep_obj = child.text
        # Detectar preposiciones y su objeto
        if token.dep_ == "prep":
            for child in token.children:
                prep_obj = child.text

    # Si hay un objeto de preposición, usarlo como el objeto principal

```

```

if prep_obj:
    obj = prep_obj

return (subject, obj), [relation]

# Lista de frases
sentences = ["Juan completó la tarea",
             "María ganó la competencia",
             "Pedro escribió el libro",
             "Sofía conoce a Juan",
             "Carlos cocinó la cena",
             "Sofía es amiga de María",
             "Pedro engañó a María",
             "María quiere a Carlos",
             "Sofía come la cena",
             "Juan lee el libro"]

# Crear un grafo dirigido
G = nx.DiGraph()

# Diccionario para almacenar las etiquetas de las aristas (relaciones)
edge_labels = {}

# Procesar cada frase, extraer entidades y relaciones, y añadir al grafo
for sentence in sentences:
    doc = nlp(sentence)
    entities, relations = extract_entities_relations(doc)
    print(entities, relations)
    if entities[0] and entities[1] and relations:
        G.add_edge(entities[0], entities[1])
        edge_labels[(entities[0], entities[1])] = relations[0]

# Dibujar el grafo
# Definir un tamaño de figura
plt.figure(figsize=(10, 10))

# Aumentar el espacio entre nodos

```

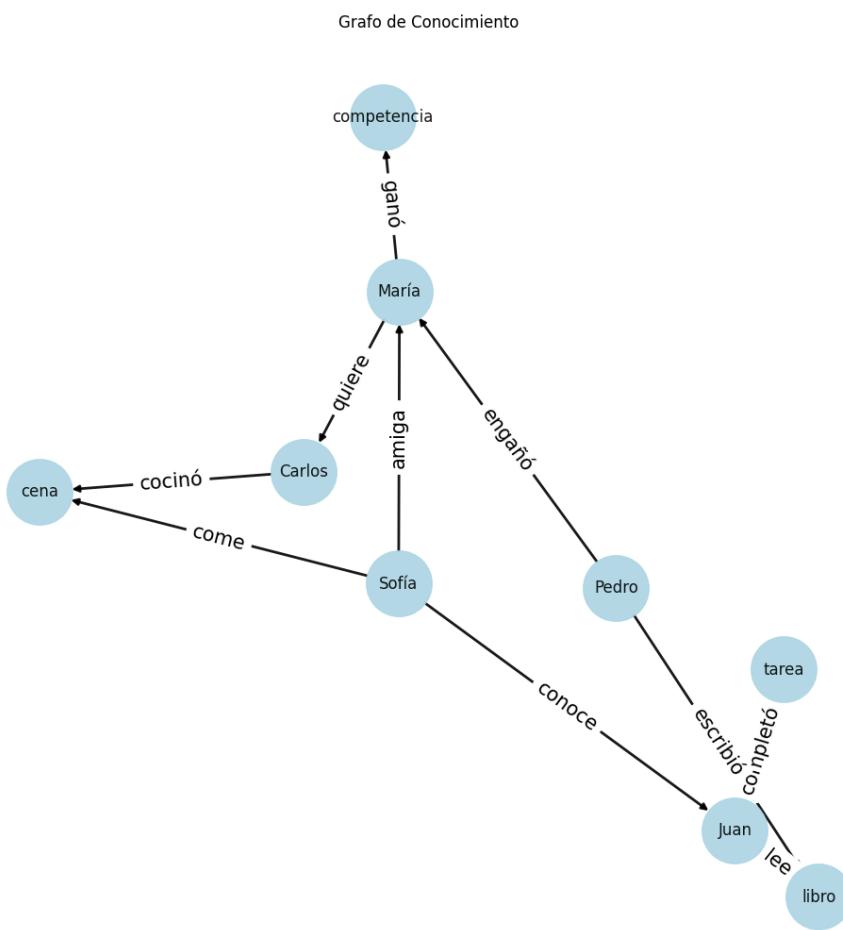
```

pos = nx.spring_layout(G, k=0.5)

nx.draw(G, pos, edge_color='black', width=2.0, linewidths=1,
        node_size=2500, node_color='lightblue', alpha=0.9,
        labels={node: node for node in G.nodes()})
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=1)
plt.title('Grafo de Conocimiento')
plt.axis('off')
plt.show()

```

Y el script generará una imagen como la siguiente imagen:



Este ejemplo demuestra cómo construir un grafo de conocimiento a partir de una serie de frases utilizando técnicas de Procesamiento del Lenguaje Natural (NLP). Se inicia cargando un modelo lingüístico en español que permite realizar tareas como tokenización, etiquetado de partes del discurso (POS) y reconocimiento de entidades nombradas (NER). A través de la función

`extract_entities_relations`, se analiza la estructura gramatical de cada frase para identificar el sujeto, el objeto y la relación principal entre ellos.

Una vez extraídas las entidades y relaciones, el código procede a construir un grafo dirigido donde cada nodo representa una entidad (por ejemplo, "Juan" o "libro") y cada arista representa una relación (por ejemplo, "completó" o "escribe"). El grafo se visualiza al final, mostrando gráficamente cómo se interconectan las diferentes entidades a través de las relaciones identificadas. Este enfoque permite representar visualmente el conocimiento extraído del texto, facilitando su interpretación y análisis.

## Exportar RDF

Supongamos que queremos exportar los grafos de conocimiento que acabamos de crear. Podríamos hacerlo usando el formato RDF/XML, como se muestra a continuación:

```
# !pip install rdflib

from rdflib import Graph, Literal, RDF, Namespace, URIRef

# Crear un grafo RDF
g = Graph()

# Definir un espacio de nombres para tus recursos
n = Namespace("http://example.org/")

# Añadir nodos y relaciones al grafo RDF
for edge in G.edges():
    subject_name, object_name = edge
    relation_name = edge_labels[edge]

    subject = URIRef(n + subject_name)
    predicate = URIRef(n + relation_name)
    obj = URIRef(n + object_name)

    g.add((subject, predicate, obj))

# Serializar y exportar el grafo a RDF/XML
```

```

rdf_output = g.serialize(format='xml')
print(rdf_output)

# Si deseas guardar el RDF en un archivo
with open("graph.rdf", "w") as file:
    file.write(rdf_output)

```

Resultado:

```

<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF
  xmlns:ns1="http://example.org/"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
>
  <rdf:Description rdf:about="http://example.org/Juan">
    <ns1:completó rdf:resource="http://example.org/tarea"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://example.org/María">
    <ns1:ganó rdf:resource="http://example.org/competencia"/>
    <ns1:quiere rdf:resource="http://example.org/Carlos"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://example.org/Sofía">
    <ns1:conoce rdf:resource="http://example.org/Juan"/>
    <ns1:amiga rdf:resource="http://example.org/María"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://example.org/Carlos">
    <ns1:cocinó rdf:resource="http://example.org/cena"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://example.org/Pedro">
    <ns1:escribió rdf:resource="http://example.org/libro"/>
    <ns1:engaño rdf:resource="http://example.org/María"/>
  </rdf:Description>
</rdf:RDF>

```

También es posible exportar a otros formatos, como por ejemplo Turtle:

```

from rdflib import Graph, Literal, RDF, Namespace, URIRef

```

```

# Crear un grafo RDF
g = Graph()

# Definir un espacio de nombres para tus recursos
n = Namespace("http://example.org/")

# Añadir nodos y relaciones al grafo RDF
for edge in G.edges():
    subject_name, object_name = edge
    relation_name = edge_labels[edge]

    subject = URIRef(n + subject_name)
    predicate = URIRef(n + relation_name)
    obj = URIRef(n + object_name)

    g.add((subject, predicate, obj))

# Serializar y exportar el grafo a formato Turtle
turtle_output = g.serialize(format='turtle')
print(turtle_output)

# Si deseas guardar el Turtle en un archivo
with open("graph.ttl", "w") as file:
    file.write(turtle_output)

```

Resultado:

```

@prefix ns1: <http://example.org/> .

ns1:Pedro ns1:engañó ns1:María ;
      ns1:escribió ns1:libro .

ns1:Sofía ns1:amiga ns1:María ;
      ns1:conoce ns1:Juan .

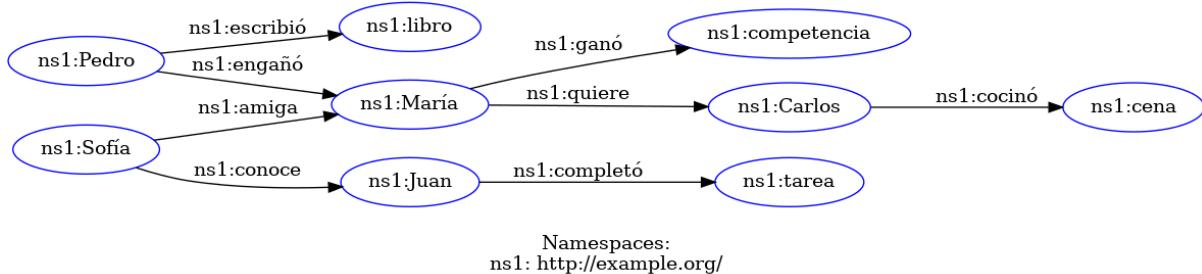
ns1:Carlos ns1:cocinó ns1:cena .

ns1:Juan ns1:completó ns1:tarea .

```

```
ns1:María ns1:ganó ns1:competencia ;  
ns1:quiere ns1:Carlos .
```

Ahora que hemos exportado nuestros grafos, podemos validarlos y visualizarlos con otras herramientas, por ejemplo con esta página:  
<https://www.ldf.fi/service/rdf-grapher>



## 4. Graph databases (Bases de Datos de Grafos)

Un sistema de gestión de bases de datos de grafos (de ahora en adelante, una base de datos de grafos) es un sistema de gestión de bases de datos en línea con métodos de Crear, Leer, Actualizar y Eliminar (CRUD, por sus siglas en inglés) que expone un modelo de datos de grafo. Las bases de datos de grafos generalmente se construyen para ser utilizadas con sistemas transaccionales (OLTP). En consecuencia, normalmente están optimizadas para el rendimiento transaccional y están diseñadas con la integridad transaccional y la disponibilidad operacional en mente.

Hay dos propiedades de las bases de datos de grafos que deberíamos considerar al investigar tecnologías de bases de datos de grafos:

- **El almacenamiento subyacente**

Algunas bases de datos de grafos utilizan almacenamiento nativo de grafos que está optimizado y diseñado para almacenar y gestionar grafos. Sin embargo, no todas las tecnologías de bases de datos de grafos utilizan almacenamiento nativo de grafos. Algunas serializan los datos del grafo en una base de datos relacional, una base de datos orientada a objetos o algún otro almacén de datos de propósito general.

- **El motor de procesamiento**

Algunas definiciones exigen que una base de datos de grafos utilice

adyacencia sin índices, lo que significa que los nodos conectados físicamente se "apuntan" entre sí en la base de datos. Aquí tomamos una perspectiva un poco más amplia: cualquier base de datos que, desde el punto de vista del usuario, se comporte como una base de datos de grafos (es decir, expone un modelo de datos de grafo a través de operaciones CRUD) califica como una base de datos de grafos. Sin embargo, reconocemos las ventajas significativas en rendimiento de la adyacencia sin índices, y por lo tanto, utilizamos el término procesamiento de grafos nativo para describir las bases de datos de grafos que aprovechan la adyacencia sin índices.



Es importante señalar que el almacenamiento de grafos nativo y el procesamiento de grafos nativo no son ni buenos ni malos, simplemente son compensaciones clásicas de ingeniería. El beneficio del almacenamiento de grafos nativo es que su pila diseñada específicamente está diseñada para rendimiento y escalabilidad. La ventaja del almacenamiento de grafos no nativo, en contraste, es que típicamente depende de un backend no gráfico maduro (como MySQL) cuyas características de producción son bien comprendidas por los equipos de operaciones y soportes de IT. El procesamiento de grafos nativo (adyacencia sin índices) beneficia el rendimiento del recorrido, pero a expensas de hacer que algunas consultas que no utilizan recorridos sean difíciles o intensivas en memoria.

Las relaciones son ciudadanos de primera clase del modelo de datos de grafo. Esto no es el caso en otros sistemas de gestión de bases de datos, donde tenemos que inferir conexiones entre entidades utilizando cosas como claves foráneas o procesamiento fuera de banda como map-reduce. Al ensamblar las simples abstracciones de nodos y relaciones en estructuras conectadas, las bases de datos de grafos nos permiten construir modelos arbitrariamente sofisticados que se asemejan estrechamente a nuestro dominio del problema. Los modelos resultantes son más simples y al mismo tiempo más expresivos que aquellos producidos usando bases de datos relacionales tradicionales y otros almacenes NOSQL (No Solo SQL).

## The graph database ecosystem 2019



Mapa de las principales bases de datos orientadas a grafos.

Podemos destacar diversas bases de datos Open Source populares, por ejemplo:

- Memgraph -
- Neo4J -
- NebulaGraph -
- ArangoDB -
- Apache TinkerPop -
- Dgraph -
- RedisGraph -

Para encontrar más proyectos relacionados con GDB Open Source, podemos ver este repositorio:

### El Poder de las Bases de Datos de Grafos

A pesar del hecho de que prácticamente cualquier cosa puede ser modelada como un grafo, vivimos en un mundo pragmático de presupuestos, cronogramas de proyectos, estándares corporativos y conjuntos de habilidades estandarizadas. Que una base de datos de grafos proporcione una técnica poderosa pero novedosa de modelado de datos no proporciona por sí misma

una justificación suficiente para reemplazar una plataforma de datos bien establecida y bien entendida; también debe haber un beneficio práctico inmediato y muy significativo. En el caso de las bases de datos de grafos, esta motivación existe en forma de un conjunto de casos de uso y patrones de datos cuyo rendimiento mejora en uno o más órdenes de magnitud cuando se implementa en un grafo, y cuya latencia es mucho menor en comparación con el procesamiento por lotes de agregados. Además de este beneficio de rendimiento, las bases de datos de grafos ofrecen un modelo de datos extremadamente flexible y un modo de entrega alineado con las prácticas actuales de entrega de software ágil.

### **Rendimiento**

Una razón convincente, entonces, para elegir una base de datos de grafos es el enorme aumento de rendimiento al tratar con datos conectados versus bases de datos relacionales y almacenes NOSQL. En contraste con las bases de datos relacionales, donde el rendimiento de las consultas intensivas en uniones se deteriora a medida que el conjunto de datos crece, con una base de datos de grafos el rendimiento tiende a permanecer relativamente constante, incluso a medida que el conjunto de datos crece. Esto se debe a que las consultas están localizadas en una porción del grafo. Como resultado, el tiempo de ejecución de cada consulta es proporcional solo al tamaño de la parte del grafo recorrida para satisfacer esa consulta, en lugar del tamaño del grafo en general.

### **Flexibilidad**

Como desarrolladores y arquitectos de datos, queremos conectar datos según lo dicte el dominio, permitiendo que la estructura y el esquema emergan en tandem con nuestra creciente comprensión del espacio del problema, en lugar de ser impuestos desde el principio, cuando sabemos menos sobre la verdadera forma y complejidades de los datos. Las bases de datos de grafos abordan esta necesidad directamente. El modelo de datos de grafo expresa y acomoda las necesidades empresariales de una manera que permite que las TI se muevan al ritmo de los negocios. Los grafos son naturalmente aditivos, lo que significa que podemos agregar nuevos tipos de relaciones, nuevos nodos, nuevas etiquetas y nuevos subgrafos a una estructura existente sin perturbar las consultas y funcionalidades de la aplicación existentes. Estas cosas generalmente tienen implicaciones positivas para la productividad del desarrollador y el riesgo del proyecto. Debido a la flexibilidad del modelo de grafo, no tenemos que modelar nuestro dominio en detalle exhaustivo de antemano, una práctica que es poco menos que imprudente ante los

cambiantes requisitos empresariales. La naturaleza aditiva de los grafos también significa que tendemos a realizar menos migraciones, reduciendo así la sobrecarga y el riesgo de mantenimiento.

## **Agilidad**

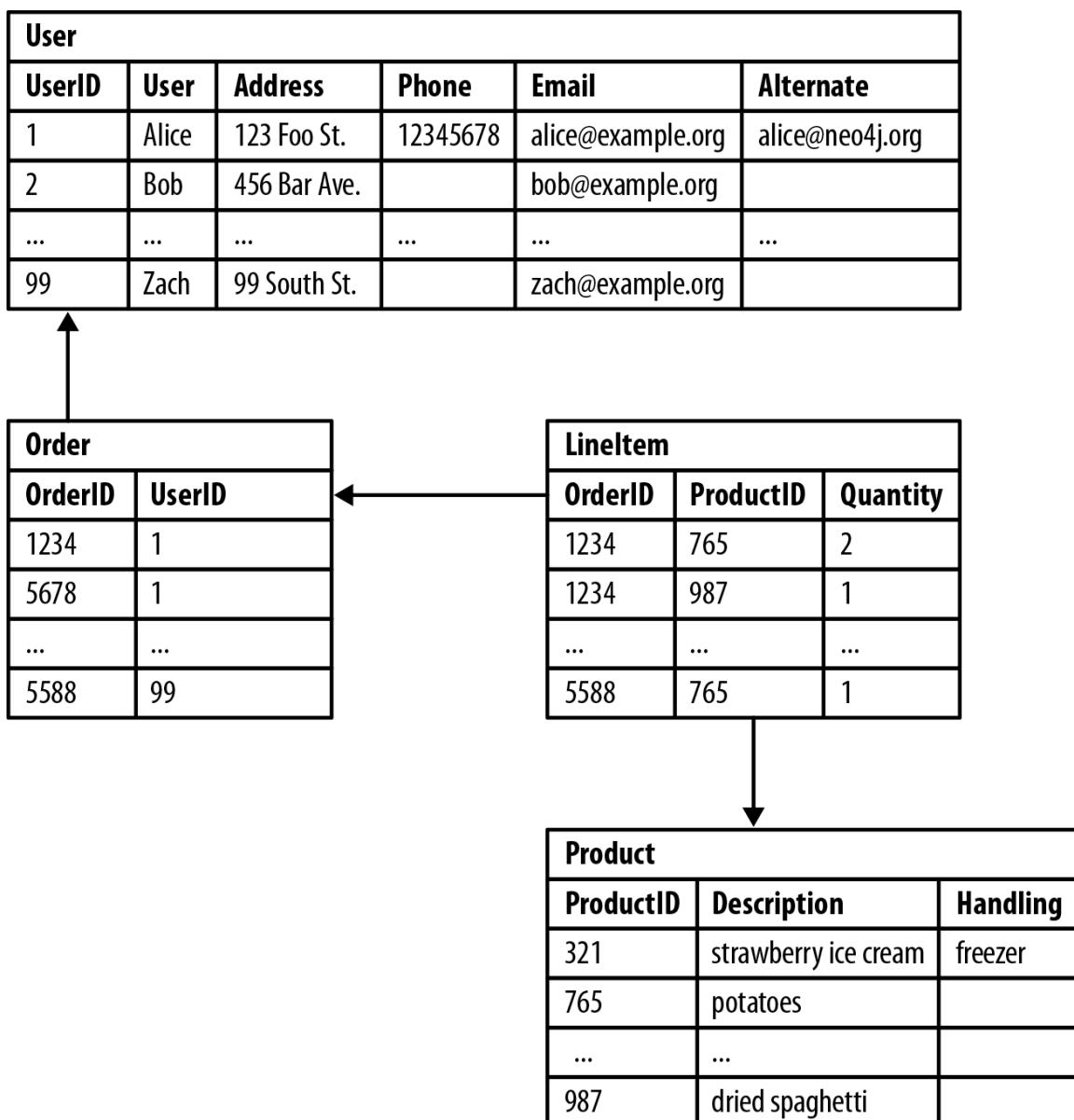
Queremos ser capaces de evolucionar nuestro modelo de datos al ritmo del resto de nuestra aplicación, utilizando una tecnología alineada con las prácticas actuales de entrega de software incremental e iterativa. Las modernas bases de datos de grafos nos equipan para realizar un desarrollo sin fricciones y un mantenimiento de sistemas elegante. En particular, la naturaleza libre de esquema del modelo de datos de grafo, junto con la naturaleza comprobable de la interfaz de programación de aplicaciones (API) y el lenguaje de consulta de una base de datos de grafo, nos empodera para evolucionar una aplicación de manera controlada. Al mismo tiempo, precisamente porque no tienen esquema, las bases de datos de grafos carecen de los mecanismos de gobernanza de datos orientados a esquemas con los que estamos familiarizados en el mundo relacional. Pero esto no es un riesgo; más bien, invoca un tipo de gobernanza mucho más visible y accionable. La gobernanza se aplica típicamente de manera programática, utilizando pruebas para desarrollar el modelo de datos y las consultas, así como para afirmar las reglas empresariales que dependen del grafo. Esta ya no es una práctica controvertida: más que el desarrollo relacional, el desarrollo de bases de datos de grafos se alinea bien con las prácticas actuales de desarrollo de software ágil y dirigido por pruebas, permitiendo que las aplicaciones respaldadas por bases de datos de grafos evolucionen al ritmo de los cambiantes entornos empresariales.

## **Las Bases de Datos Relacionales Carecen de Relaciones**

Durante varias décadas, los desarrolladores han intentado acomodar conjuntos de datos conectados y semi-estructurados dentro de bases de datos relacionales. Pero, mientras que las bases de datos relacionales fueron inicialmente diseñadas para codificar formularios en papel y estructuras tabulares —algo que hacen excepcionalmente bien—, enfrentan dificultades al intentar modelar las relaciones ad hoc y excepcionales que surgen en el mundo real. Irónicamente, las bases de datos relacionales lidian mal con las relaciones.

Las relaciones sí existen en el lenguaje de las bases de datos relacionales, pero solo en el momento de modelar, como medio para unir tablas. A menudo necesitamos desambiguar la semántica de las relaciones que conectan entidades, así como calificar su peso o fuerza. Las relaciones relacionales no

hacen nada de eso. Peor aún, a medida que se multiplican los datos atípicos y la estructura general del conjunto de datos se vuelve más compleja y menos uniforme, el modelo relacional se ve sobrecargado con grandes tablas de unión, filas escasamente pobladas y mucha lógica de comprobación de valores nulos. El aumento de la conectividad se traduce en el mundo relacional en un incremento de uniones, lo que dificulta el rendimiento y nos hace difícil evolucionar una base de datos existente en respuesta a las cambiantes necesidades empresariales.

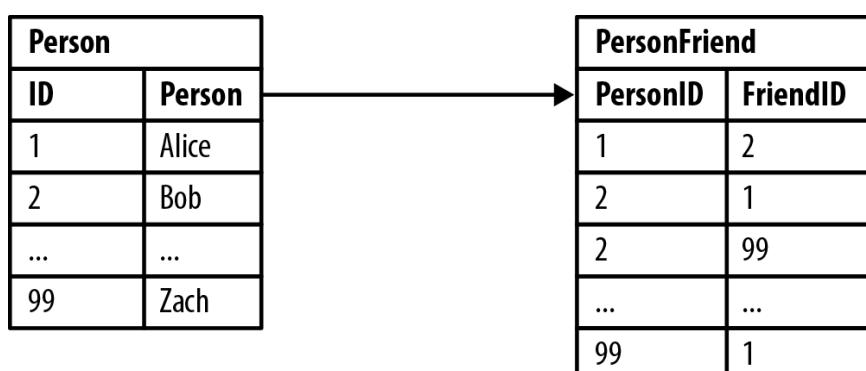


Las relaciones semánticas están ocultas en una base de datos relacional

La aplicación ejerce una tremenda influencia sobre el diseño de este esquema, haciendo que algunas consultas sean muy fáciles y otras más difíciles:

- Las tablas de unión añaden complejidad accidental; mezclan datos empresariales con metadatos de claves foráneas.
- Las restricciones de claves foráneas añaden sobrecarga adicional de desarrollo y mantenimiento solo para hacer funcionar la base de datos.
- Las tablas dispersas con columnas anulables requieren comprobaciones especiales en el código, a pesar de la presencia de un esquema.
- Se necesitan varias uniones costosas solo para descubrir qué compró un cliente.
- Las consultas recíprocas son aún más costosas. "¿Qué productos compró un cliente?" es relativamente barato en comparación con "¿qué clientes compraron este producto?", que es la base de los sistemas de recomendación. Podríamos introducir un índice, pero incluso con un índice, las preguntas recursivas como "¿qué clientes que compraron este producto también compraron aquel producto?" rápidamente se vuelven prohibitivamente caras a medida que aumenta el grado de recursión.
- Las bases de datos relacionales tienen dificultades con dominios altamente conectados. Para entender el coste de realizar consultas conectadas en una base de datos relacional, veremos algunas consultas simples y no tan simples en un dominio de red social.

La siguiente figura muestra una disposición simple de tabla de unión para registrar amistades:



The diagram illustrates a relational database schema for managing friendships. It consists of two tables: 'Person' and 'PersonFriend'. The 'Person' table contains columns for 'ID' and 'Person', with data entries for Alice, Bob, and Zach. The 'PersonFriend' table contains columns for 'PersonID' and 'FriendID', with data entries showing various friendships between the individuals in the 'Person' table.

Person	
ID	Person
1	Alice
2	Bob
...	...
99	Zach

PersonFriend	
PersonID	FriendID
1	2
2	1
2	99
...	...
99	1

Modelando amigos y amigos de amigos en una base de datos relacional

Preguntar "¿quiénes son los amigos de Bob?" es relativamente fácil, como se muestra en el ejemplo:

```
SELECT p1.Person  
FROM Person p1 JOIN PersonFriend  
ON PersonFriend.FriendID = p1.ID  
JOIN Person p2  
ON PersonFriend.PersonID = p2.ID  
WHERE p2.Person = 'Bob'
```

Basándonos en nuestros datos de muestra, la respuesta es Alice y Zach. Esta no es una consulta especialmente costosa o difícil, porque limita el número de filas bajo consideración usando el filtro WHERE Person.person='Bob'.

La amistad no siempre es una relación reflexiva, así que en el ejemplo, hacemos la consulta recíproca, que es, "¿quién es amigo de Bob?"

```
SELECT p1.Person  
FROM Person p1 JOIN PersonFriend  
ON PersonFriend.PersonID = p1.ID  
JOIN Person p2  
ON PersonFriend.FriendID = p2.ID  
WHERE p2.Person = 'Bob'
```

La respuesta a esta consulta es Alice; lamentablemente, Zach no considera a Bob como un amigo. Esta consulta recíproca sigue siendo fácil de implementar, pero desde el lado de la base de datos es más costosa, porque ahora tiene que considerar todas las filas en la tabla PersonFriend.

Podemos añadir un índice, pero esto todavía implica una costosa capa de indirección. Las cosas se vuelven aún más problemáticas cuando preguntamos, "¿quiénes son los amigos de mis amigos?"

Las jerarquías en SQL utilizan uniones recursivas, lo que hace que la consulta sea sintáctica y computacionalmente más compleja, como se muestra en el ejemplo a continuación. (Algunas bases de datos relacionales proporcionan "azúcar sintáctica" para esto, por ejemplo, Oracle tiene una función CONNECT BY, lo que simplifica la consulta, pero no la complejidad computacional subyacente).

```
SELECT p1.Person AS PERSON, p2.Person AS FRIEND_OF_FRIEND  
FROM PersonFriend pf1 JOIN Person p1
```

```
ON pf1.PersonID = p1.ID  
JOIN PersonFriend pf2  
ON pf2.PersonID = pf1.FriendID  
JOIN Person p2  
ON pf2.FriendID = p2.ID  
WHERE p1.Person = 'Alice' AND pf2.FriendID <> p1.ID
```

Esta consulta es computacionalmente compleja, aunque solo trata sobre los amigos de los amigos de Alice, y no profundiza más en la red social de Alice. Las cosas se vuelven más complejas y costosas cuanto más nos adentramos en la red. Aunque es posible obtener una respuesta a la pregunta "¿quiénes son los amigos de los amigos de mis amigos?" en un período razonable de tiempo, las consultas que se extienden a cuatro, cinco o seis grados de amistad se deterioran significativamente debido a la complejidad computacional y espacial de unir tablas recursivamente.

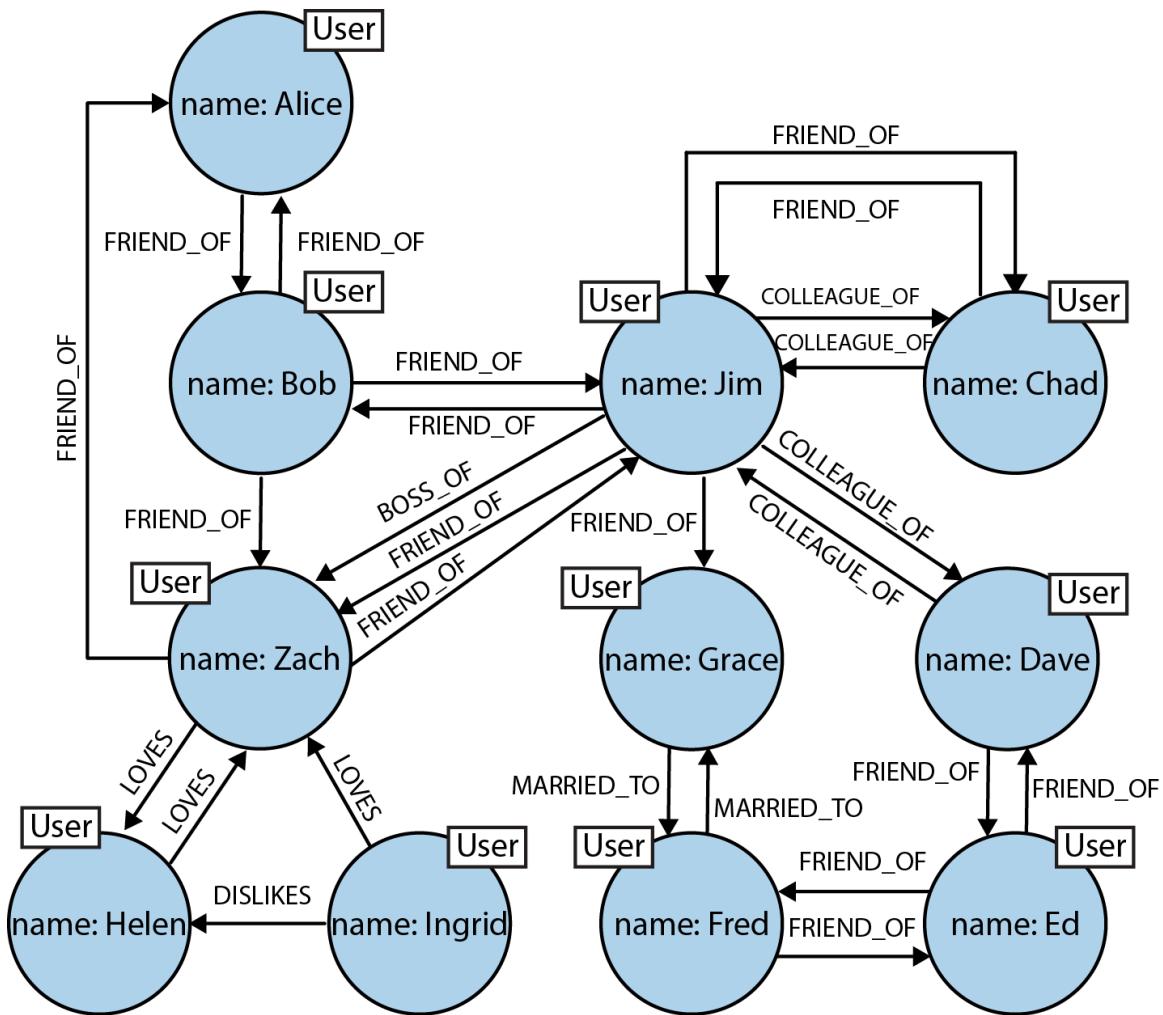
Trabajamos en contra de la corriente cada vez que intentamos modelar y consultar la conectividad en una base de datos relacional. Además de la complejidad de consulta y computacional que acabamos de describir, también tenemos que lidiar con la espada de doble filo del esquema. En la mayoría de las ocasiones, el esquema resulta ser tanto demasiado rígido como demasiado frágil. Para subvertir su rigidez, creamos tablas poco pobladas con muchas columnas anulables y código para manejar los casos excepcionales, todo porque no hay un esquema único que se adapte a la variedad de datos que encontramos. Esto aumenta el acoplamiento y destruye cualquier apariencia de cohesión. Su fragilidad se manifiesta como el esfuerzo y cuidado extra requeridos para migrar de un esquema a otro a medida que una aplicación evoluciona.

### **Las Bases de Datos de Grafos Abrazan las Relaciones**

Los ejemplos anteriores han tratado sobre datos implícitamente conectados. Como usuarios, inferimos dependencias semánticas entre entidades, pero los modelos de datos, y las bases de datos en sí, son ciegos a estas conexiones. Para compensar, nuestras aplicaciones deben crear una red a partir de los datos planos y desconectados disponibles, y luego lidiar con cualquier consulta lenta y escrituras latentes a través de tiendas desnormalizadas que surjan.

Lo que realmente queremos es una imagen cohesiva del conjunto, incluidas las conexiones entre elementos. En contraste con las tiendas que acabamos de

examinar, en el mundo del grafo, los datos conectados se almacenan como datos conectados. Donde hay conexiones en el dominio, hay conexiones en los datos. Por ejemplo, consideremos la red social mostrada en la figura:



Modelado fácil de amigos, colegas, trabajadores y amores (no correspondidos) en un grafo.

En esta red social, como en muchos casos del mundo real de datos conectados, las conexiones entre entidades no muestran uniformidad en todo el dominio; el dominio tiene una estructura variable. Una red social es un ejemplo popular de una red densamente conectada y con estructura variable, una que se resiste a ser capturada por un esquema único o dividida cómodamente a través de agregados desconectados. Nuestra simple red de amigos ha crecido en tamaño (ahora hay posibles amigos hasta a seis grados de distancia) y en riqueza expresiva. La flexibilidad del modelo de grafo nos ha permitido agregar nuevos nodos y nuevas relaciones sin comprometer la red existente o migrar datos; los datos originales y su intención permanecen intactos.

El grafo ofrece una imagen mucho más rica de la red. Podemos ver quién AMA a quién (y si ese amor es correspondido). Podemos ver quién es COLEGA de quién, y quién es JEFE de todos. Podemos ver quién no está disponible porque está CASADO con alguien más; incluso podemos ver los elementos antisociales en nuestra red social, representados por las relaciones de DISGUSTO. Con este grafo a nuestra disposición, ahora podemos observar las ventajas de rendimiento de las bases de datos de grafo al tratar con datos conectados.



### Etiquetas en el Grafo

A menudo queremos categorizar los nodos en nuestras redes según los roles que desempeñan. Algunos nodos, por ejemplo, podrían representar usuarios, mientras que otros representan pedidos o productos. En las GDB se usan etiquetas para representar los roles que un nodo desempeña en el grafo. Dado que un nodo puede cumplir varios roles diferentes en un grafo, algunos motores como Neo4J permite agregar más de una etiqueta a un nodo.

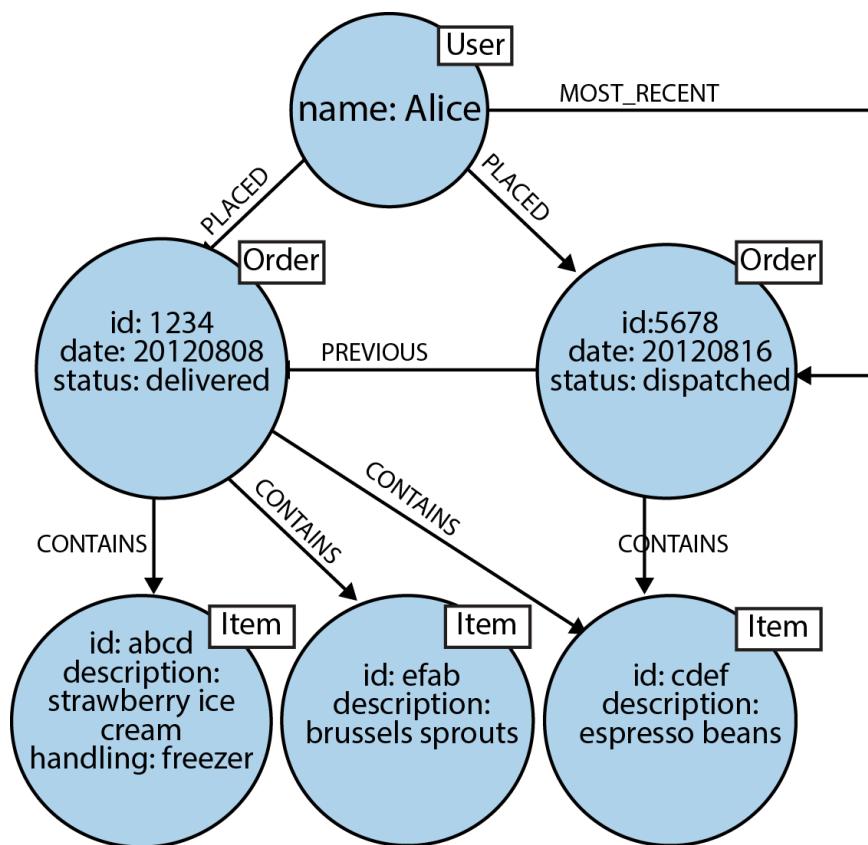
Usando etiquetas de esta manera, podemos agrupar nodos. Podemos pedirle a la base de datos, por ejemplo, que encuentre todos los nodos etiquetados como Usuario. Las etiquetas también proporcionan un punto de referencia para indexar nodos de forma declarativa. Donde un nodo representa a un usuario, hemos agregado una etiqueta de Usuario; donde representa un pedido, hemos agregado una etiqueta de Pedido, y así sucesivamente.

Las relaciones en un grafo naturalmente forman rutas. Consultar —o recorrer— el grafo implica seguir rutas. Debido a la naturaleza fundamentalmente orientada a rutas del modelo de datos, la mayoría de las operaciones de bases de datos de grafo basadas en rutas están altamente alineadas con la forma en que se organiza el dato, lo que las hace extremadamente eficientes.

El ejemplo de la red social ayuda a ilustrar cómo diferentes tecnologías tratan los datos conectados, ¿pero es un caso de uso válido? ¿Realmente necesitamos encontrar "amigos" tan remotos? Quizás no. Pero si sustituimos cualquier otro dominio por la red social, veremos que experimentamos beneficios similares en rendimiento, modelado y mantenimiento. Ya sea música o gestión de centros de datos, bioinformática o estadísticas de fútbol, sensores de red o series temporales de operaciones, los grafos proporcionan una visión poderosa de nuestros datos. Veamos, entonces, otra aplicación contemporánea de grafos: recomendar productos basados en el historial de

compras de un usuario y los historiales de sus amigos, vecinos y otras personas similares a él. Con este ejemplo, reuniremos varias facetas independientes del estilo de vida de un usuario para hacer recomendaciones precisas y rentables.

Comenzaremos modelando el historial de compras de un usuario como datos conectados. En un grafo, esto es tan simple como vincular al usuario con sus pedidos, y vincular pedidos entre sí para proporcionar un historial de compras, como se muestra en la figura:



El grafo mostrado en la figura ofrece una gran visión sobre el comportamiento del cliente. Podemos ver todos los pedidos que un usuario ha REALIZADO y podemos razonar fácilmente sobre qué CONTIENE cada pedido. A esta estructura de datos de dominio principal, luego hemos agregado soporte para varios patrones de acceso conocidos. Por ejemplo, los usuarios a menudo quieren ver su historial de pedidos, así que hemos añadido una estructura de lista enlazada al grafo que nos permite encontrar el pedido más reciente de un usuario siguiendo una relación MÁS RECIENTE. Luego podemos iterar a través de la lista, retrocediendo en el tiempo, siguiendo cada relación ANTERIOR. Si queremos avanzar en el tiempo, podemos seguir cada relación ANTERIOR en la dirección opuesta, o agregar una relación SIGUIENTE recíproca.

Ahora podemos comenzar a hacer recomendaciones. Si notamos que muchos usuarios que compran helado de fresa también compran granos de espresso, podemos empezar a recomendar esos granos a usuarios que normalmente solo compran el helado. Pero esta es una recomendación bastante unidimensional: podemos hacerlo mucho mejor. Para aumentar el poder de nuestro grafo, podemos unirlo a grafos de otros dominios. Dado que los grafos son estructuras naturalmente multidimensionales, es bastante sencillo hacer preguntas más sofisticadas a los datos para acceder a un segmento de mercado afinado. Por ejemplo, podemos preguntarle al grafo que nos encuentre "todos los sabores de helado que les gustan a las personas que disfrutan del espresso pero no les gustan las coles de Bruselas, y que viven en un barrio en particular".

Para el propósito de nuestra interpretación de los datos, podemos considerar el grado en que alguien compra repetidamente un producto como indicativo de si le gusta o no ese producto. Pero, ¿cómo podríamos definir "vivir en un barrio"? Resulta que las coordenadas geoespaciales se modelan muy convenientemente como grafos. Una de las estructuras más populares para representar coordenadas geoespaciales se llama R-Tree (Árbol R). Un R-Tree es un índice tipo grafo que describe cajas limitadas alrededor de geografías. Usando tal estructura, podemos describir jerarquías superpuestas de ubicaciones. Por ejemplo, podemos representar el hecho de que Londres está en el Reino Unido y que el código postal SW11 1BD está en Battersea, que es un distrito en Londres, que está en el sudeste de Inglaterra, que, a su vez, está en Gran Bretaña. Y debido a que los códigos postales del Reino Unido son detallados, podemos usar ese límite para dirigirnos a personas con gustos algo similares.

## **Lenguajes de consulta sobre bases de datos de grafos**

Las bases de datos de grafos están diseñadas para manejar datos en forma de nodos y relaciones. A diferencia de las bases de datos relacionales, que estructuran los datos en tablas y utilizan SQL para las consultas, las bases de datos de grafos necesitan un lenguaje diferente que pueda aprovechar eficientemente su estructura única.

El SQL (Structured Query Language) se utiliza en bases de datos relacionales y se centra en la realización de operaciones basadas en tablas como selección, inserción, actualización y eliminación. Los lenguajes de consulta para bases de

datos de grafos, en cambio, se centran en patrones de nodos, relaciones y propiedades para extraer, modificar y navegar por datos interconectados.

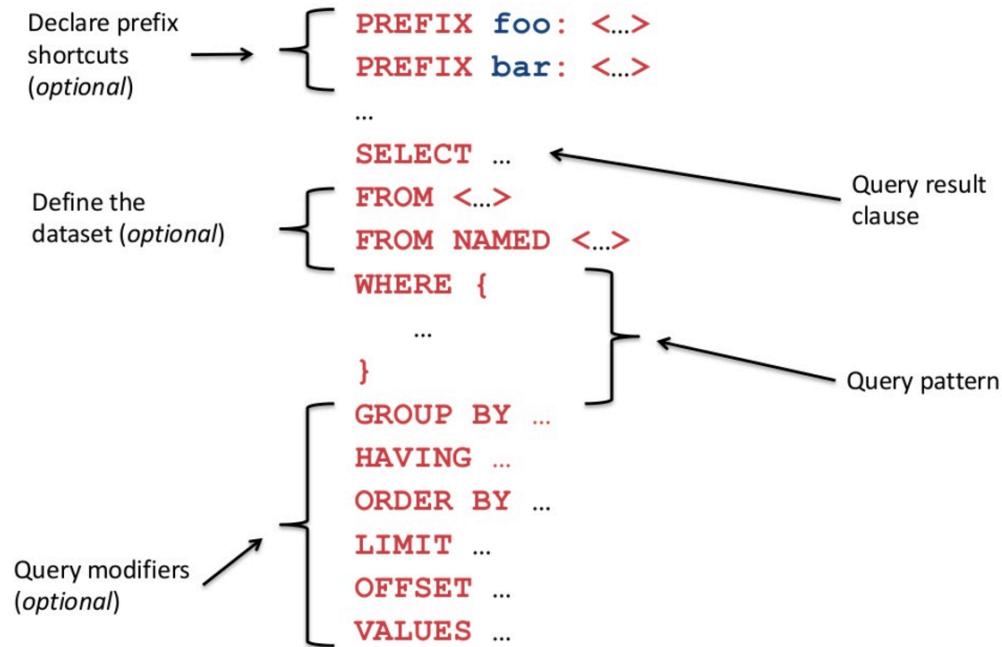
Lenguajes de consulta a bases de datos de grafos más populares:

- **Cypher**: Es el lenguaje de consulta de la base de datos de grafos Neo4j. Cypher es expresivo y eficiente para consultas relacionadas con patrones de grafos. Permite a los usuarios describir patrones en los grafos visualmente utilizando una sintaxis ASCII-art. Por ejemplo, una consulta para encontrar una relación entre dos nodos podría verse así: `(a)-[r]→(b)`.
- **SPARQL**: SPARQL es un lenguaje de consulta utilizado para consultar bases de datos que almacenan datos en el formato RDF (Resource Description Framework). Es especialmente popular en la Web Semántica y en aplicaciones de linked data. SPARQL permite a los usuarios definir patrones de triples (sujeto, predicado, objeto) para extraer información de un grafo RDF.
- **Gremlin**: Gremlin es un lenguaje de manipulación y consulta de grafos utilizado por Apache TinkerPop, un marco de computación de grafos. Gremlin permite realizar tanto consultas declarativas como imperativas y es compatible con diversos sistemas de bases de datos de grafos.

Mientras que SQL es un lenguaje diseñado para operar sobre tablas y relaciones en bases de datos relacionales, los lenguajes de consulta de bases de datos de grafos están optimizados para trabajar con estructuras de nodos y relaciones, permitiendo consultas más expresivas y eficientes en escenarios de datos interconectados. Cada lenguaje tiene sus propias ventajas y aplicaciones específicas según el sistema de base de datos y el tipo de datos con los que se esté trabajando.

### **Lenguaje SPARQL**

Una consulta SPARQL se utiliza para interrogar bases de datos que utilizan el formato RDF (Resource Description Framework).



Hoja de trucos (cheat sheet): [https://www.iro.umontreal.ca/~lapalme/ift6281/sparql-1\\_1-cheat-sheet.pdf](https://www.iro.umontreal.ca/~lapalme/ift6281/sparql-1_1-cheat-sheet.pdf)

<https://github.com/andrecastro0o/SPARQL-cheatsheet>

Aquí podemos ver una estructura bien básica de una consulta SPARQL y una explicación de sus componentes:

```

PREFIX prefix: <URI>
SELECT ?variable WHERE {
    sujeto predicado objeto.
}
LIMIT n

```

1. **PREFIX**: Es una manera de definir prefijos para las URIs, lo que facilita la escritura y lectura de la consulta. Por ejemplo, en vez de escribir <http://www.w3.org/2001/XMLSchema#string>, se puede usar un prefijo para acortarlo como `xsd:string`.
2. **SELECT**: Indica qué variables deseamos recuperar en la consulta. Las variables en SPARQL se denotan con un signo de interrogación (?) seguido por el nombre de la variable, como `?variable`.
3. **WHERE**: Define el patrón que debe coincidir con los datos en la base de datos RDF. Los patrones se escriben en tríadas que consisten en un sujeto, un predicado y un objeto. Si alguna de estas partes es una variable, la

consulta tratará de encontrar coincidencias para esa variable en la base de datos.

- **sujeto**: Es el recurso del que se quiere obtener información.
- **predicado**: Es la relación o propiedad que se quiere consultar del sujeto.
- **objeto**: Es el valor o recurso que se espera como resultado del predicado.

4. **LIMIT**: Es opcional y se utiliza para limitar el número de resultados devueltos por la consulta.

Además de **SELECT**, SPARQL ofrece otros tipos de consultas, como **ASK** (que devuelve un booleano), **CONSTRUCT** (que devuelve datos en formato RDF) y **DESCRIBE** (que devuelve una descripción RDF de un recurso específico).

Otros componentes útiles en las consultas SPARQL incluyen:

- **FILTER**: Permite aplicar condiciones para filtrar resultados.
- **ORDER BY**: Ordena los resultados según algún criterio.
- **GROUP BY**: Agrupa resultados según algún criterio.
- **OPTIONAL**: Permite que ciertas partes de la consulta sean opcionales.

Es importante mencionar que una consulta SPARQL puede llegar a ser bastante compleja dependiendo de las necesidades y del esquema de datos con el que se esté trabajando. Sin embargo, la estructura básica presentada aquí es un buen punto de partida para entender cómo funciona.

Veamos un ejemplo:

```
SELECT ?painter ?painterLabel ?painting ?paintingLabel ?style ?styleLabel  
WHERE {  
    ?painter wdt:P106 wd:Q1028181 ; # Occupation: Painter  
            rdfs:label ?painterLabel .  
    ?painting wdt:P170 ?painter ; # Creator: painter  
             rdfs:label ?paintingLabel ;  
             wdt:P135 ?style . # Movement: style  
    ?style rdfs:label ?styleLabel .  
    FILTER (LANG(?painterLabel) = "en" && LANG(?paintingLabel) = "en" && LA
```

```
}
```

LIMIT 50

### Explicación:

1. **SELECT**: Indica qué variables se desean recuperar en la consulta. En este caso, queremos obtener información sobre pintores (`?painter` y `?painterLabel`), sus pinturas (`?painting` y `?paintingLabel`), y el estilo o movimiento al que pertenecen esas pinturas (`?style` y `?styleLabel`).
2. **WHERE**: Define el patrón que debe coincidir con los datos en la base de datos RDF.
  - `?painter wdt:P106 wd:Q1028181 ;`: Este patrón busca recursos que tengan la ocupación (P106) de "Pintor" (Q1028181). Es decir, estamos buscando pintores.
  - `rdfs:label ?painterLabel .`: Para cada pintor encontrado, obtenemos su etiqueta (nombre usualmente) y lo almacenamos en la variable `?painterLabel`.
  - `?painting wdt:P170 ?painter ;`: Luego, para cada pintor, buscamos pinturas (`?painting`) que tengan como creador (`P170`) a dicho pintor.
  - `rdfs:label ?paintingLabel ;`: Obtenemos la etiqueta (nombre) de cada pintura y la almacenamos en `?paintingLabel`.
  - `wdt:P135 ?style .`: Además, para cada pintura, buscamos el movimiento o estilo (`P135`) al que pertenece y lo almacenamos en la variable `?style`.
  - `?style rdfs:label ?styleLabel .`: Por último, obtenemos la etiqueta (nombre) de cada estilo o movimiento y lo almacenamos en `?styleLabel`.
3. **FILTER**: Se utiliza para filtrar los resultados según ciertas condiciones. En este caso, la consulta está filtrando para que solo se devuelvan los resultados cuyas etiquetas (`?painterLabel`, `?paintingLabel` y `?styleLabel`) estén en inglés (`"en"`).
4. **LIMIT 50**: Limita los resultados a un máximo de 50 entradas.
5. **Prefijos**:
  - `wdt` : y `wd` : son prefijos de Wikidata.
  - `wd` : (Wikidata Entity) es el prefijo para las entidades/elementos de Wikidata. Ejemplo: `wd:Q1028181` representa la entidad "pintor" en

Wikidata.

- `wdt` : (Wikidata Property Truthy) es el prefijo para las propiedades "verdaderas" directas de Wikidata. Ejemplos:
  - `wdt:P106` (ocupación)
  - `wdt:P170` (creador)
  - `wdt:P135` (movimiento artístico)

## 6. Puntuación:

- a. El punto (.) en SPARQL tiene un significado sintáctico importante, ya que es un terminador de triplete (triple pattern). El punto indica el final de una declaración completa en RDF y separa diferentes patrones de tripletas en la consulta.
- b. El punto y coma (;) permite reutilizar el mismo sujeto
- c. Reglas Importantes: Cada patrón de triplete DEBE terminar con un punto. El punto y coma es opcional y se usa para abreviar. El último patrón de una serie SIEMPRE termina con punto, no punto y coma:

```
# Estructura completa
?painter wdt:P106 wd:Q1028181 ; # Mismo sujeto (?painter)
    rdfs:label ?painterLabel . # Termina con punto

?painting wdt:P170 ?painter ; # Nuevo sujeto (?painting)
    wdt:P135 ?style . # Termina con punto

?style rdfs:label ?styleLabel . # Nueva triplete independiente
```

En resumen, esta consulta busca pintores, sus pinturas y el estilo o movimiento de esas pinturas, y solo devuelve aquellos resultados cuyas etiquetas estén en español. Además, limita los resultados a un máximo de 50 registros.

Veamos este ejemplo en Python, utilizando la librería `SPARQLWrapper()`, consultando a la base de datos abierta de WikiData:

```
from SPARQLWrapper import SPARQLWrapper, XML
import xml.etree.ElementTree as ET

# Configurar el endpoint de Wikidata y la consulta SPARQL
```

```

sparql = SPARQLWrapper("https://query.wikidata.org/sparql")
sparql.setQuery("""
    SELECT ?painter ?painterLabel ?painting ?paintingLabel ?style ?styleLabel
    WHERE {
        ?painter wdt:P106 wd:Q1028181 ; # Occupation: Painter
        rdfs:label ?painterLabel .
        ?painting wdt:P170 ?painter ; # Creator: painter
        rdfs:label ?paintingLabel ;
        wdt:P135 ?style . # Movement: style
        ?style rdfs:label ?styleLabel .
        FILTER (LANG(?painterLabel) = "en" && LANG(?paintingLabel) = "en" &&
    }
    LIMIT 50
""")
sparql.setReturnFormat(XML)
results = sparql.query().convert()

# Convertir el objeto Document a una cadena
xml_string = results.toxml()

# Parsear el resultado XML
root = ET.fromstring(xml_string)

# El espacio de nombres (namespace) que usaremos para extraer los datos
namespace = '{http://www.w3.org/2005/sparql-results#}'

# Verificar la cantidad de resultados
print(len(root.findall(f".//{namespace}result")))

print('Autores, sus pinturas y estilos\n')
print('-----')

# Iterar sobre cada resultado y extraer los datos relevantes
for result in root.findall(f".//{namespace}result"):
    painter = result.find(f'.//{namespace}binding[@name="painterLabel"]/{nam
    painting = result.find(f'.//{namespace}binding[@name="paintingLabel"]/{na
    style = result.find(f'.//{namespace}binding[@name="styleLabel"]/{namespa

```

```
print(f'("{painter}", "has_painted", "{painting}")')
print(f'("{painting}", "has_style", "{style}")')
```

Obtendremos algo como esto:

Autores, sus pinturas y estilos

```
-----
("Salvador Dalí", "has_painted", "Portrait of Paul Éluard")
("Portrait of Paul Éluard", "has_style", "Dada")
("Leonardo da Vinci", "has_painted", "The Last Supper")
("The Last Supper", "has_style", "Renaissance")
("Man Ray", "has_painted", "Gift")
("Gift", "has_style", "Dada")
("George Grosz", "has_painted", "Pillars of Society")
("Pillars of Society", "has_style", "Dada")
("El Greco", "has_painted", "Altarpiece of Talavera la Vieja")
("Altarpiece of Talavera la Vieja", "has_style", "Renaissance")
("El Greco", "has_painted", "Altarpiece Santo Domingo Antiguo of the El Greco family pantheon")
("Altarpiece Santo Domingo Antiguo of the El Greco family pantheon", "has_style", "Renaissance")
("Leonardo da Vinci", "has_painted", "Virgin of the Rocks")
("Virgin of the Rocks", "has_style", "Renaissance")
...
...
```

El sitio de WikiData nos permite realizar consultas de forma interactiva (<https://query.wikidata.org/>) donde podemos realizar la misma consulta SPARQL que realizamos en el ejemplo. La librería **SPARQLWrapper** nos permite realizar consultas a bases de datos que soporten SPARQL como mecanismo de obtención de datos.

Otra forma de realizar consultas, pero de manera local, es usando **RDFLib**. Con esta librería podemos crear una base de datos “in-memory”, sin necesidad de un motor de base de datos remoto. Veamos un ejemplo:

```
from rdflib import Graph, Literal, Namespace, URIRef

# Crear un grafo vacío
g = Graph()

# Definir algunos espacios de nombres y recursos
```

```

n = Namespace("http://example.org/people/")
EX = Namespace("http://example.org/terms/")

# Añadir triplets al grafo
g.add((n.bob, EX.age, Literal(28)))
g.add((n.bob, EX.name, Literal("Bob")))
g.add((n.alice, EX.age, Literal(24)))
g.add((n.alice, EX.name, Literal("Alice")))

# Serializar y mostrar el grafo en formato "turtle" (opcional)
print(g.serialize(format="turtle"))

```

Ahora que hemos agregado grafos a nuestra base de datos en memoria, podemos también realizar búsquedas con SPARQL, como vemos aquí:

```

# Definir una consulta SPARQL para obtener todas las personas y sus edades
q1 = """
SELECT ?person ?age WHERE {
    ?person <http://example.org/terms/age> ?age .
}
"""

# Ejecutar la consulta
results = g.query(q1)

# Imprimir los resultados
print('Query 1:')
for r in results:
    print(f"{r['person']} tiene {r['age']} años.")

# Definir otra consulta SPARQL para obtener personas mayores de 25 años
q2 = """
SELECT ?person WHERE {
    ?person <http://example.org/terms/age> ?age .
    FILTER(?age > 25)
}
"""

```

```

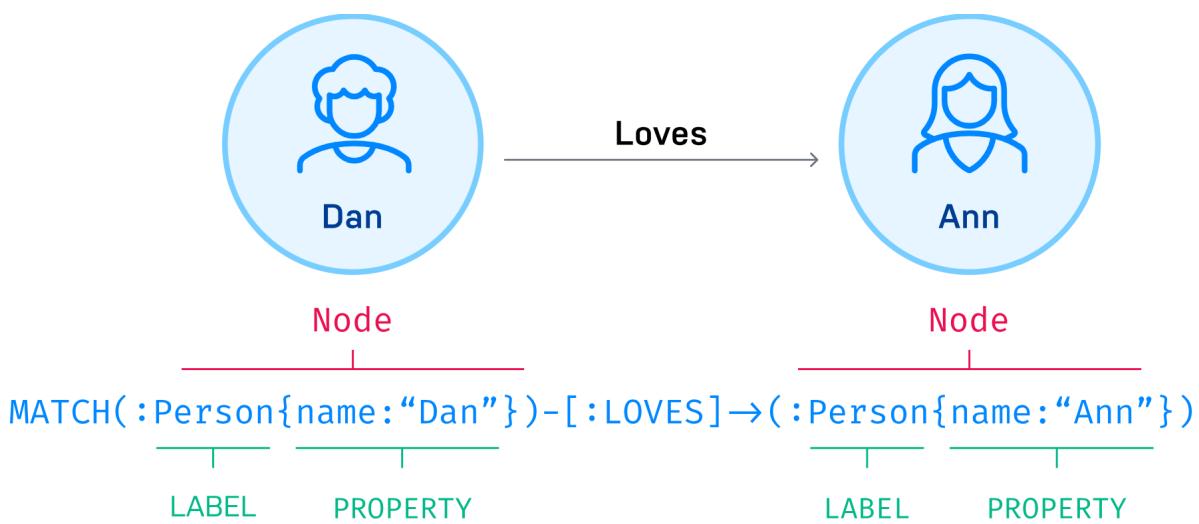
# Ejecutar la consulta
results = g.query(q2)

# Imprimir los resultados
print('Query 2:')
for r in results:
    print(f"{r['person']} es mayor de 25 años.")

```

## Lenguaje Cypher

La consulta Cypher funciona de manera similar a otros lenguajes de consulta basados en grafos, como SPARQL, pero está diseñada específicamente para bases de datos de grafos como RedisGraph o Neo4j. A continuación, vemos la estructura básica de una consulta Cypher:



Cypher, a diferencia de los lenguajes SQL que están optimizados para bases de datos relacionales, permite consultar y manipular grafos de manera natural, enfocándose en relaciones y nodos. Los grafos están compuestos por nodos (entidades o personas) y relaciones (conexiones entre los nodos).

Con Cypher, puedes realizar una variedad de operaciones, como:

- Buscar nodos y relaciones con patrones específicos.
- Crear, modificar o eliminar nodos y relaciones.
- Definir propiedades y aplicar filtros para obtener datos precisos.

Cypher utiliza una sintaxis muy visual e intuitiva, donde los **nodos** se representan con paréntesis `()`, las **relaciones** con corchetes `[]`, y se utiliza una flecha `→` para indicar la dirección de la relación.

## Explicación de la consulta

```
MATCH (:Person {name:"Dan"})-[:LOVES]→(:Person {name:"Ann"})
```

- **MATCH:** `MATCH` es la palabra clave que se usa para buscar un patrón en el grafo. Lo que sigue después de `MATCH` es la definición del patrón que se quiere buscar.
- **(:Person):** Busca nodos con la etiqueta `Person`. Esto significa que estamos buscando un nodo que represente a una persona.
- **{name: "Dan"}:** Este bloque de propiedades es un filtro que dice que el nodo `Person` debe tener una propiedad `name` con el valor `"Dan"`. En otras palabras, estamos buscando específicamente a una persona llamada "Dan".
- **[]→:** Representa una relación entre dos nodos. Aquí, estamos buscando una relación llamada `LOVES`. Esta relación indica que un nodo tiene algún tipo de conexión de amor con otro nodo. La flecha `>` indica la dirección de la relación. En este caso, estamos buscando nodos donde "Dan" ama a otro nodo, en lugar de ser amado por otro.
- **{name:"Ann"}:** De nuevo, esto busca un nodo con la etiqueta `Person`. `{name: "Ann"}` es el filtro para que el nodo `Person` tenga una propiedad `name` con el valor `"Ann"`. Así que estamos buscando específicamente a una persona llamada "Ann".

Esta consulta busca una relación **LOVES** en la base de datos donde: "**Dan** ama a **Ann**."

El patrón indica que Dan tiene una relación con Ann, y esa relación está etiquetada como `LOVES`. El nodo que representa a Dan está conectado a un nodo que representa a Ann, y la relación está dirigida desde Dan hacia Ann, no en la otra dirección. Esta consulta es útil cuando queremos encontrar conexiones específicas entre nodos. En este caso, estás buscando un tipo de relación (LOVES) entre dos personas. Este tipo de consulta se puede ampliar para:

- **Buscar relaciones bidireccionales:** Puedes usar patrones sin flechas para buscar relaciones sin importar la dirección.
- **Buscar diferentes tipos de relaciones:** Cambiar la etiqueta de la relación para buscar otras conexiones, como FRIENDS , KNOWS , etc.
- **Aplicar filtros más avanzados:** Puedes combinar más propiedades de los nodos y las relaciones para realizar búsquedas más específicas.

Otros ejemplos de Cypher:

```
# 1. Esta consulta busca todos los nodos con la etiqueta `Person` que tienen
# una relación `KNOWS` con el nodo que representa a "Alice".
# Luego devuelve los nombres de todas las personas que conocen a Alice.
MATCH (p:Person)-[:KNOWS]→(alice:Person {name: 'Alice'}) RETURN p.name

# 2. Esta consulta cuenta el número total de nodos con la etiqueta `Person` en
# El resultado es el número de personas presentes en la base de datos.
MATCH (p:Person) RETURN COUNT(p)

# 3. Aquí se buscan todas las personas (`Person`) cuyo atributo `age` (edad) es
# mayor a 30. Luego devuelve sus nombres y edades. Esta consulta permite filtrar
# nodos basados en una propiedad específica.
MATCH (p:Person) WHERE p.age > 30 RETURN p.name, p.age

# 4. Esta consulta busca personas (`Person`) que conocen a alguien, quien a su vez
# conoce a "Bob". El objetivo es encontrar a las personas que están indirectamente
# conectadas con Bob a través de una cadena de relaciones `KNOWS`.
MATCH (p:Person)-[:KNOWS]→(:Person)-[:KNOWS]→(bob:Person {name: 'Bob'})

# 5. Esta consulta encuentra a "Alice" en el grafo y actualiza su edad a 31.
# Luego, devuelve el nombre y la nueva edad de Alice. Es un ejemplo de cómo
# pueden modificar los datos de un nodo.
MATCH (a:Person {name: 'Alice'}) SET a.age = 31 RETURN a.name, a.age

# 6. Aquí se buscan todas las relaciones `KNOWS` entre personas en ambas direcciones.
# Devuelve los nombres de las dos personas conectadas por cada relación `KNOWS`
# sin importar la dirección de la relación.
MATCH (p1:Person)-[:KNOWS]-(p2:Person) RETURN p1.name, p2.name
```

## Ejemplos:

- **RedisGraph**

Google Colab

🔗 [https://colab.research.google.com/drive/14rv\\_eRloTaZfb2HOFUqe36N2hxLzUTaS?usp=sharing](https://colab.research.google.com/drive/14rv_eRloTaZfb2HOFUqe36N2hxLzUTaS?usp=sharing)



- **Neo4j**

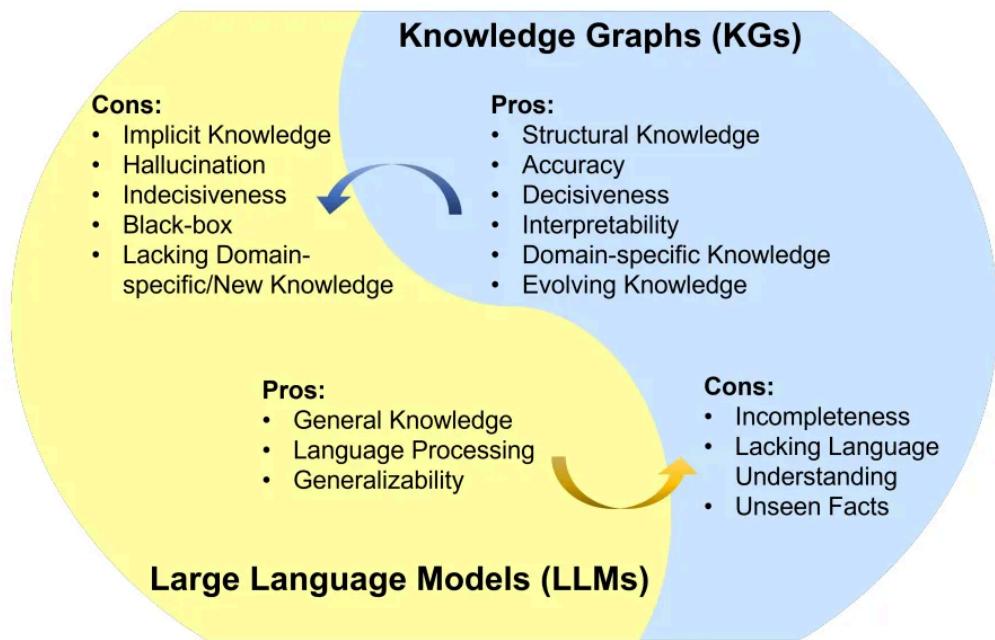
Google Colab

🔗 [https://colab.research.google.com/drive/1gj2NbdCyS5nhEofjzj7f\\_Vcbx\\_dapYWa?usp=sharing](https://colab.research.google.com/drive/1gj2NbdCyS5nhEofjzj7f_Vcbx_dapYWa?usp=sharing)



## Utilización de GDB con LLMs

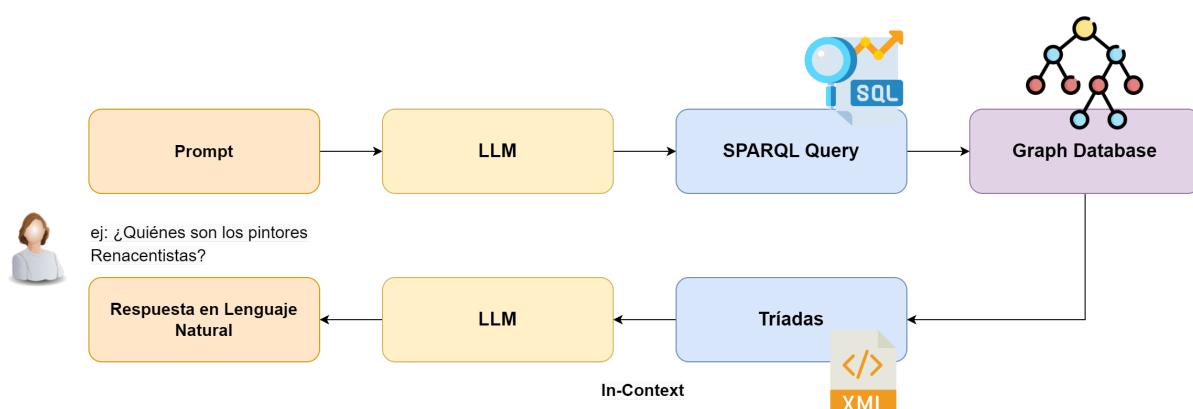
Como señalan Shirui Pan y colaboradores en su paper, "Unificando Modelos de Lenguaje a Gran Escala y Grafos de Conocimiento: Una Hoja de Ruta", los LLMs y las KGs pueden complementarse. Algunas de las principales debilidades de los LLM, que son modelos de caja negra y tienen dificultades con el conocimiento factual, son algunas de las mayores fortalezas de los KGs. Los KGs son, esencialmente, colecciones de hechos y son completamente interpretables.



Las LLMs y las KGs pueden complementarse de diversas maneras, para aprovechar sus potenciales. Shirui Pan et al., 2023. <<https://arxiv.org/abs/2306.08302>>

Shirui y colaboradores proponen muchas formas potenciales en las que los LLM y los KGs pueden complementarse mutuamente.

Por ejemplo, la utilización de Modelos de Lenguaje a Gran Escala (LLM por sus siglas en inglés, como GPT-3 o GPT-4) pueden generar consultas en SPARQL a partir de lenguaje natural y luego procesar las respuestas en forma de tríadas RDF para usar la información como parte del contexto de una conversación. Veamos cómo se puede lograr esto:



## 1. Generación de consultas SPARQL a partir de lenguaje natural:

- Prompt Engineering:** Por medio de ingeniería de prompts, podemos solicitar al LLM que genere una consulta en SPARQL a partir de una

pregunta del usuario en lenguaje natural.

2. **Generación de consulta:** Al proporcionar una frase en lenguaje natural al modelo, este generará una consulta SPARQL correspondiente que se utilizará para realizar una consulta a una base de datos de conocimiento.
3. **Procesamiento de tríadas:** Una vez que se obtiene una respuesta en forma de tríadas RDF de una consulta SPARQL de la base de datos, se pueden procesar esas tríadas para extraer la información esencial. Por ejemplo, una tríada podría ser **(BarackObama, presidentOf, USA)**, que indica que Barack Obama fue presidente de EE.UU.

## 2. Conversión de tríadas RDF en información contextual:

1. **Conversión a lenguaje natural:** Usando un LLM, se puede convertir la tríada procesada en una frase coherente en lenguaje natural, como "Barack Obama fue presidente de EE.UU."
2. **Contextualización:** Si la conversación requiere más contexto o detalles, el modelo puede utilizar la tríada como punto de partida y generar oraciones adicionales para proporcionar más información o responder a preguntas de seguimiento.

## 3. Realizar una conversación en lenguaje natural:

1. **Respuestas dinámicas:** Basándose en las tríadas y en el contexto de la conversación, el LLM puede generar respuestas dinámicas. Por ejemplo, si se pregunta "¿Quién sucedió a Barack Obama?", el modelo podría responder "Donald Trump" basándose en la información de la base de datos y el contexto de la conversación anterior.
2. **Iteración:** A medida que el usuario sigue haciendo preguntas o solicitando detalles, el LLM puede iterar sobre las tríadas disponibles y el contexto de la conversación para proporcionar respuestas cada vez más detalladas o específicas.