✨ Member-only story

A Simpler Way to Query Neo4j Knowledge Graphs

Exploring the new Llama Pack for Neo4j query engine

Wenqi Glantz · Follow

Published in Level Up Coding

7 min read · Dec 7, 2023

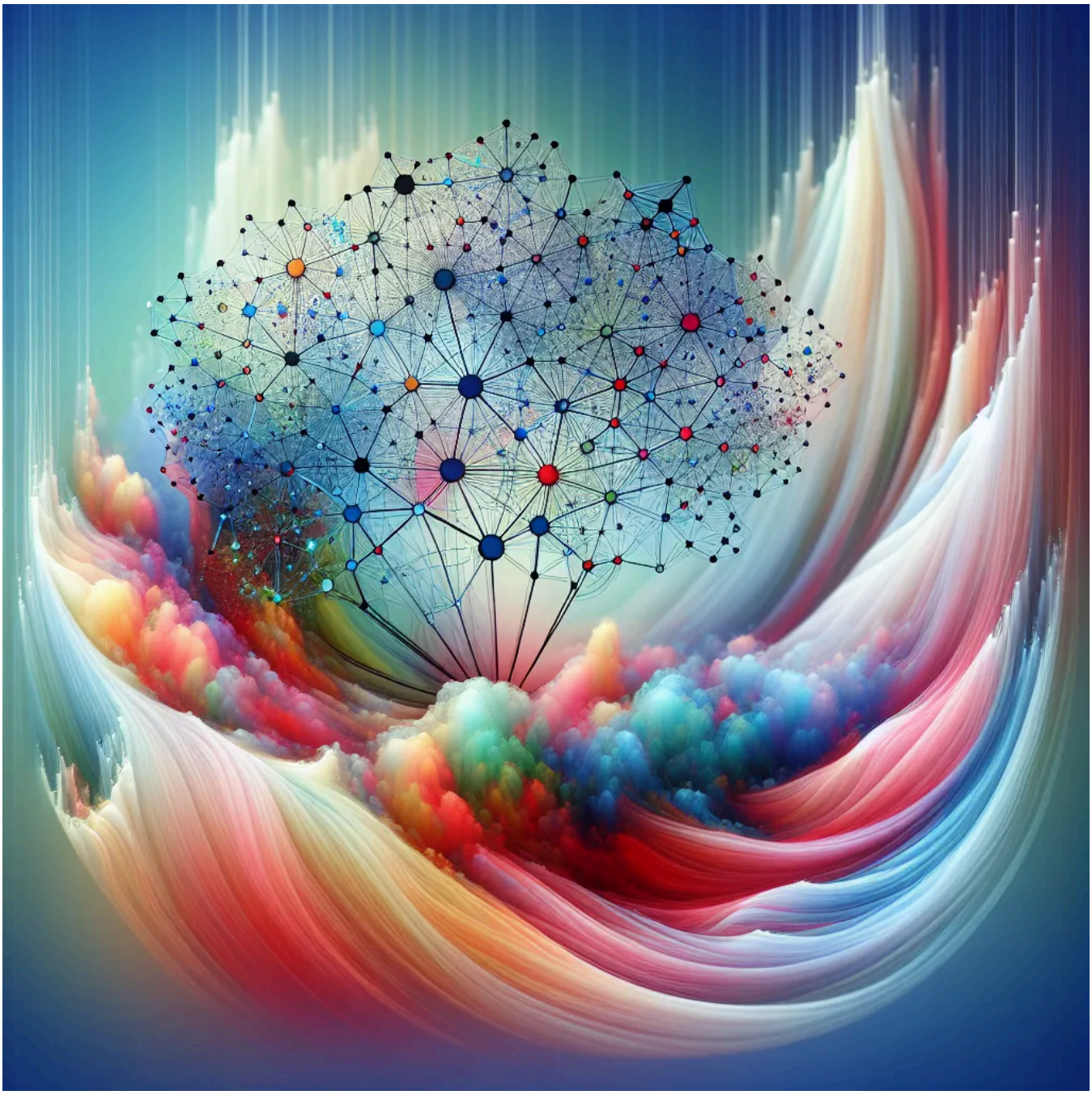


Image by DALL-E 3

We explored 7 Query Strategies for Navigating Knowledge Graphs With LlamaIndex a few months ago, in which we learned the various query strategies we can apply in querying NebulaGraph. In this article, we are going to experiment with the brand new Llama Pack for Neo4j and how to apply the 7 query strategies in Neo4j using this Llama Pack.

First, let's take a high-level look at NebulaGraph and Neo4j.

NebulaGraph vs Neo4j

NebulaGraph and Neo4j are popular graph databases, each with strengths and weaknesses. Here's a high-level comparison of the two:

Scalability:

- **NebulaGraph**: Horizontally scalable with a shared-nothing architecture. NebulaGraph is a distributed graph database, meaning it can scale horizontally by adding more nodes to the cluster. It supports the automatic sharding of data across multiple servers. This makes it well-suited for handling large datasets with billions of vertices and trillions of edges.
- **Neo4j**: Neo4j scales out as data grows with sharding. This technique divides a single logical database into several smaller databases (shards). Running your shards on Autonomous Clusters allows you to achieve unlimited horizontal scalability for a very large graph.

Performance:

- **NebulaGraph**: NebulaGraph is designed for high performance, particularly for large datasets. It utilizes a variety of optimization techniques to achieve fast query execution.
- **Neo4j**: Neo4j also offers good performance, especially for smaller datasets. However, it may not be as efficient as NebulaGraph for very large graphs. Per this DZ article on the performance comparison of NebulaGraph, Neo4j, and HughGraph, in terms of data import, NebulaGraph is a bit slower than Neo4j when the data size is small. However, NebulaGraph is much faster than the other two when the data size is large. For the three graph queries, NebulaGraph shows clearly better performance compared to Neo4j.

Query Language:

- **NebulaGraph**: NebulaGraph uses a query language called nGQL, based on openCypher. nGQL is similar to SQL but specifically designed for graph databases. It provides a rich set of features for traversing and analyzing graph data.
- **Neo4j**: Neo4j uses a query language called Cypher, which is also a graph-specific language. Cypher is widely used and has a large community of developers.

Use Cases:

- **NebulaGraph**: NebulaGraph is well-suited for applications that require high scalability and performance, such as social network analysis, fraud detection, and knowledge graph management.
- **Neo4j**: Neo4j is a good choice for applications that need a balance of performance and ease of use, such as real-time recommendations, master data management, and identity and access management.

Deployment:

- **NebulaGraph**: NebulaGraph can be deployed on-premises or in the cloud. It offers a variety of deployment options, including Docker and Kubernetes.
- **Neo4j**: Neo4j can also be deployed on-premises or in the cloud. It provides a managed cloud service called Neo4j Aura.

Community and Support:

- **NebulaGraph**: NebulaGraph has a growing community of users and developers. It offers documentation, tutorials, and support forums.
- **Neo4j**: Neo4j has a larger and more established community. It offers extensive documentation, training courses, and commercial support options.

Overall, NebulaGraph is a good choice for applications that require high scalability and performance, while Neo4j is a good choice for applications that need a balance of performance and ease of use.

Now, let's dive into the 7 strategies for querying Neo4j knowledge graphs (KG).

7 Strategies for Querying Neo4j Knowledge Graphs

The same 7 query strategies for NebulaGraph can be applied for Neo4j as well. Please refer to the above link for a detailed explanation of each of the 7 query strategies. Let's briefly recap these 7 strategies from a high level:

KG vector-based entity retrieval

This query strategy looks up KG entities with vector similarity, pulls in linked text chunks and optionally explores relationships. See the sample code snippet below (details on how `neo4j_index` is defined can be found in the next section).

```
query_engine = neo4j_index.as_query_engine()
```

KG keyword-based entity retrieval

This query strategy uses keywords to search for relevant KG entities. See the sample code snippet below for defining this query strategy.

```
query_engine = neo4j_index.as_query_engine(
    # setting to false uses the raw triplets instead of adding the text from th
    include_text=False,
    retriever_mode="keyword",
    response_mode="tree_summarize",
)
```

KG hybrid entity retrieval

This query strategy uses both vector-based and keyword-based entity retrieval to search for relevant nodes and documents in a KG. See the sample snippet as follows:

```
query_engine = neo4j_index.as_query_engine(
    include_text=True,
    response_mode="tree_summarize",
    embedding_mode="hybrid",
    similarity_top_k=3,
    explore_global_knowledge=True,
)
```

Raw vector index retrieval

This query strategy uses a vector index to search for relevant documents, removes relationships, and represents entities in a flat vector store. See the sample snippet as follows:

```
query_engine = vector_index.as_query_engine()
```

Custom Combo query engine

This query strategy uses both vector similarity and KG entity retrieval (vector-based, keyword-based, hybrid). See the sample snippet as follows:

```
neo4j_vector_retriever = VectorIndexRetriever(index=vector_index)
neo4j_kg_retriever = KGTableRetriever(
    index=neo4j_index, retriever_mode="keyword", include_text=False
)
neo4j_custom_retriever = CustomRetriever(
    neo4j_vector_retriever, neo4j_kg_retriever
)

# create neo4j response synthesizer
neo4j_response_synthesizer = get_response_synthesizer(
    service_context=service_context,
    response_mode="tree_summarize",
)

# Custom combo query engine
query_engine = RetrieverQueryEngine(
    retriever=neo4j_custom_retriever,
    response_synthesizer=neo4j_response_synthesizer,
)
```

KnowledgeGraphQueryEngine

This query strategy queries knowledge graphs using natural language, Text2Cypher. See the sample snippet as follows:

```
query_engine = KnowledgeGraphQueryEngine(
    storage_context=neo4j_storage_context,
    service_context=service_context,
    llm=llm,
    verbose=True,
)
```

KnowledgeGraphRAGRetriever

This query strategy was designed to work with knowledge graphs built using RAG. See the sample snippet as follows:

```
neo4j_graph_rag_retriever = KnowledgeGraphRAGRetriever(
    storage_context=neo4j_storage_context,
```

```
    service_context=service_context,
    llm=llm,
    verbose=True,
)

query_engine = RetrieverQueryEngine.from_args(
    neo4j_graph_rag_retriever, service_context=service_context
)
```

Packaging the 7 Strategies into a Llama Pack

To templatize query strategies for Neo4j, let's package them into a Llama Pack. In my previous article, Llama Packs: The Low-Code Solution to Building Your LLM Apps, we explored what a Llama Pack is, why we need it, and how to use it. Let's dive in to see how simple it is to package a new Llama Pack for querying the Neo4j knowledge graphs.

Neo4jQueryEngineType

This is the `Enum` class for the 7 query engine types, very straightforward. The KG vector-based entity retrieval is the default type, thus it is omitted in this class.

```
class Neo4jQueryEngineType(str, Enum):
    """Neo4j query engine type"""

    KG_KEYWORD = "keyword"
    KG_HYBRID = "hybrid"
    RAW_VECTOR = "vector"
    RAW_VECTOR_KG_COMBO = "vector_kg"
    KG_QE = "KnowledgeGraphQueryEngine"
    KG_RAG_RETRIEVER = "KnowledgeGraphRAGRetriever"
```

Neo4jQueryEnginePack

This is the main class for our new Llama Pack. A few key steps in its `__init__` function:

- Define `neo4j_graph_store`
- Construct knowledge graph index `neo4j_index`
- Construct vector store index `vector_index`
- Based on the `query_engine_type` passed in, define the respective `query_engine`

```
class Neo4jQueryEnginePack(BaseLlamaPack):
    """Neo4j Query Engine pack."""

    def __init__(
        self,
        username: str,
        password: str,
        url: str,
        database: str,
        docs: List[Document],
        query_engine_type: Optional[Neo4jQueryEngineType] = None,
        **kwargs: Any,
    ) -> None:
        """Init params."""

        neo4j_graph_store = Neo4jGraphStore(
            username=username,
            password=password,
            url=url,
            database=database,
        )

        neo4j_storage_context = StorageContext.from_defaults(
            graph_store=neo4j_graph_store
        )

        # define LLM
        self.llm = OpenAI(temperature=0.1, model="gpt-3.5-turbo")
        self.service_context = ServiceContext.from_defaults(llm=self.llm)

        neo4j_index = KnowledgeGraphIndex.from_documents(
            documents=docs,
            storage_context=neo4j_storage_context,
            max_triplets_per_chunk=10,
            service_context=self.service_context,
            include_embeddings=True,
        )

        # create node parser to parse nodes from document
        node_parser = SentenceSplitter(chunk_size=512)

        # use transforms directly
        nodes = node_parser(docs)
        print(f"loaded nodes with {len(nodes)} nodes")

        # based on the nodes and service_context, create index
        vector_index = VectorStoreIndex(
            nodes=nodes, service_context=self.service_context
        )

        if query_engine_type == Neo4jQueryEngineType.KG_KEYWORD:
```

```
        # KG keyword-based entity retrieval
        self.query_engine = neo4j_index.as_query_engine(
            # setting to false uses the raw triplets instead of adding the
            include_text=False,
            retriever_mode="keyword",
            response_mode="tree_summarize",
        )

    elif query_engine_type == Neo4jQueryEngineType.KG_HYBRID:
        # KG hybrid entity retrieval
        self.query_engine = neo4j_index.as_query_engine(
            include_text=True,
            response_mode="tree_summarize",
            embedding_mode="hybrid",
            similarity_top_k=3,
            explore_global_knowledge=True,
        )

    elif query_engine_type == Neo4jQueryEngineType.RAW_VECTOR:
        # Raw vector index retrieval
        self.query_engine = vector_index.as_query_engine()

    elif query_engine_type == Neo4jQueryEngineType.RAW_VECTOR_KG_COMBO:
        from llama_index.query_engine import RetrieverQueryEngine

        # create neo4j custom retriever
        neo4j_vector_retriever = VectorIndexRetriever(index=vector_index)
        neo4j_kg_retriever = KGTableRetriever(
            index=neo4j_index, retriever_mode="keyword", include_text=False
        )
        neo4j_custom_retriever = CustomRetriever(
            neo4j_vector_retriever, neo4j_kg_retriever
        )

        # create neo4j response synthesizer
        neo4j_response_synthesizer = get_response_synthesizer(
            service_context=self.service_context,
            response_mode="tree_summarize",
        )

        # Custom combo query engine
        self.query_engine = RetrieverQueryEngine(
            retriever=neo4j_custom_retriever,
            response_synthesizer=neo4j_response_synthesizer,
        )

    elif query_engine_type == Neo4jQueryEngineType.KG_QE:
        # using KnowledgeGraphQueryEngine
        from llama_index.query_engine import KnowledgeGraphQueryEngine

        self.query_engine = KnowledgeGraphQueryEngine(
            storage_context=neo4j_storage_context,
            service_context=self.service_context,
```

```
                llm=self.llm,
                verbose=True,
            )

        elif query_engine_type == Neo4jQueryEngineType.KG_RAG_RETRIEVER:
            # using KnowledgeGraphRAGRetriever
            from llama_index.query_engine import RetrieverQueryEngine
            from llama_index.retrievers import KnowledgeGraphRAGRetriever

            neo4j_graph_rag_retriever = KnowledgeGraphRAGRetriever(
                storage_context=neo4j_storage_context,
                service_context=self.service_context,
                llm=self.llm,
                verbose=True,
            )

            self.query_engine = RetrieverQueryEngine.from_args(
                neo4j_graph_rag_retriever, service_context=self.service_context
            )

        else:
            # KG vector-based entity retrieval
            self.query_engine = neo4j_index.as_query_engine()

    def get_modules(self) -> Dict[str, Any]:
        """Get modules."""
        return {
            "llm": self.llm,
            "service_context": self.service_context,
            "query_engine": self.query_engine,
        }

    def run(self, *args: Any, **kwargs: Any) -> Any:
        """Run the pipeline."""
        return self.query_engine.query(*args, **kwargs)
```

Usage Pattern

CLI Usage

You can download Llama Packs directly using `llamaindex-cli`, which comes installed with the `llama-index` python package:

```
llamaindex-cli download-llamapack Neo4jQueryEnginePack --download-dir ./neo4j_p
```

You can then inspect the files at `./neo4j_pack` and use them as a template or for customization.

Code Usage

You can download the pack to a `./neo4j_pack` directory:

```
from llama_index.llama_pack import download_llama_pack

# download and install dependencies
Neo4jQueryEnginePack = download_llama_pack(
  "Neo4jQueryEnginePack", "./neo4j_pack"
)
```

From here, you can use the pack, or inspect and modify the pack in `./neo4j_pack`. Refer to the "Extending Llama Packs" section in our last article for detailed instructions on how to customize the pack.

Then, you can set up the pack like so:

```
# Load the docs (example of Paleo diet from Wikipedia)
from llama_index import download_loader

WikipediaReader = download_loader("WikipediaReader")
loader = WikipediaReader()
docs = loader.load_data(pages=['Paleolithic diet'], auto_suggest=False)
print(f'Loaded {len(docs)} documents')

# get Neo4j credentials (assume it's stored in credentials.json)
with open('credentials.json') as f:
  neo4j_connection_params = json.load(f)
  username = neo4j_connection_params['username']
  password = neo4j_connection_params['password']
  url = neo4j_connection_params['url']
  database = neo4j_connection_params['database']

# create the pack
neo4j_pack = Neo4jQueryEnginePack(
  username = username,
  password = password,
  url = url,
  database = database,
  docs = docs,
```

```
    query_engine_type = Neo4jQueryEngineType.KG_HYBRID
)

# run the pack
response = neo4j_pack.run("Tell me about the benefits of paleo diet.")
```

Optionally, you can pass in the `query_engine_type` from `Neo4jQueryEngineType` to construct `Neo4jQueryEnginePack`. If `query_engine_type` is not defined, it defaults to Knowledge Graph vector-based entity retrieval.

Summary

We explored the brand new Llama Pack for Neo4j query engine in this article. Using the prepackaged template, querying Neo4j knowledge graphs is now much simplified. By passing in the value for `Neo4jQueryEngineType` when constructing the pack, it routes the queries to their respective query engines. Do feel free to explore this new pack and customize it as you see fit. I hope you find this article helpful.

Please refer to my Colab notebook for a sample on how to use the Neo4j query engine Llama Pack.

Happy coding!

Reference:

- 7 Query Strategies for Navigating Knowledge Graphs With LlamaIndex
- Llama Packs: The Low-Code Solution to Building Your LLM Apps
- Neo4j Graph Store
- Custom Retriever combining KG Index and Vector Store Index
- Scaling Neo4j Graph Database
- NebulaGraph Database Architecture — A Bird's Eye View
- Performance Comparison: Neo4j vs Nebula Graph vs JanusGraph

Llamaindex Neo4j Nebula Graph Knowledge Graph

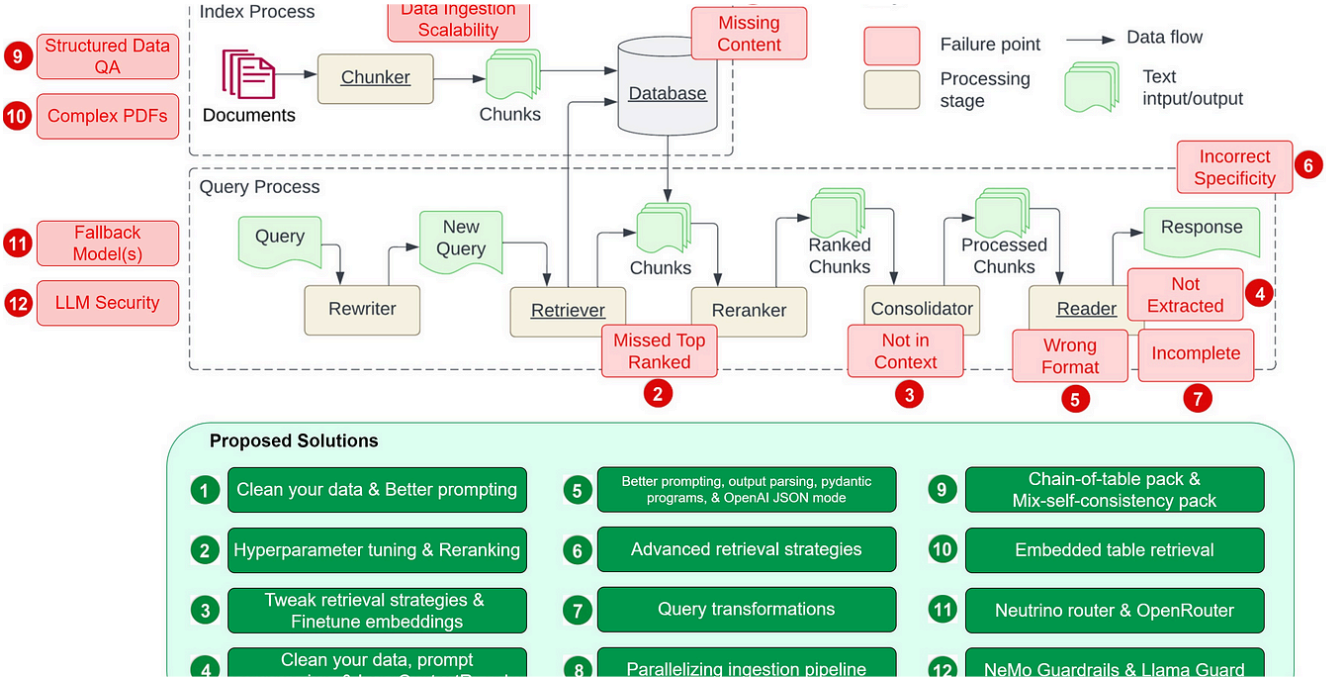
Follow

Written by Wenqi Glantz

7.9K Followers · Writer for Level Up Coding

Mom, wife, architect with a passion for technology and crafting quality products linkedin.com/in/wenqi-glantz-b5448a5a/ twitter.com/wenqi_glantz

More from Wenqi Glantz and Level Up Coding



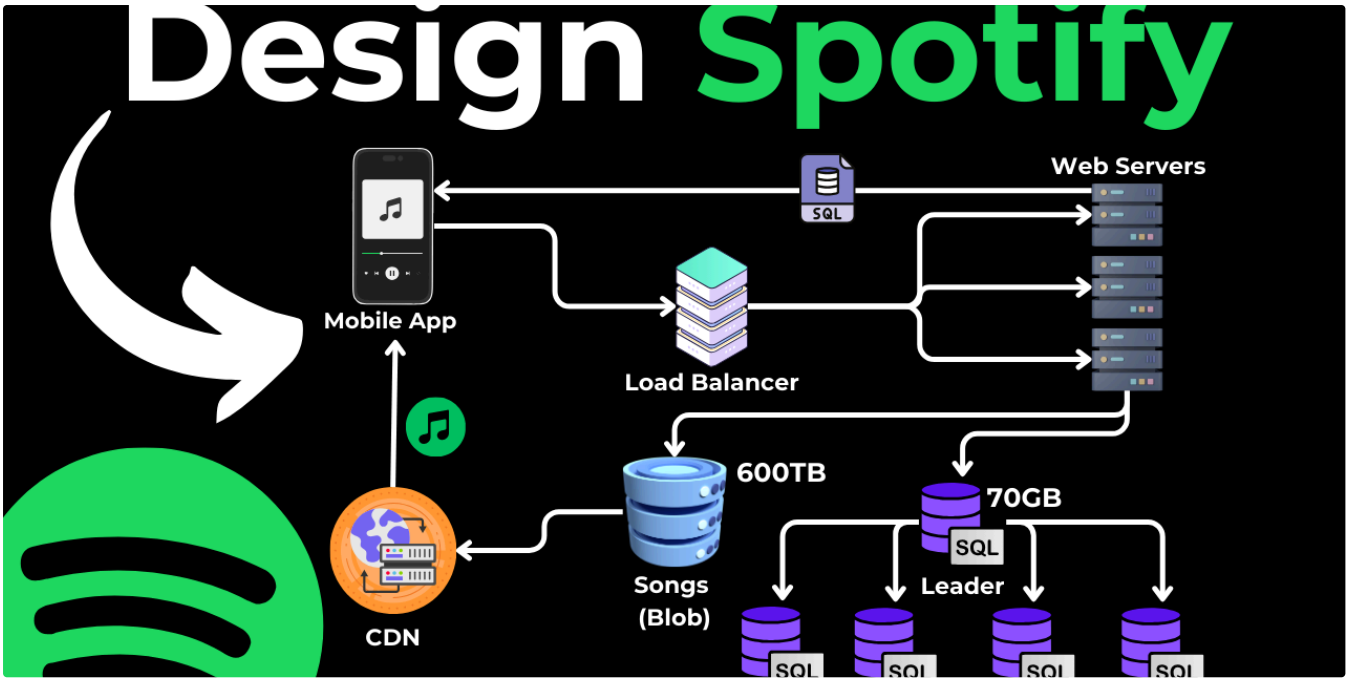
Wenqi Glantz in Towards Data Science

12 RAG Pain Points and Proposed Solutions

Solving the core challenges of Retrieval-Augmented Generation

✨ · 18 min read · Jan 30, 2024

-- 9



Hayk Simonyan in Level Up Coding

System Design Interview Question: Design Spotify

High-level overview of a System Design Interview Question - Design Spotify.

6 min read · Feb 20, 2024

-- 29

Tirendaz AI in Level Up Coding

How to Use ChatGPT in Daily Life?

Save time and money using ChatGPT

9 min read · Apr 3, 2023

-- 88

Wenqi Glantz in Towards Data Science

The Journey of RAG Development: From Notebook to Microservices

Converting a Colab notebook to two microservices with support for Milvus and NeMo Guardrails

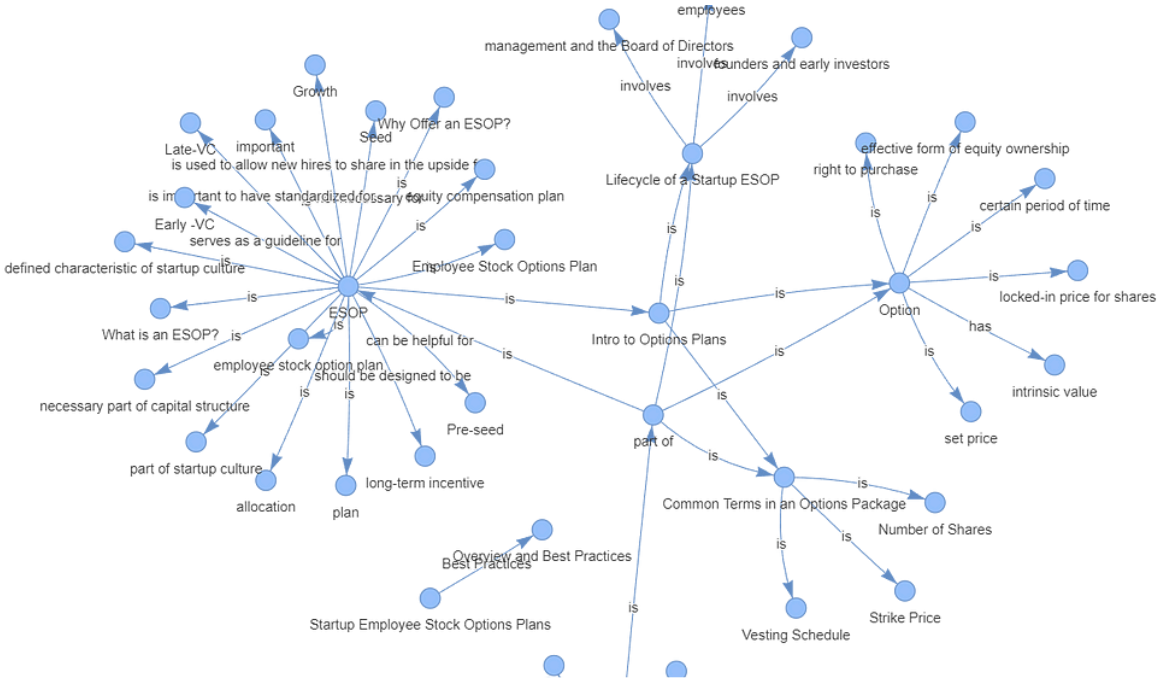
· 11 min read · Feb 21, 2024

-- 3

See all from Wenqi Glantz

See all from Level Up Coding

Recommended from Medium



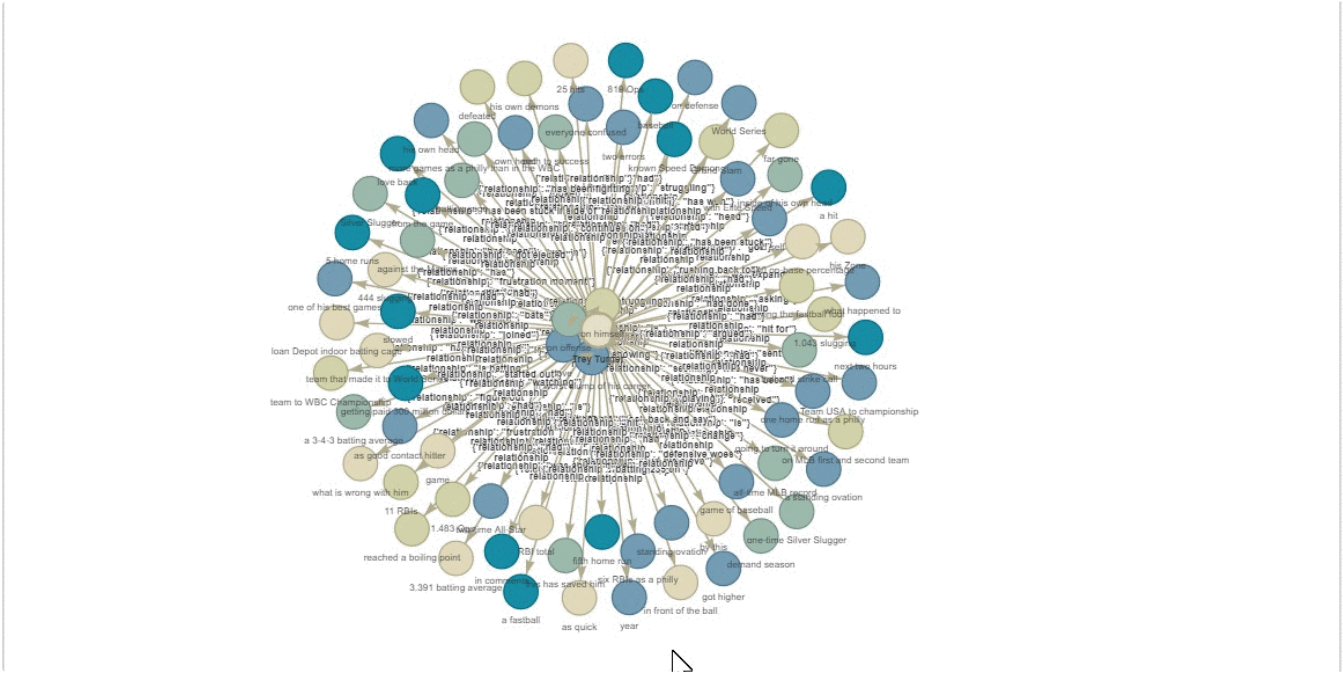
Plaban Nayak in AI Planet

Implement RAG with Knowledge Graph and Llama-Index

Hallucination is a common problem when working with large language models (LLMs). LLMs generate fluent and coherent text but often generate...

25 min read · Dec 3, 2023

-- 9



Wenqi Glantz in Better Programming

7 Query Strategies for Navigating Knowledge Graphs With LlamaIndex

Exploring NebulaGraph RAG Pipeline with the Philadelphia Phillies

· 17 min read · Sep 29, 2023

-- 4

Lists

data science and AI

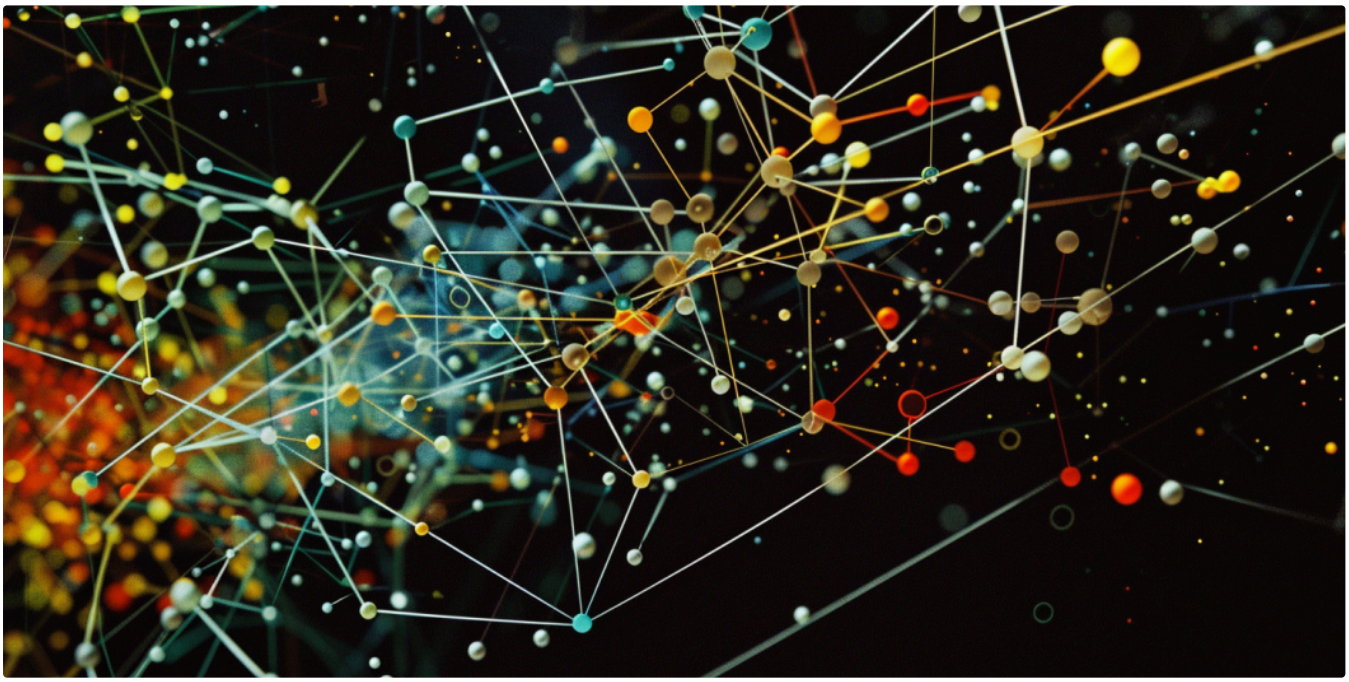
40 stories · 105 saves

Natural Language Processing

1293 stories · 782 saves

Productivity

237 stories · 358 saves



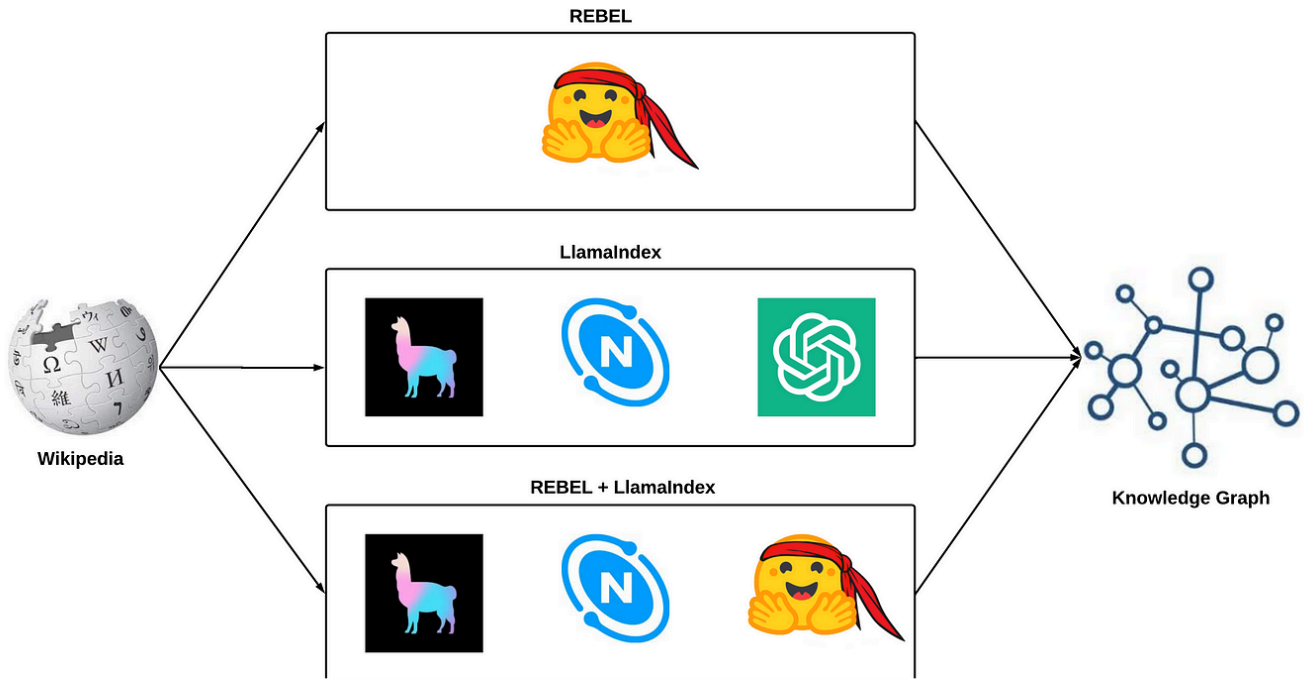
Mirko Peters in Mirko Peters—Data & Analytics Blog

Understanding Named Graphs in RDF Databases: Querying Semantic Web with RDF Graph Literals

The Semantic Web, a vision that seeks to structure and give meaning to the vast amounts of information on the internet, leverages several...

· 11 min read · Mar 2, 2024

--



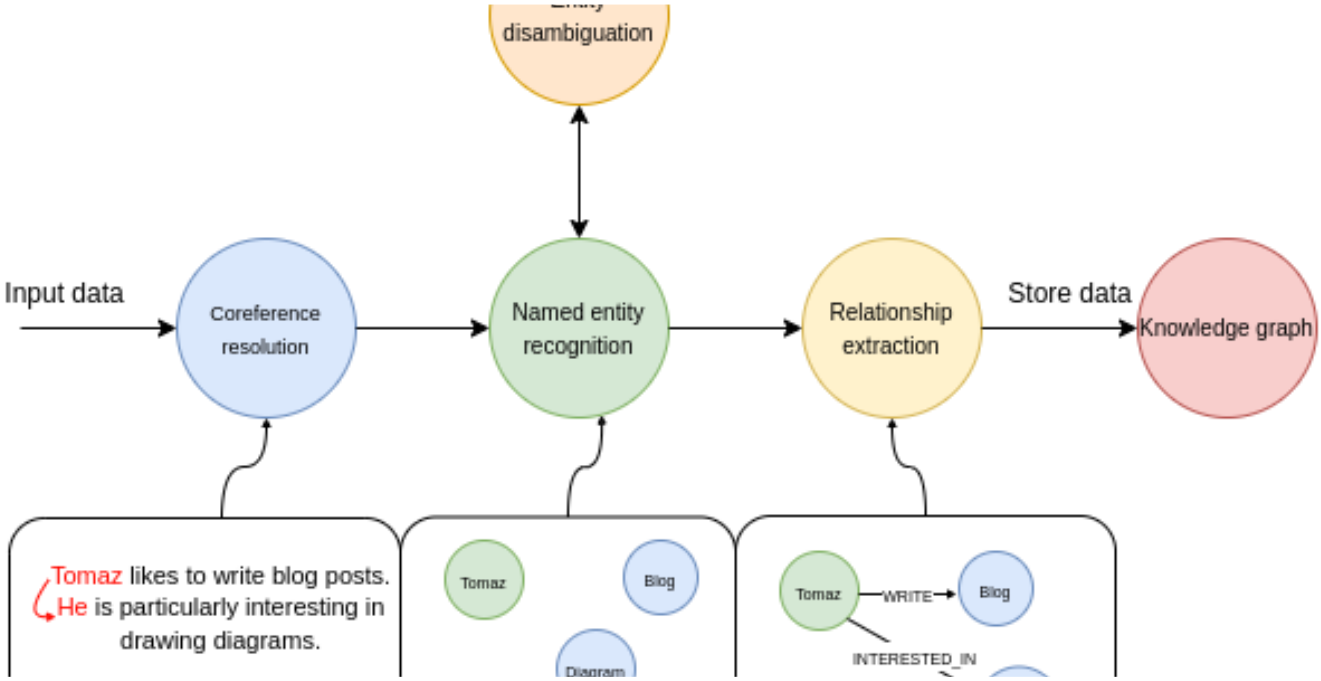
Saurav Joshi

Building Knowledge Graphs: REBEL, LlamaIndex, and REBEL + LlamaIndex

Exploring building knowledge graphs using LlamaIndex and NebulaGraph

10 min read · Oct 3, 2023

-- 5



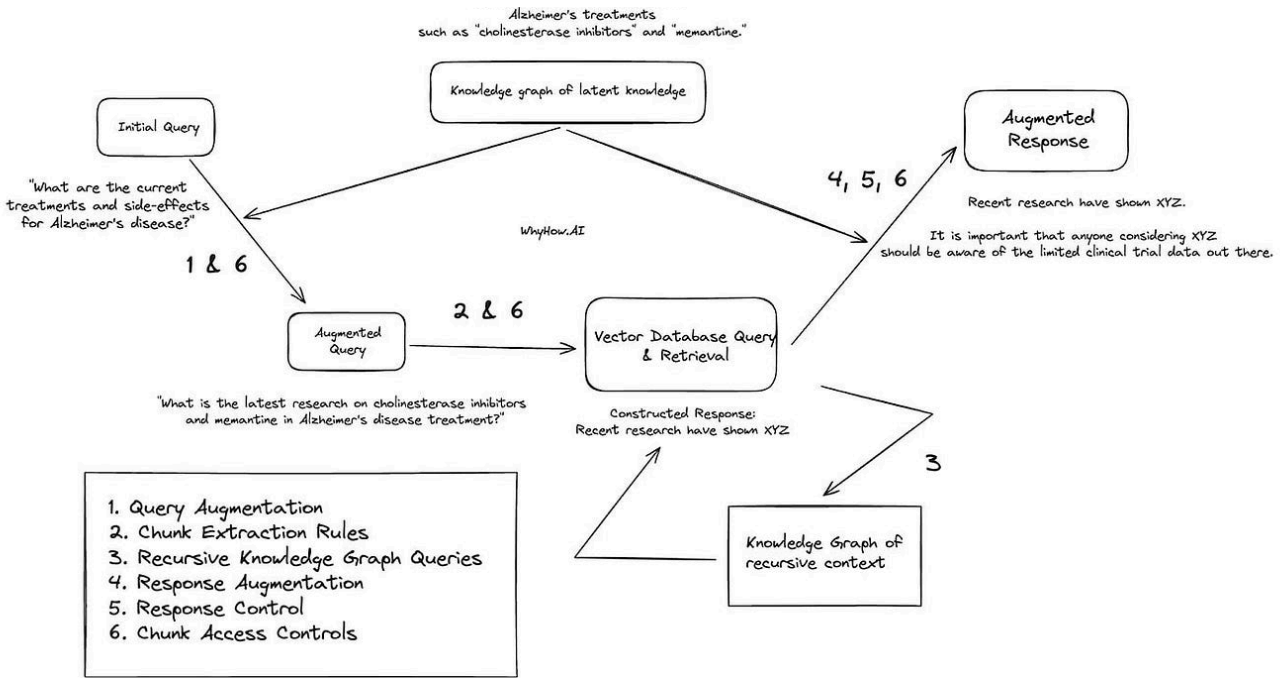
Tomaz Bratanic

Constructing knowledge graphs from text using OpenAI functions

Seamlessy implement information extraction pipeline with LangChain and Neo4j

11 min read · Oct 20, 2023

-- 10



Chia Jeng Yang in WhyHow.AI

Injecting Knowledge Graphs in different RAG stages

Injecting KGs in RAG in pre-processing, post-processing, chunk extraction, document and contextual hierarchies, with a concrete example.

10 min read · Jan 5, 2024

-- 3