How Data Lineage Stopped Our Analysts' Table Tantrums 🚀

Anurag Sharma 4 min read · Jun 8, 2025

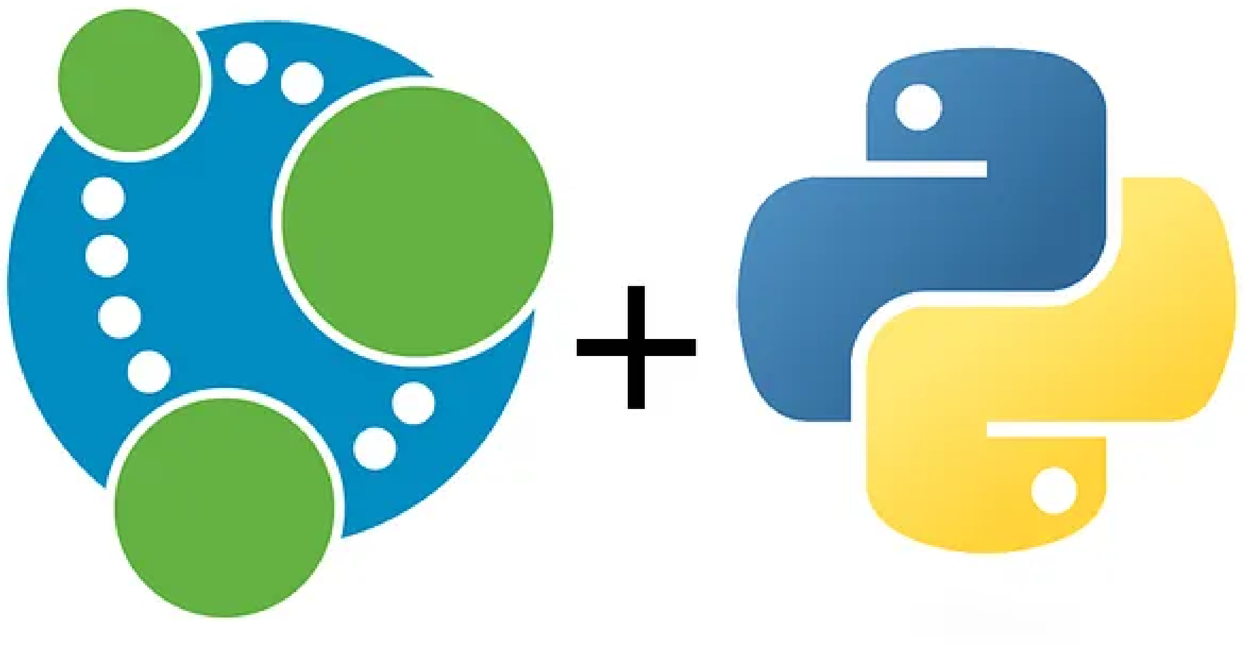
Imagine a world where your data analysts are bombarding you with questions like, *"Where's this column coming from?"* or *"What's the grain of this table?"* 😩 At our data analytics company, we were having 500–600 tables, with new ones popping up like mushrooms after rain. Our data engineers were busy building pipelines, but the lack of clear documentation left our analysts struggling to create BI reports and dashboards.

The solution? A **data lineage framework** powered by a Python crawler and Graph DB like Neo4j that turned chaos into clarity.

The Problem: A Documentation Disaster
With 500–600 tables (and counting), our data ecosystem was a sprawling metropolis of schemas, columns, and relationships. Data engineers like us were focused on building pipelines, but our documentation was more like an old library book with missing pages.

📚 Analysts were constantly raising tickets:
• *"What's the source of this column?"*
• *"Is this table at the daily or hourly grain?"*
• *"Why doesn't this table match the dashboard requirements?"*
This lack of sync between tables and documentation was killing productivity. Analysts spent hours scanning through databases instead of building BI dashboards.

Our goal was clear: build a system to make table relationships and metadata crystal clear, so analysts could focus on insights and dashboards.



The Solution: A Python Crawler + Neo4j = Lineage Magic

We built a **data lineage framework** that became our analysts' new best friend.

Where hate has no platform.
Medium members help preserve a kinder, more thoughtful internet.
Learn more

Here is how we pulled it off:

1️⃣ **Python Crawler: The Metadata Harvester**

We created a **Python crawler** to scrape metadata from the database's information_schema.

The crawler extracted:

• **Table Names:** Every table in our database (all 600+ of them).

• **Column Details:** Names, data types, and constraints (e.g., primary keys, foreign keys).

• **Relationships**: How tables were linked via keys or joins.
• **Metadata Extras**: Creation dates, last updated timestamps, and schema details.
We used Python's sqlalchemy to connect to our database (in our case, a mix of Amazon Redshift and PostgreSQL) and query the information_schema tables. The crawler ran nightly to capture any new tables or schema changes, ensuring our metadata was always fresh.

2️⃣ Neo4j: The Graph Database

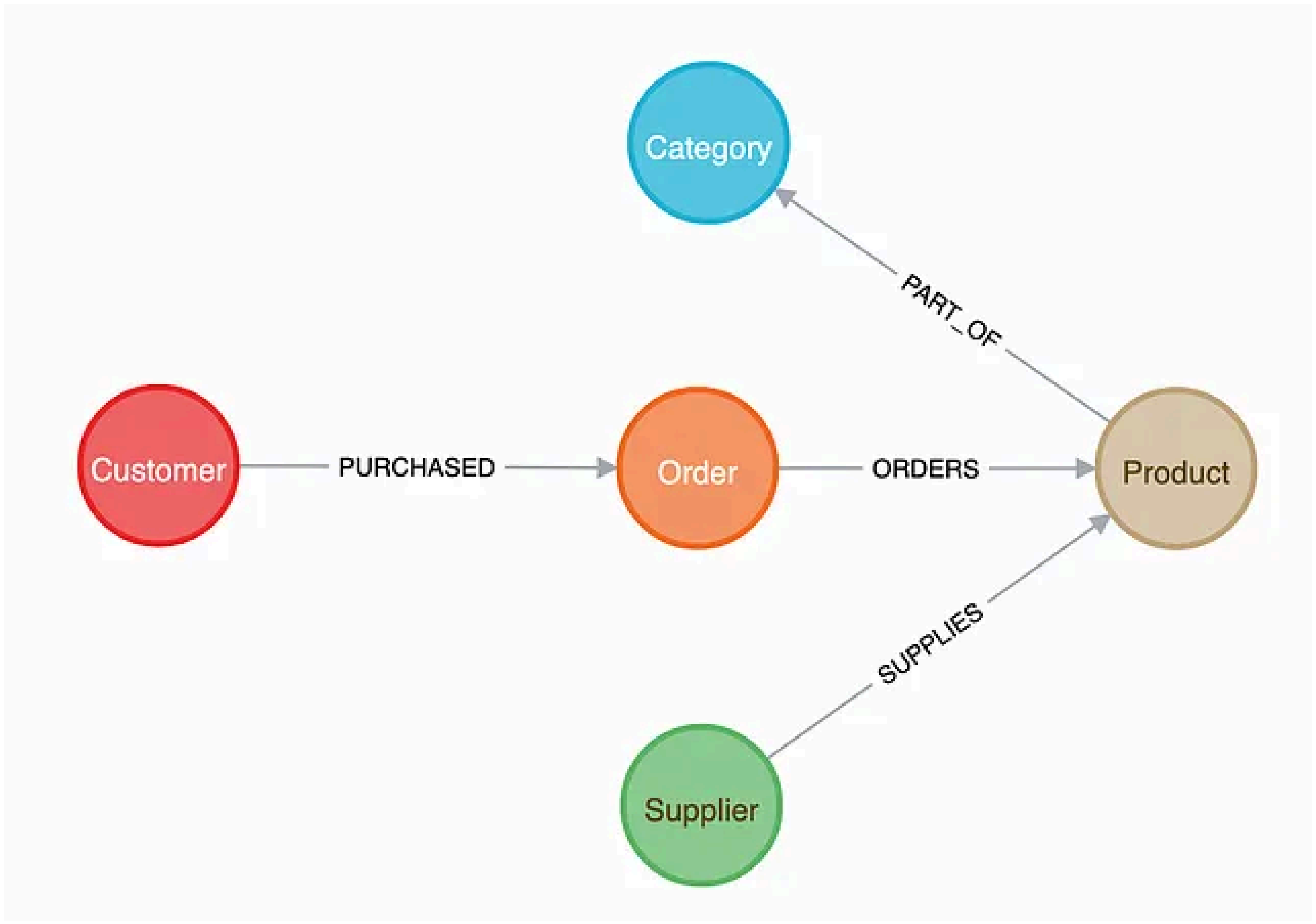
Once we had the metadata, we needed a way to visualize and query table relationships. Enter **Neo4j**, a graph database that's perfect for mapping complex relationships. We pushed our metadata into Neo4j, creating a **top-to-bottom lineage structure**:
• **Nodes**: Represented tables, columns, and schemas.
• **Edges**: Showed relationships like foreign key connections, joins, or data flow (e.g., which table feeds into another via a pipeline).
This graph turned our chaotic table sprawl into a clear, navigable map. Analysts could now trace a column's origin, understand table dependencies, and see the grain (e.g., daily, hourly) in a few clicks.

3️⃣ Analyst-Friendly Interface: No Extra Work Required

To make the lineage accessible, we built a simple **web-based dashboard** using Streamlit (a Python framework for quick UIs).

Analysts could:
• Search for a table or column and see its full lineage (e.g., source tables, transformations, and downstream dependencies).
• Check metadata like grain, data types, and update frequency.
• Visualize the graph of table relationships, with clickable nodes to drill down.

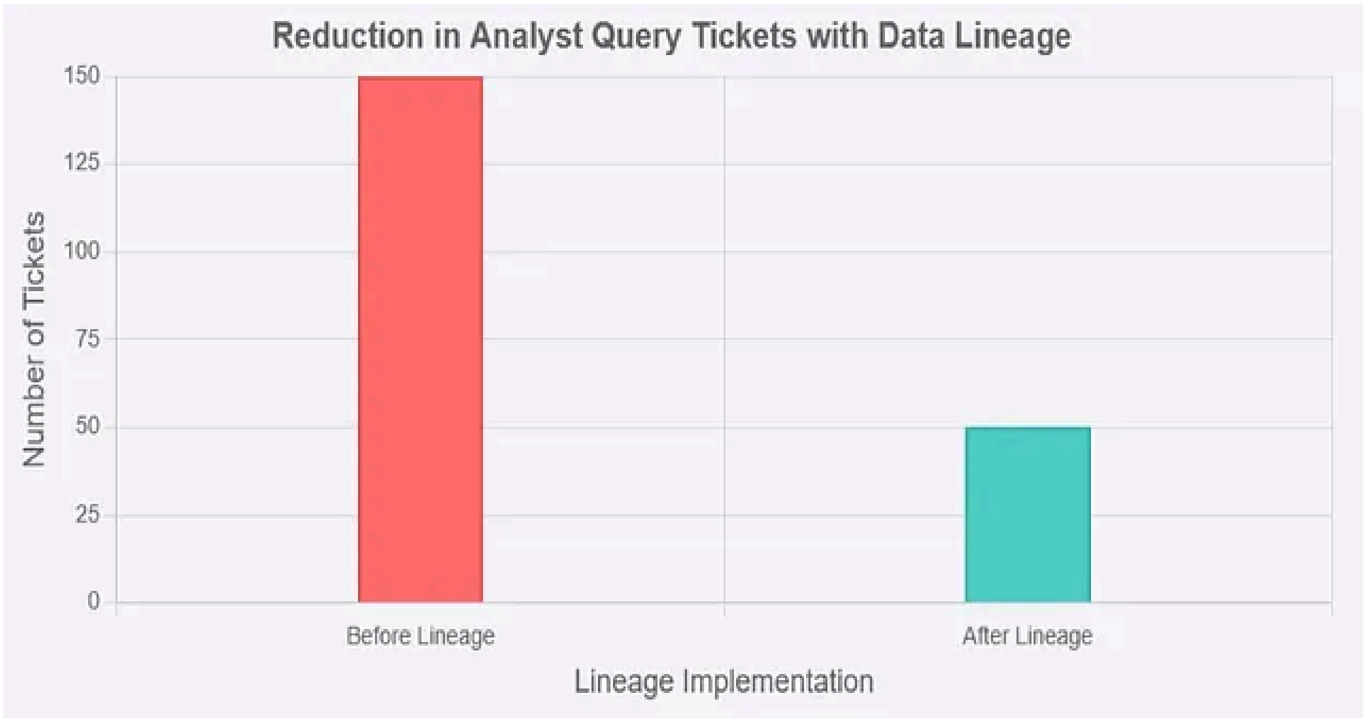
This dashboard was a game-changer. Instead of pinging engineers, analysts could self-serve answers in seconds.



The Impact: Productive Analysts and Engineers

- 📉 **Fewer Queries:** Analyst questions about columns and tables dropped by 67%. They could find answers themselves via the Neo4j dashboard.
- 🚀 **Faster BI Development:** Dashboards and reports were built 2x faster because analysts knew exactly which tables to use and their grain.
- 🛡️ **Better Sync:** Automated metadata updates kept documentation in lockstep with the database, reducing errors in reporting.
- 😊 **Happier Teams:** Engineers focused on pipelines, not answering

repetitive questions, and analysts felt empowered to explore data confidently.



How to Build Your Own Data Lineage Framework

Here is a step-by-step guide:

1. Crawl Your Metadata: Write a Python script using sqlalchemy or psycopg2 to query your database's information_schema. Capture table names, columns, relationships, and metadata.

2. Store in Neo4j: Set up a Neo4j instance (cloud or local) and load your metadata as nodes and edges. Use the py2neo or neo4j Python library to automate this.

3. Automate Updates: Schedule your crawler with Airflow or a cron job to keep the lineage current as new tables are added.

4. Build a UI: Create a simple dashboard with Streamlit or Flask to let analysts query and visualize the lineage.

5. Train Your Team: Show analysts how to use the dashboard and interpret lineage. A 10-minute demo can save hours of back-and-forth.

The Bottom Line

• **Problem:** Poor documentation and unclear table relationships led to constant analyst queries and delayed BI reports.
• **Solution:** A Python crawler to harvest metadata and a Neo4j graph database to map data lineage, paired with an analyst-friendly dashboard.
• **Impact:** 67% fewer queries, faster report-building, and a happier, more productive team.

Python Automation Data Engineering Data Analysis Data Visualization

Written by Anurag Sharma

82 followers · 3 following

Edit profile

Data Engineering Specialist with 10+ exp. Passionate about optimizing pipelines, data lineage, and Spark performance and sharing insights to empower data pros!

No responses yet

Anurag Sharma him/he

What are your thoughts?