Streaming Knowledge Graph Search

1. What is the Purpose of this Project?

The purpose of a **Streaming Knowledge Graph** is to build a system that **"learns" in real-time.**

- **The Problem:** Traditional search (like Google or SQL) retrieves documents based on keywords. It doesn't understand *relationships* or *new events* as they happen.
- **The Solution:** This system ingests a stream of live data (like tweets, financial news, or server logs), uses AI to understand it instantly, and updates a "brain" (the Graph) that connects new facts to old facts.

Real-World Use Cases (The "Business Value"):

1. **Financial Intelligence:** A news stream says *"Company A acquires Company B."* Your graph instantly updates the relationship. If you query "Who owns Company B?", the system knows the answer immediately, unlike a static model trained months ago.
2. **Cybersecurity:** Streaming logs into a graph to detect attack patterns (e.g., IP X accessed Server Y which connects to Database Z) in real-time.

Why it differentiates: It proves you can handle **concurrency** (Kafka), **unstructured data** (LLMs), and **complex data structures** (Graphs) simultaneously.

---

2. Recommended Open Source Stack & Repositories

Since you already know FastAPI, Neo4j, and Python, you should stick to tools that integrate well with them.

The Tech Stack

- **Streaming: Apache Kafka** (Standard) or **Redpanda** (Lighter, easier, C++ based equivalent).
- **Orchestration/LLM: LlamaIndex** (Highly recommended over LangChain for this specific use case because it has built-in "Knowledge Graph Indexing" capabilities).
- **Database: Neo4j** (Graph) + **Elasticsearch** (Vector/Keyword search) OR **Qdrant** (Pure vector search, simpler setup).
- **Extraction: Ollama** (Local LLMs like Llama 3) or **OpenAI API**.

Useful Repositories for Reference

While there isn't one repo that does *exactly* this custom stack, these libraries are your building blocks:

1. **Microsoft GraphRAG:** microsoft/graphrag
   - *Use for:* Understanding how to extract structured graphs from unstructured text. This is the gold standard right now.
2. **LlamaIndex Knowledge Graph Recipes:** run-llama/llama_index
   - *Use for:* Copying their logic on how to turn text chunks into Neo4j nodes.
3. **FastAPI + Kafka Example:** folkrett/fastapi-kafka
   - *Use for:* Setting up the async consumer pattern.

---

3. Step-by-Step Implementation Guide

Here is a roadmap to build this efficiently.

Phase 1: The Infrastructure (Docker)

Don't install these locally. Create a `docker-compose.yml` file to spin up your services.

- **Services needed:** Zookeeper (for Kafka), Kafka, Neo4j, and your Python app.
- *Goal:* Run `docker-compose up` and have a blank Neo4j browser and a running Kafka broker.

Phase 2: The Producer Layer (Hybrid Ingestion)

We now use a dual-producer strategy to feed the `news-stream` Kafka topic.

1. The Batch Simulator (Background)

- **Script**: `app/kafka/producer.py`
- **Role**: Reads historical data from `news_data.csv` to load-test the system.
- **Behavior**: Simulates a steady stream of events (1 event/2 sec).

2. The Real-Time Ingestion API (Interactive)

- **Endpoint**: `POST /ingest`
- **Role**: Allows manual injection of high-priority or breaking news via HTTP requests (e.g., from a frontend or Postman).
- **Behavior**: Instantly pushes the payload to Kafka, triggering the Consumer pipeline immediately.

Phase 3: The Consumer & Entity Extraction (The Core)

This is the hardest part. Create a FastAPI service that listens to `news-stream`.

- **Step A:** Ingest the text message.
- **Step B (LLM):** Send the text to an LLM with a prompt like:
  "Extract entities (Person, Company, Location) and relationships from this text: '{text}'. Return JSON format: [{subject, relation, object}]."
- **Step C:** Parse the JSON response.

Phase 4: Graph Ingestion (The Storage)

- **Task:** Take the parsed triples (Subject -> Predicate -> Object) and write them to Neo4j.
- **Optimization:** Use `MERGE` statements in Cypher so you don't create duplicate nodes (e.g., if "Elon Musk" appears twice, it maps to the same node).

Phase 5: The Search API (The Interface)

- Create a FastAPI endpoint: `/search?q=...`
- **Vector Search:** Convert the user's query to a vector, find relevant nodes.
- **Graph Traversal:** Look at the neighbors of those nodes in Neo4j (2-hop traversal).
- **Synthesis:** Send the graph data to the LLM to generate a natural language answer.

About Project Architecture and Data Flow

Part 1: The Technology Deep Dive

1. Apache Zookeeper (The Coordinator)

- **Technical Role:** It is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and group services.
- **In Your Project:** Zookeeper is the "heartbeat" monitor. It constantly checks if your Kafka broker is alive.
- **Key Concept: Leader Election**. If you had 3 Kafka servers and one died, Zookeeper decides which remaining server takes over.
- **Why it matters:** Without Zookeeper, the Kafka broker cannot start or know its own identity. (Note: Newer versions of Kafka utilize "KRaft" mode to remove Zookeeper, but the standard Docker image still uses it for stability).

2. Apache Kafka (The Event Streaming Platform)

- **Technical Role:** A distributed commit log. It stores streams of records in categories called **topics**.
- **Internal Structure:**
    - **Topic:** Logical bucket (e.g., `news-stream`).
    - **Partition:** To handle huge data, a topic is split into "partitions." (We are using 1 partition for simplicity).
    - **Offset:** A unique ID (integer) assigned to every message. It acts like a bookmark. If your consumer crashes, it remembers "I read up to Offset #50" so it doesn't lose data or read duplicates.
- **In Your Project:** It decouples your "Input" (Producer) from your "Processing" (AI). If your AI takes 5 seconds to process one article, but 100 articles arrive in 1 second, Kafka holds the other 99 safely.

3. Hugging Face Inference API / Local LLM (The Processor)

- **Technical Role:** Performs **Named Entity Recognition (NER)** and **Relation Extraction (RE)**.
- **In Your Project:**
    - It takes unstructured text: *"Elon Musk bought Twitter."*
    - It returns structured JSON: `{"head": "Elon Musk", "type": "BOUGHT", "tail": "Twitter"}`.
- **Why Hugging Face:** It gives you access to models like `Mistral-7B` or `Bert-base-NER` which are specifically trained to understand language structure.

4. Neo4j (The Graph Database)

- **Technical Role:** A native graph database that stores data as **Nodes** (objects), **Relationships** (lines connecting them), and **Properties** (details like name, date).
- **Key Language: Cypher** (SQL for Graphs).
    - Instead of `SELECT * FROM table`, you write `MATCH (n)-[r]->(m) RETURN n, r, m`.
- **In Your Project:** It allows for "Index-Free Adjacency." This means finding friends of friends doesn't require scanning the whole table (like SQL JOINs); it just "hops" from one node to the next, making it millions of times faster for connected data.

5. FastAPI (The Interface)

- **Technical Role:** An asynchronous web framework for building APIs with Python.
- **Key Feature: Async/Await**. This is crucial. It allows your code to send a request to the LLM or Database and *do other work* while waiting for the response, rather than freezing.

Part 2: The End-to-End Data Flow

This is the journey of a single piece of data (e.g., a news headline) through your system.

Step 1: Ingestion (The Producer)

1. **Trigger:** You run `producer.py`.
2. **Action:** The script reads a row from `news_data.csv`.
3. **Serialization:** Python converts the dictionary object `{'text': 'Apple buys AI startup'}` into **bytes** (JSON string).
4. **Transport:** The Producer pushes these bytes to the Kafka Broker on port `9092` into the topic `news-stream`.

Step 2: Buffering (Kafka)

1. **Storage:** Kafka receives the message and appends it to the end of the log for `news-stream`.
2. **Assignment:** It assigns the message an **Offset ID** (e.g., `Offset: 101`).
3. **Waiting:** The message sits there on the disk/memory. It waits. It does not "push" itself to the consumer; the consumer must come and "pull" it.

Step 3: Processing (The Consumer)

1. **Polling:** Your `consumer.py` (running in a `while True` loop) asks Kafka: *"Do you have anything new for me?"*
2. **Fetching:** Kafka says *"Yes, here is Offset 101"* and sends the byte data.
3. **Deserialization:** Python converts the bytes back into a dictionary.
4. **LLM Call:**
    - The Consumer takes the text: *"Apple buys AI startup..."*
    - It sends an HTTP POST request to **Hugging Face API**.
    - **The Wait:** The script waits (e.g., 0.5s - 2s) for the AI to "read" and extract facts.

      Response: Hugging Face returns:
      JSON
      ```
      [
        {"entity": "Apple", "label": "ORG"},
        {"entity": "AI startup", "label": "ORG"},
        {"relation": "ACQUIRES"}
      ]
      ```

Step 4: Persistence (Neo4j)

1. **Transformation:** The Consumer converts that JSON into a Cypher query:
   Cypher
   ```
   MERGE (a:Company {name: "Apple"})
   MERGE (b:Company {name: "AI startup"})
   MERGE (a)-[:ACQUIRES]->(b)
   ```

` **2. Execution:** The Python driver sends this query to Neo4j (Port `7687`) via the **Bolt Protocol**.

3. Commit: Neo4j updates its graph structure on disk.

Step 5: Access (The User)

1. **Query:** A user (or frontend) hits your FastAPI endpoint: `GET /search?q=Apple`.
2. **Search:** FastAPI runs a Cypher query against Neo4j to find the "Apple" node and all its relationships.
3. **Result:** The user sees *"Apple acquired AI startup"*—data that started as raw text and was transformed into knowledge.

About Input of new data

1. Where do I PUT new data? (The Input Source)

Right now, in our "Simulation Phase," you insert data by **editing the CSV file**.

- **Location:** `data/news_data.csv`
- **Action:** You add a new row to this file.
- **Trigger:** When you run `python -m app.kafka.producer`, the script reads that new row and pushes it into the system.

In a Real-World Production App: You wouldn't edit a CSV. Instead, you would create a **FastAPI Endpoint** (e.g., `POST /ingest`).

- **Action:** You send a JSON payload via Postman or a Frontend.
- **Flow:** `User` → `FastAPI` → `Kafka Producer` → `Topic`.