neo4j

EBOOK

THE DEVELOPER'S GUIDE:

How to Build a Knowledge Graph

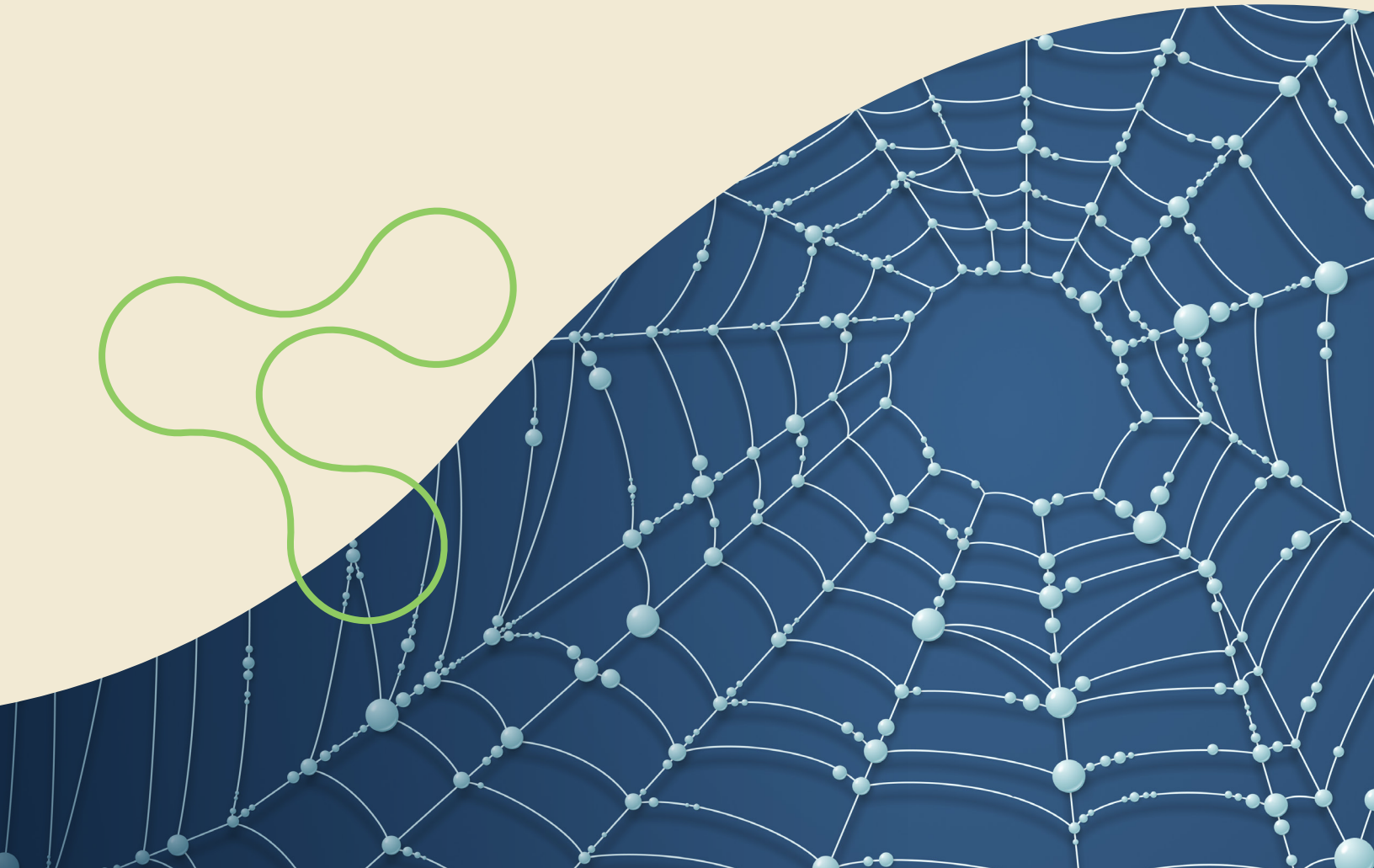

Table of Contents

The Developer's Guide: How to Build a Knowledge Graph

Our minds make sense of data by connecting different pieces of information to form a cohesive picture. Traditional database systems like MySQL and PostgreSQL store data in rigid boxes that don't connect easily. This causes headaches for companies and the developers working with these systems. Examples include:

- Information silos block natural collaboration between teams.
- Complex join operations and foreign keys lead to poor runtime performance.
- Fixed schemas resist adaptation as business needs change.

The biggest problem with traditional database systems is that the intricate context around the data — how everything fits together — often gets lost. It's like trying to understand a complex codebase without any documentation or understanding of the relationships between different modules. Over time, traditional databases become increasingly difficult to maintain and modify, which steadily erodes their business value.

A knowledge graph solves these problems. Rather than holding data static in rows and columns, a knowledge graph organizes information in its natural form: a web of interconnected entities. A flexible schema makes it simple to add new entities and relationships as they emerge. Patterns that get lost easily in a traditional database, such as similar purchasing behaviors or fraudulent transactions, are clearly visible in a knowledge graph.

This guide walks you through everything you need to know to build your first knowledge graph. You'll learn core concepts and how to think about modeling data with relationships. Then, you'll set up your own knowledge graph and start querying it to answer questions that you can't answer in a "traditional database."

What Is a Knowledge Graph?

A knowledge graph maps entities — objects, events, or concepts — and their relationships into an interconnected structure. This relationship-centric approach models real-world scenarios with precision while embedding domain-specific knowledge and business rules as organizing principles.

Used to integrate different types of information, a knowledge graph works well for use cases that pull data from multiple sources, including structured (traditional database entries) and unstructured (documents, social media posts) data. This unified view of a company's knowledge is highly valuable, especially compared to traditional data design where information is fragmented, and relationships must be reconstructed through JOIN queries.

For example, a knowledge graph might represent patients, symptoms, diseases, etc., as depicted in the diagram below. A patient might have symptoms similar to patients with overlapping symptoms, diseases, or side effects. Prescriptions could also be linked to the pharmaceutical companies that make them or the doctors who often prescribe them for several different illnesses.

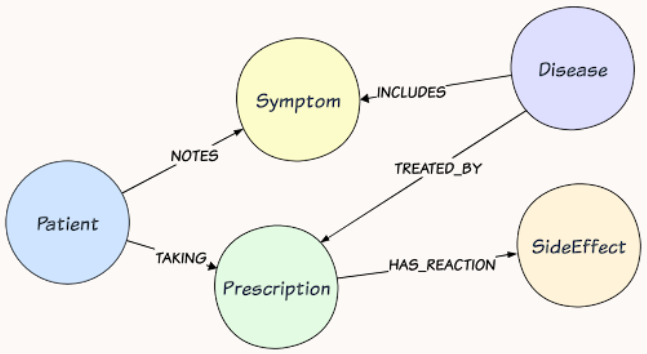

Figure 1. Knowledge graph example

Knowledge graphs surface hidden patterns through connections in data. For instance, a medication manufacturer depends on a supplier network for product components. A knowledge graph could reveal that several key suppliers are located in a hurricane-prone region — a risk that might go unnoticed in a traditional database.

Components of a Knowledge Graph

There are three major components of any knowledge graph: nodes, relationships, and organizing principles.

Nodes represent instances of specific entities, such as tangible objects (people, places, things), abstract concepts, or events. Nodes are the fundamental building blocks of a knowledge graph. You can have as many nodes as needed in a graph.

In the healthcare example, nodes represent individual patients and diseases:

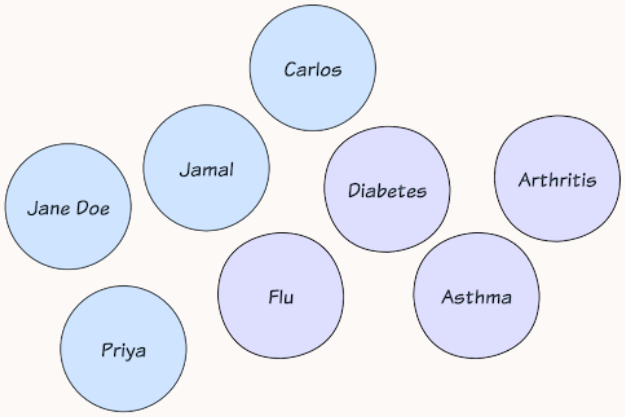

Figure 2. Healthcare example — nodes

Labels identify nodes by role or type, serving as a classifier or tag that defines their function or purpose in your domain. They add semantic meaning to nodes, making the graph more intuitive to understand and query. When you specify a label in your query, it helps the graph database find the type of node you're looking for.

Two labels from the healthcare example would be "Patient" and "Disease":

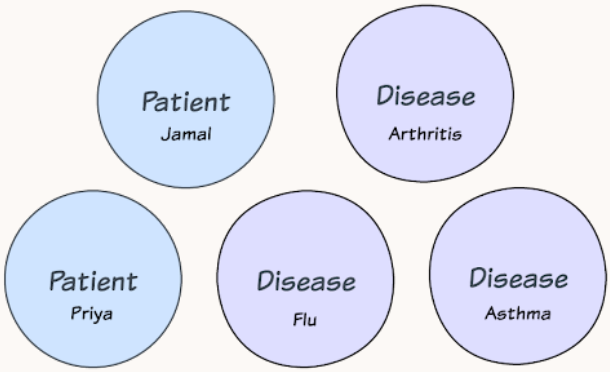

Figure 3. Healthcare example — labels

Relationships, also known as connections, contain information about how nodes interact with or relate to one another. They add context and meaning to the data — a patient linked to a health condition with a "DIAGNOSED_WITH" relationship or connected to a prescription to show what medications they are "TAKING," for instance:

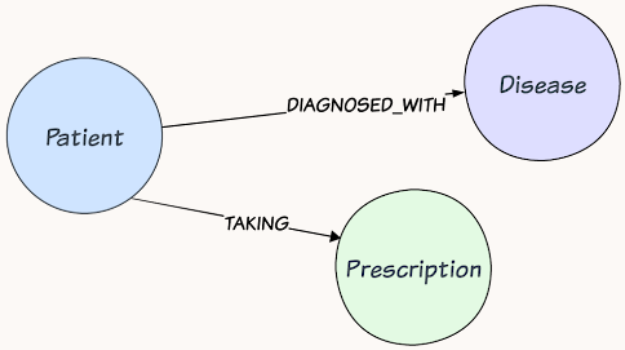

Figure 4. Healthcare example — relationships

Properties are attributes that provide information about nodes and relationships. They enrich the graph with detailed metadata and domain context.

In the healthcare example, the "Patient" node could have properties like name, date of birth, and contact information, while the "Disease" node could include properties like name and description:

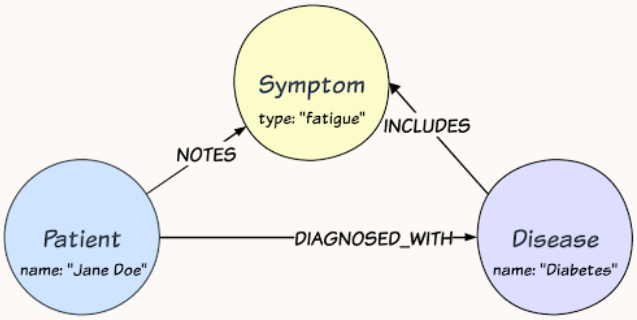

Figure 5. Healthcare example — properties

Organizing principles bring business context to the graph by defining how entities, relationships, and properties are structured and used. They specify the types of nodes and relationships, establish hierarchies or categories, and guide interactions within the graph. This structure makes the data

easier to understand and enables more efficient querying, analysis, and inference across different levels of detail.

In the healthcare knowledge graph, diseases could be organized into categories (such as cardiovascular or respiratory diseases), while patients could be grouped by risk factors or age ranges. This structure enables analysis at various levels, from individual patient-disease relationships to broader population health trends.

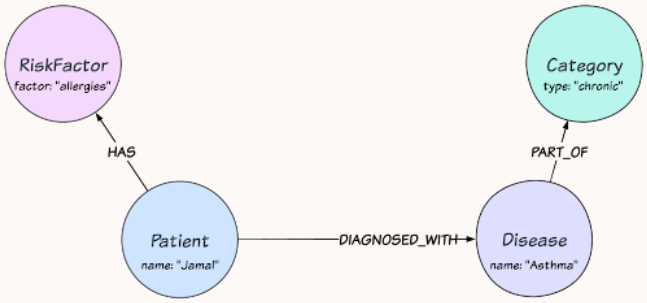

Figure 6. Healthcare example — organizing principles

Build and Query Your Knowledge Graph

Now that you know the fundamentals, you can create your first knowledge graph using a Neo4j graph database. Though you *could* create a knowledge graph in another type of database, a property graph database like Neo4j is purpose-built for this task. A property graph database aligns naturally with the structure of a knowledge graph, making it the most intuitive option for implementation.

This guide teaches you to build a knowledge graph from start to finish. The example is retail transaction data with products, product categories, customers, and orders. You'll learn how to design a knowledge graph, populate it with data, and query it using Cypher. As you move through this process, you'll see how a knowledge graph allows you to answer multi-step questions — sometimes even answering two questions with a single streamlined query.

Once you feel confident querying your knowledge graph, you'll have a chance to experiment with another layer: unstructured data. This is where the magic starts to happen in a knowledge graph: the ability to add new and different types of data and then query relationships across all the data.

Sign Up for a Neo4j Account

You'll build your knowledge graph on the cloud-hosted, fully managed Neo4j AuraDB Graph Database. Neo4j stores data as nodes and relationships, supports the Cypher graph query language, and offers tools for data visualization, data science, and data connectors. Before using AuraDB, you'll need an account.

> *If you already have a Neo4j account, you can log into the Aura Console and skip to the next step to create a database instance.*

Follow these steps to create a Neo4j Aura account:

1. Navigate to the Neo4j Aura Console.

2. Click on **Sign up** below the login box.

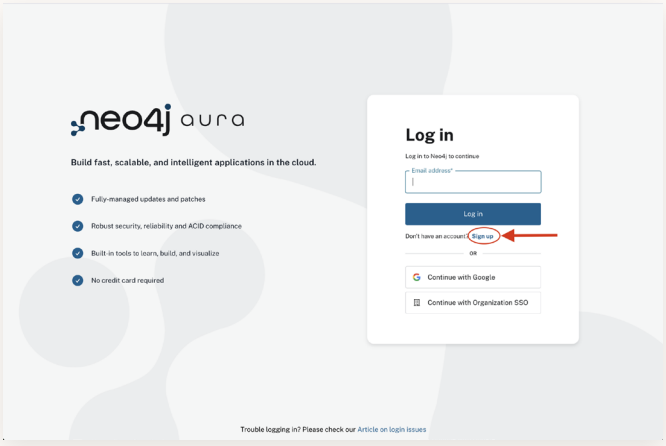

Figure 7. Neo4j Aura signup screen

3. Type your email address into the input box and click **Continue** to set up the password and other necessary information. Alternatively, you can sign in using the Google or organization account option.

4. If prompted, agree to the Neo4j terms.

Next, you'll create a graph database instance to hold your knowledge graph.

Create a Graph Database Instance

In this step, you'll create the actual database instance to store your knowledge graph. If you

haven't already, navigate to the Aura Console and log in. Then:

1. Click the **Create instance** button:

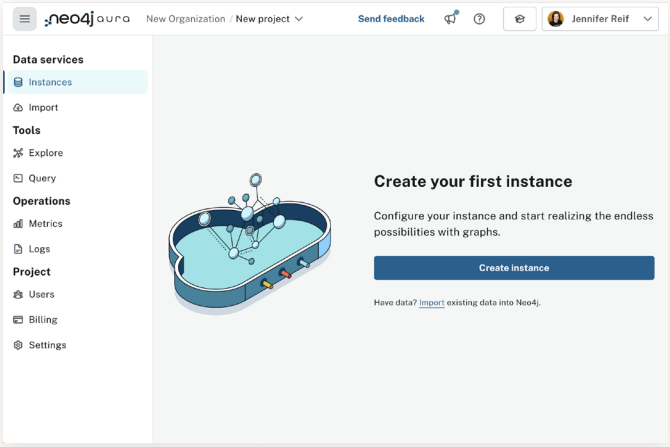

Figure 8. Neo4j Aura Create instance screen

2. You'll see a list of instance types, with the Professional tier (center option) highlighted by default. AuraDB Free is a great way to start learning and exploring knowledge graphs. When you're ready to move to production-quality, high-performance applications in the cloud, you can progress to AuraDB Professional. We'll use the Free instance for our knowledge graph, so click the **Select** button at the bottom of the Free tier.

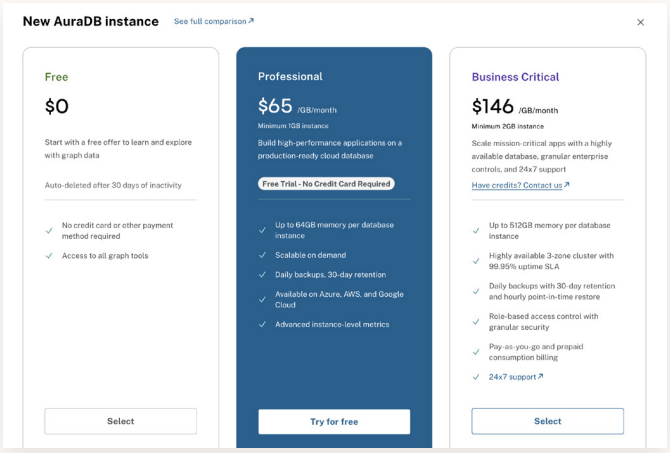

Figure 9. New AuraDB Free instance screen

3. A pop-up should appear with the credentials for your instance. Click **Download and Continue** to download the credentials file. (Important: You cannot access the password after this point.) Your database instance will take a few minutes to create. Once complete, you can move to the next section, where you'll design a graph data model for importing data.

Create a Graph Data Model

Now that you have a database instance ready, you need to populate it with data. The Data Importer tool will help you design the structure of your knowledge graph by drawing entities and relationships to represent your domain of interest.

From the main Aura console, click the **Import** option in the left menu:

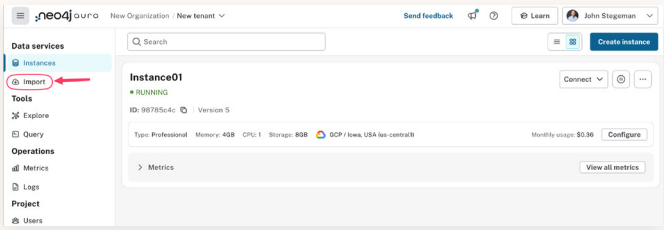

Figure 10. Data import screen

Click **New data source** in the middle:

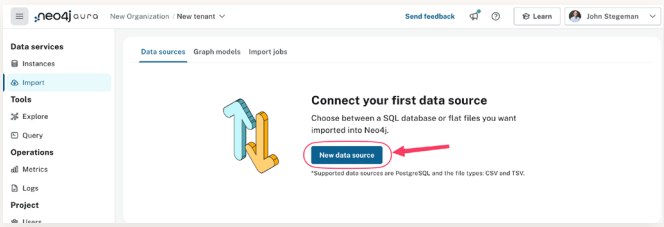

Figure 11. Selecting New data source screen

Then choose the .CSV* option. (lower part of the pop-up):

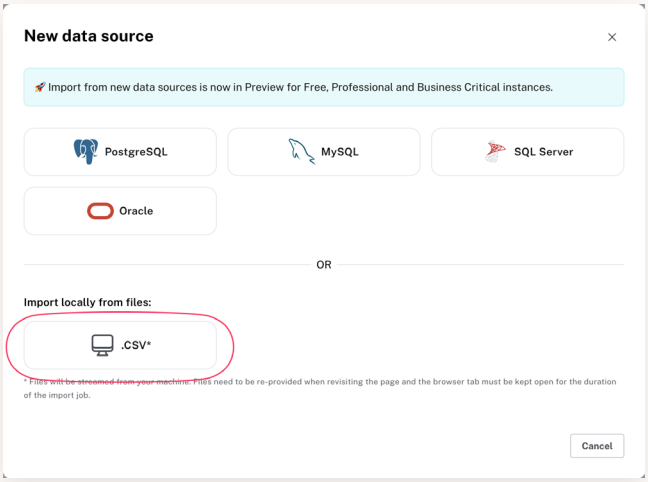

Figure 12. Aura New data source screen

Data Importer may not automatically connect to your running instance, so in the upper left, if it says "No instance connected," follow these steps:

1. Click on the drop-down next to **No instance connected** and click **Connect to instance**:

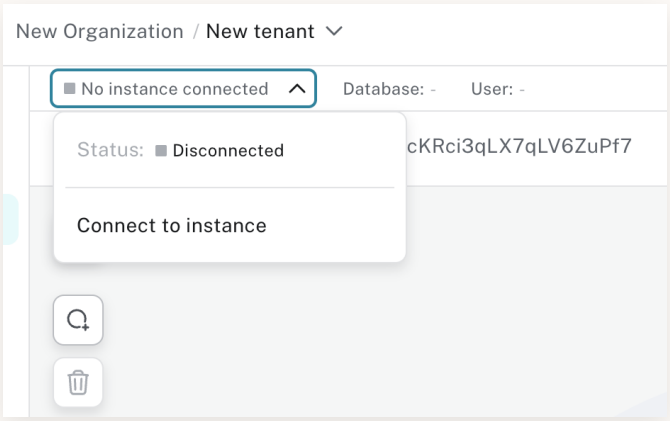

Figure 13. Aura No instance connected screen

2. Click **Connect** next to your instance:

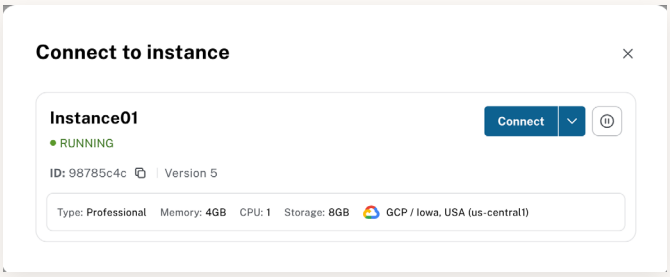

Figure 14. Aura Connect to instance screen

3. In the credentials pop-up, type in the username and password for your instance. The downloaded credential file from earlier is helpful here. Click **Connect**.

You should be on the main Data Importer screen and see your connected instance in the upper left:

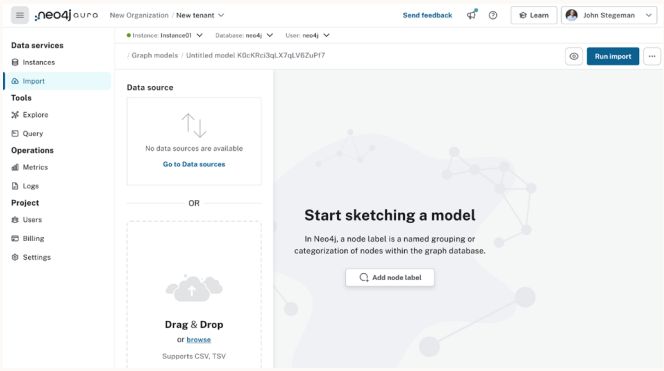

Figure 15. Aura Data Importer connected instance screen

You'll create a simplified version of the Northwind Graph example, an ecommerce demonstration based on the popular Northwind sample dataset. The dataset is formatted as CSV files containing information about customers, orders, products, categories, and suppliers. Let's create the model below in Data Importer:

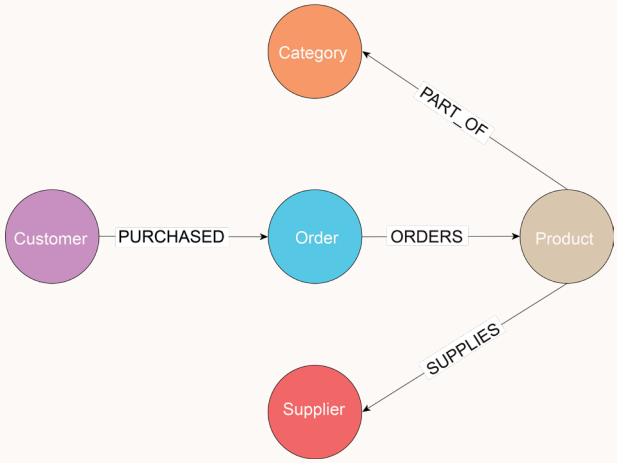

Figure 16. Northwind graph data model

To start designing the model, click **Add node label**:

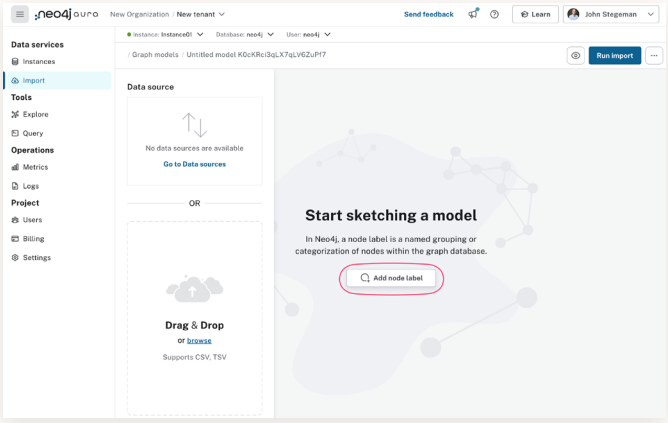

Figure 17. Initial add node label screen

Note: To minimize the Data source tab along the left side, click the **Data sources** icon in the upper left of the visualization pane:

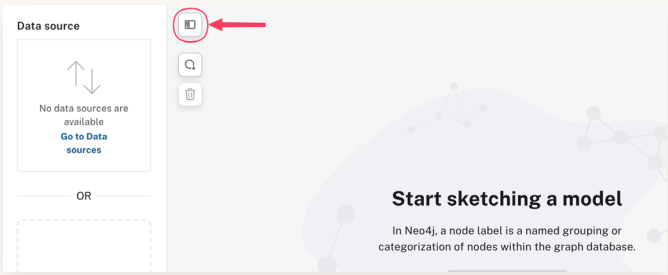

Figure 18. Data sources icon screen

A circle should appear in the pane, along with a right tab containing definition metadata:

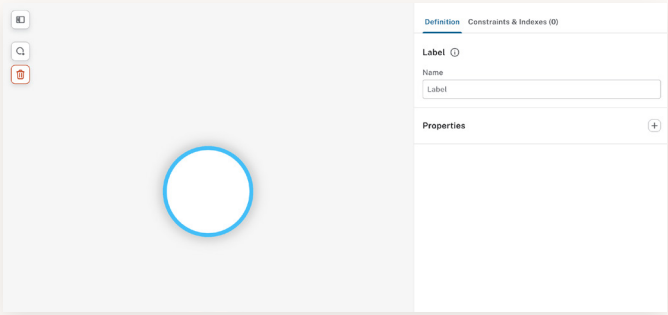

Figure 19. Definition screen

This will be the Customer node in the data model. Customer nodes will have three properties: `customerID` (string), `companyName` (string), and `city` (string). Follow these steps to define the node:

1. Type `Customer` as the label in the **Name** field. This label will identify the type of entity these nodes represent in the graph.

2. Click the **+** sign next to **Properties** to add node properties.

3. Edit the property by clicking the **Property1** button under **Properties**, type `customerID`, and press **Enter**. To the right of the property name, select the appropriate data type from the drop-down (in this case, **string**).

Repeat this process for `companyName` and `city`. Your completed Customer node should look like this:

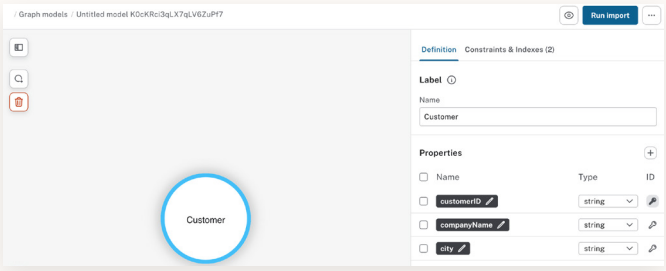

Figure 20. Customer node screen

Next, create another node type for Orders. Click the **Add node label** icon in the top-left corner of the sketch area:

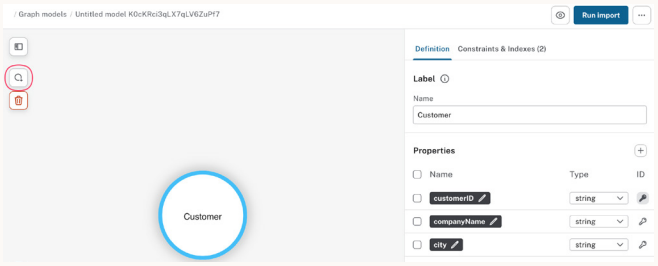

Figure 21. Add node label screen

A new blank node will appear in your workspace. Label this new node type as "Order" and add the following properties: `orderID` (integer), `orderDate` (datetime), and `shippedDate` (datetime):

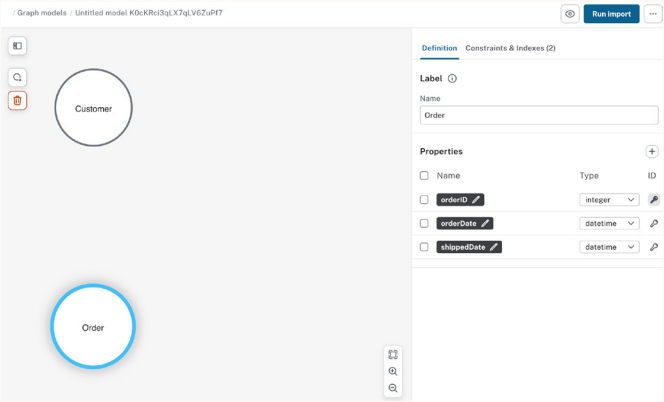

Figure 22. Order node label screen

Next, you'll create a relationship between the "Customer" and "Order" node types to represent that a customer places an order. Click the **Customer** node to select it. Hover your mouse over the border of the node to see a green **+** button. Click and hold the mouse button, then drag the gray circle that appears to the "Order" node:

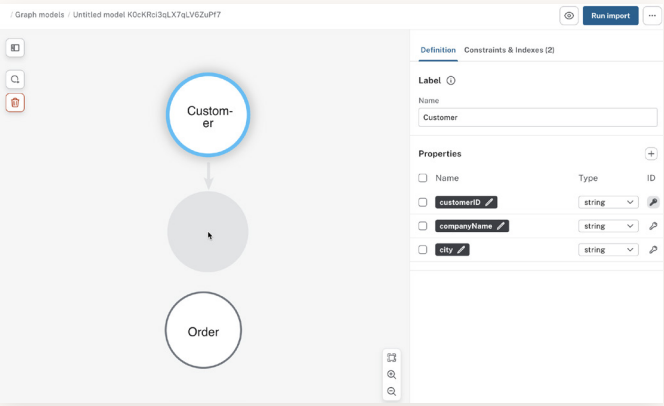

Figure 23. Create the relationship between "Customer" and "Order" screen

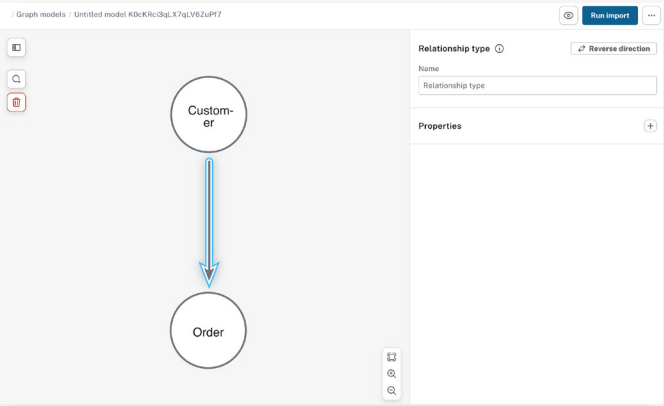

Figure 23 (cont). Create the relationship between "Customer" and "Order" screen

Release the button *only* when you're over the "Order" node to create a new relationship (or else you'll create a new blank node).

Notice that the relationship has a direction from "Customer" to "Order." By drawing a line between the "Customer" and "Order" nodes, you're modeling a customer's purchase in the knowledge graph. The relationship explicitly defines how customers relate to orders.

Relationships need to have a type. In the relationship's metadata pane on the right, name this relationship type "PURCHASED."

This intuitive representation of data and relationships enables powerful querying capabilities because the model clearly defines not only that there is a relationship but also how the entities relate to one another. A purchase relationship would help us understand customer order history and habits, but a Customer-CREATES > Order could point to a shopping cart that hasn't been purchased yet. We'll discuss more ways to answer business questions with graph data later in this guide.

This intuitive representation is how we tend to model and think of data, even for other types of databases. The difference with a knowledge graph is that what you draw is exactly what you're going to store and query in the database. With other database types, you'd have to take this intuitive model and figure out how to implement it within the technical limitations of that database (the transition the conceptual data model from the physical data model). In a knowledge graph, the conceptual data model and physical data model are one and the same.

To finish the Northwind graph model, create a third node type called "Product" and add three properties to it: `productID` (integer), `productName` (string), and `unitPrice` (float):

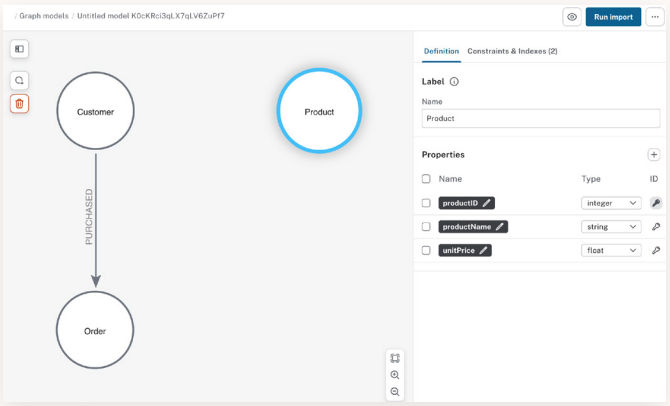

Figure 25. Create "Product" node screen

Next, create a new relationship type by drawing a line from the "Order" node to the "Product" node. Name this new relationship type "ORDERS":

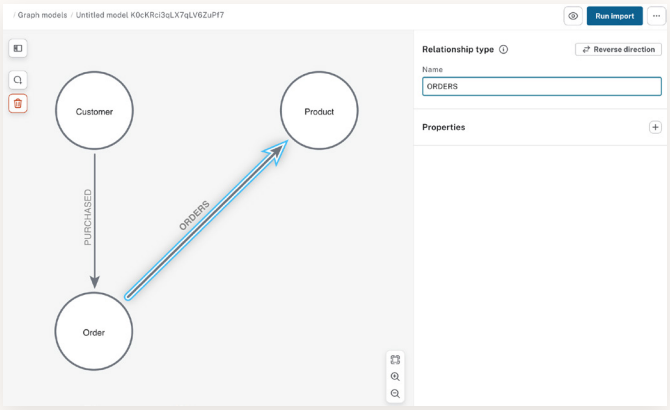

Figure 26. Create a new relationship type by drawing a line from "Order" node to "Product" node screen

Remember that relationships can have properties, too. Add the property `quantity` (integer) to store the number of that product ordered.

To add suppliers to our model, create a fourth node type called "Supplier" and add three properties to it: `supplierID` (integer), `companyName` (string), and `city` (string). Create a relationship from "Supplier" to "Product" and name it "SUPPLIES":

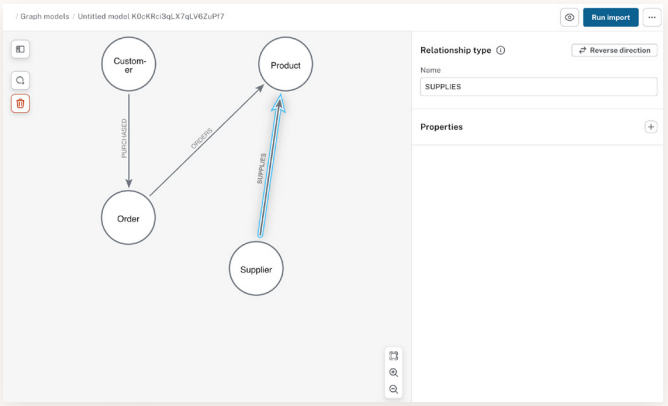

Figure 27. Create "Supplier" node type screen

Your knowledge graph model now has four node types and three relationship types, but it still lacks an organizing principle. In the Northwind example, your organizing principle could be a product hierarchy that streamlines product group searches. As another option, you could choose a process-based principle around the order fulfillment stages to optimize the supply chain and delivery network.

For this example, you'll add a product hierarchy as the organizing principle of your graph.

Start by adding another node type in the data model. Name it "Category" and include the properties `categoryID` (integer) and `categoryName` (string):

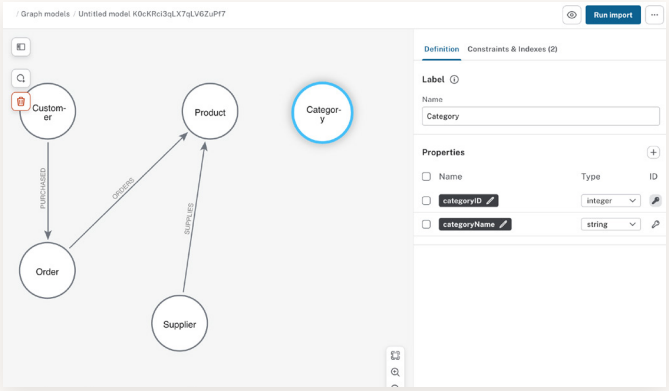

Figure 28. Create "Category" node type screen

Next, create a new relationship type starting from the "Product" node and going to the "Category" node named "PART_OF":

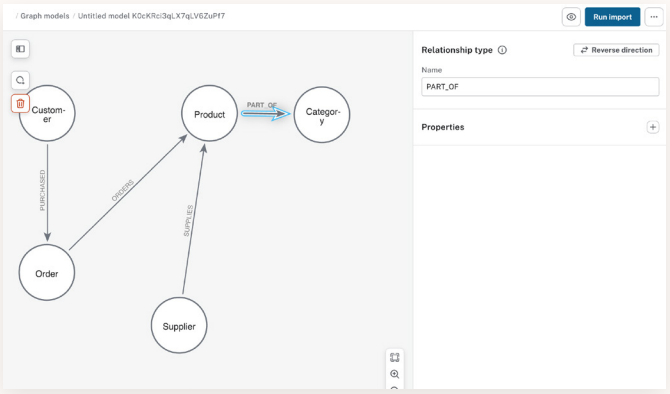

Figure 29. Create "PART_OF" relationship type screen

Now that your knowledge graph model has relevant relationships and an organizing principle, you're ready to bring it to life with data.

Load Data Into Your Knowledge Graph

Loading data into your knowledge graph creates nodes and relationships about specific customers, orders, products, and categories with the model you defined.

For this example, you'll import CSV data from the Northwind database.

Load Structured Data

Download the following CSV files from the Northwind GitHub repository:

- categories.csv
- customers.csv
- order-details.csv
- orders.csv
- products.csv
- suppliers.csv

In the Aura workspace, click the **Data sources** icon in the top-left corner of the sketch area to expand the Files menu:

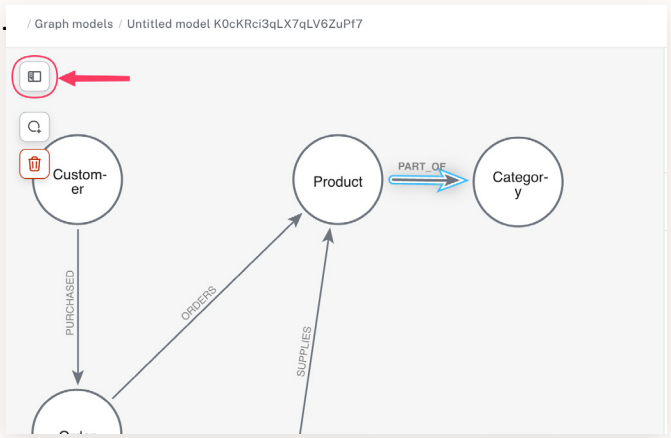

Figure 30. Data sources screen

Choose the bottom of the two options (Drag & Drop and browse support CSV) for loading data into your knowledge graph and add the files you just downloaded.

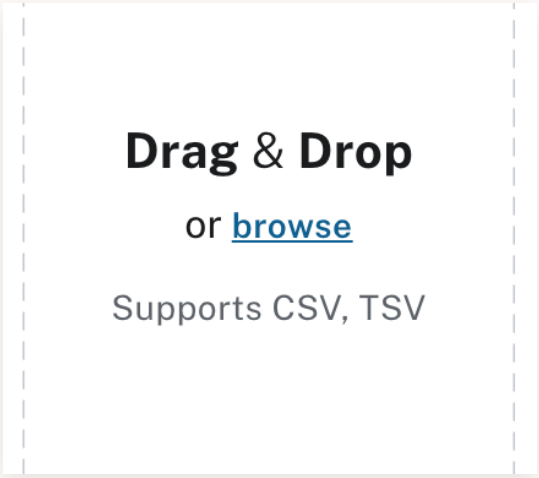

Figure 31. Drag & Drop and browse support CSV selection screen

Here's what you should see after uploading the files (properties collapsed):

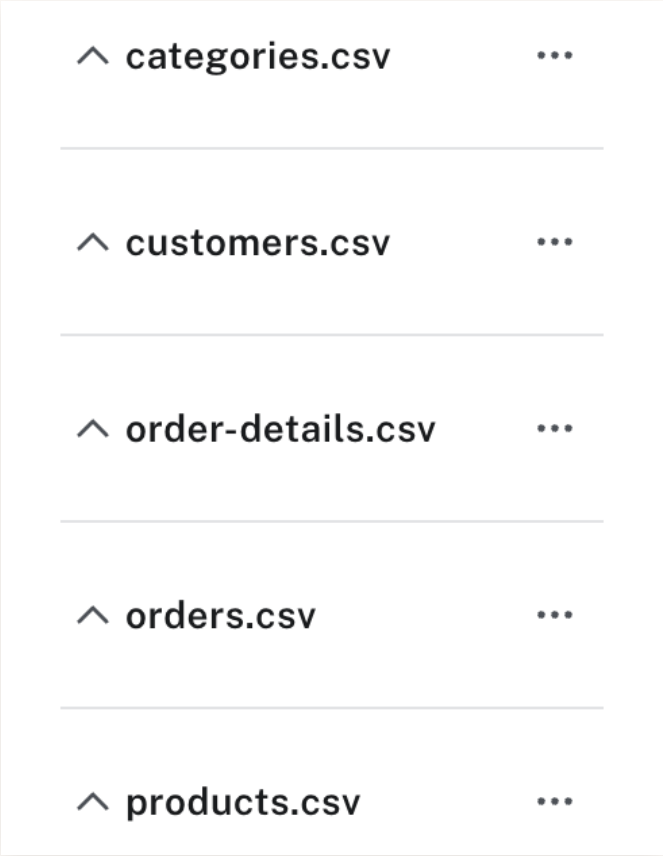

Figure 32. File upload screen

After adding your data files, close the Data source menu by clicking the icon again.

Map the data from the files to the nodes and relationships in your graph model by clicking any node or relationship in the drawing and locating the **Table > Name** field in the **Definition** tab. Open the drop-down list and select the appropriate file:

- For the "Customer" node, select `customers.csv`
- For the "Order" node, choose `orders.csv`
- For the "Product" node, pick `products.csv`
- For the "Supplier" node, select `suppliers.csv`
- For the "Category" node, use `categories.csv`
- For the "PURCHASED" relationship, select `orders.csv`
- For the "ORDERS" relationship, pick `order-details.csv`
- For the "SUPPLIES" relationship, pick `products.csv`
- For the "PART_OF" relationship, choose `products.csv`

Next, scroll down to the **Properties** section. It shows a list of the properties you defined earlier in your graph model. Since you used the same naming convention as the GitHub repository, the property names in your graph model will match the field names in the CSV files, which simplifies the mapping process. Though you can map each property from your model to the CSV field manually using the drop-downs, a simpler option is to click **Map from table** just above the properties, choose which columns from the CSV files to map, and click **Confirm**:

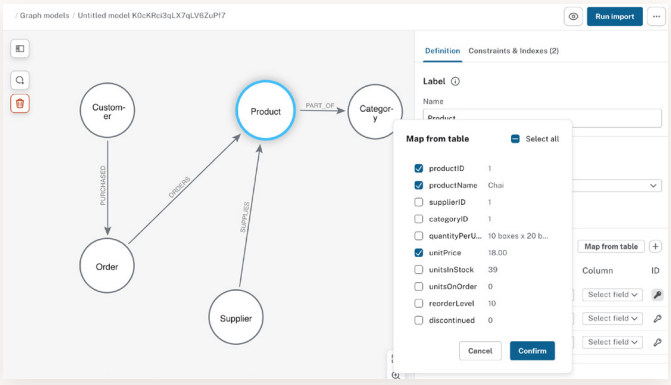

Figure 33. Map from table screen

To map relationships, you'll notice an additional section, **Node ID mapping**, with fields to map **From** and **To** nodes:

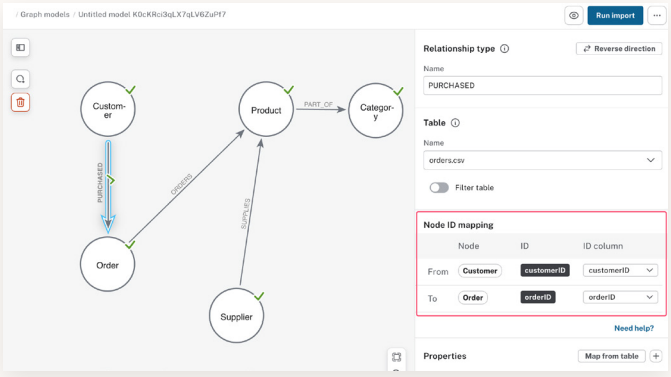

Figure 34. Node ID mapping screen

Complete mappings for each of your graph model's nodes and relationships in any order you prefer. As you map, Aura Workspace places a green checkmark next to each fully mapped element, indicating that all fields for that node or relationship have been successfully populated:

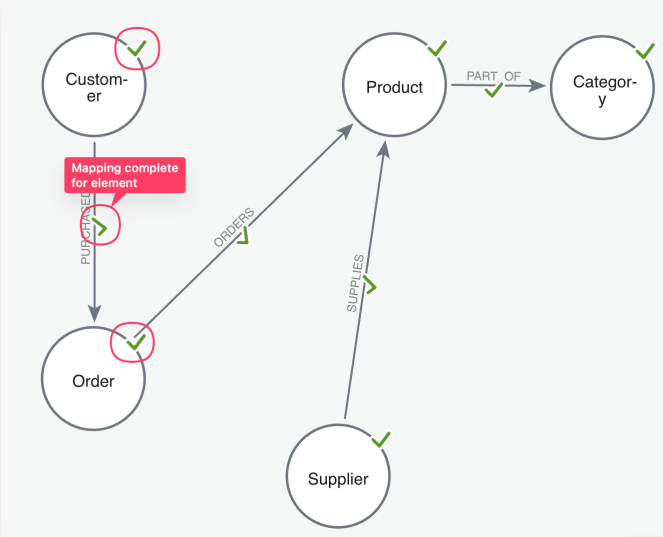

Figure 35. Complete mappings screen

After completing the mapping process for all elements of your knowledge graph, you're ready to populate the database.

Click the **Run import** button to load your data:

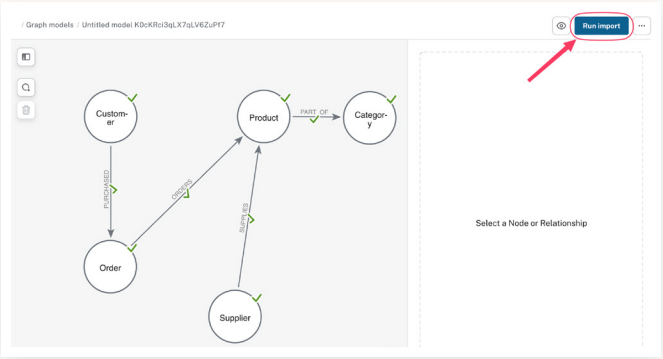

Figure 36. Run import screen

This action starts the import process. You'll see a progress bar indicating the status of the import. Once complete, a pop-up window will display the import results. The window provides a quick overview of the import process outcome and lets you verify whether the data was successfully imported into your knowledge graph.

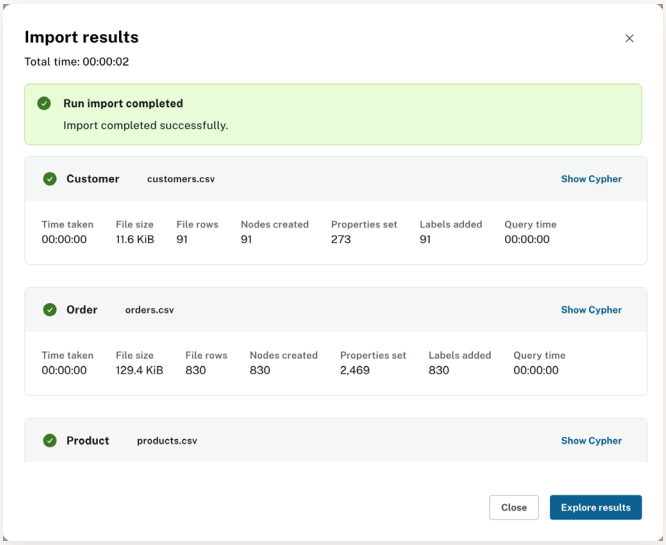

Figure 37. Import results screen

Click the **X** to close the pop-up window.

Now that the data is imported into the database, you can use queries to understand behaviors and patterns in the data.

Query Your Knowledge Graph

You will query your knowledge graph using the Cypher query language. Cypher is the most widely adopted implementation of the ISO Graph Query Language (GQL) standard designed specifically for graph databases. Cypher is a declarative language (like SQL), which means you write queries by specifying the results you want and not dictating how to get them. It offers several advantages over traditional query languages like SQL or SPARQL, including reduced code complexity, easier debugging, and intuitive representation of data patterns.

Cypher expresses graph patterns in a way that resembles how they're drawn on a whiteboard. For instance, a statement like "customer orders product" can be represented in Cypher as:

```
(c:Customer)-[r:ORDERS]->(p:Product)
```

Figure 38. Cypher "customer orders product" diagram

The next sections cover the three most important Cypher clauses you'll need to write queries and interact with your knowledge graph:

1. MATCH finds and returns the nodes or patterns specified.

2. CREATE adds new nodes or patterns specified to the graph.

3. MERGE executes a find-or-create operation, first checking if the pattern exists in the graph (MATCH), then either returns the existing pattern or creates the pattern if it doesn't exist.

> *For a more comprehensive understanding of Cypher, the Cypher Fundamentals course on Neo4j GraphAcademy offers an in-depth overview of Cypher and provides hands-on exercises to reinforce your learning.*

MATCH Clause

Cypher's MATCH clause finds nodes or patterns in a graph database. Use MATCH when you want to find nodes and relationships in your graph, an essential part of data retrieval and analysis. It serves a similar purpose to the SELECT statement in SQL, allowing you to retrieve data based on specified criteria.

Explore your Northwind knowledge graph with the queries in the next paragraphs.

Navigate to **Tools > Query** in the left menu and enter the following Cypher code in the query box at the top right:

```
MATCH (p:Product)-[rel:PART_OF]-
  >(c:Category {categoryName: "Bever-
  ages"})
RETURN p, rel, c;
```

This query finds all products that belong to the beverage category. It's a common type of query in ecommerce systems where you want to look up products in a specific category. Let's break down the query:

- `MATCH (p:Product)-[rel:PART_OF]-
  >(c:Category {categoryName:
  "Beverages"})` matches "Product" nodes (mapped to variable `p`) that have a "PART_OF" relationship to a "Category" node (variable `c`). The "Category" node is filtered to only match where `categoryName` is "Beverages."
- `RETURN p, rel, c` specifies the data to be returned from the matched pattern - products (p), PART_OF relationships (rel), and categories (c):

Figure 39. Cypher `MATCH` beverages query

To execute the query, click the **Play** icon next to the query box.

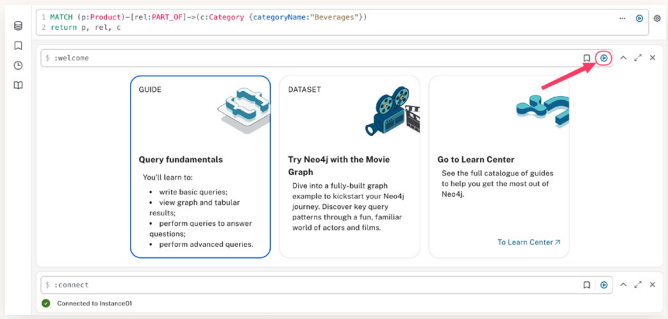

Figure 40. Executing the query screen

Here's the sample output:

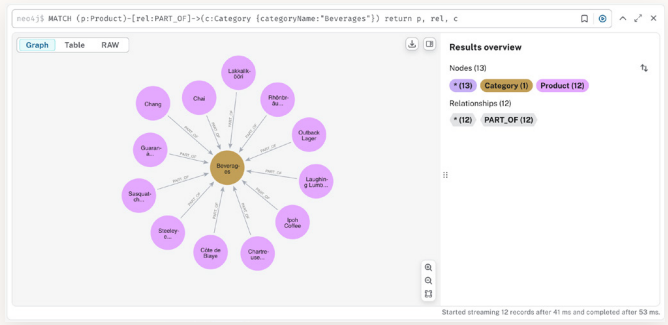

Figure 41. Sample output screen

The query above displays a visualization of nodes and relationships because we returned entire nodes and relationships, but you can specify parts of the pattern and workspace could display text, tables, or other formats.

As an example, the "Ipoh Coffee" product has run out of stock, and you need to identify which orders need to be updated and customers contacted:

```
MATCH (c:Customer)-[r1:PURCHASED]-
  >(o:Order)-[r2:ORDERS]->(p:Product
  {productName: "Ipoh Coffee"})
RETURN c.companyName, COUNT(o) AS or-
  ders, collect(o.orderID)
ORDER BY orders DESC;
```

First, the query searches for customers who purchased orders that contain the "Ipoh Coffee" product. The next line returns the customer's company name, the count of orders impacted, and the affected order IDs in a list. The last line orders the results by the number of orders, sorting from highest to lowest (`DESC`, for descending order).

The output is as follows:

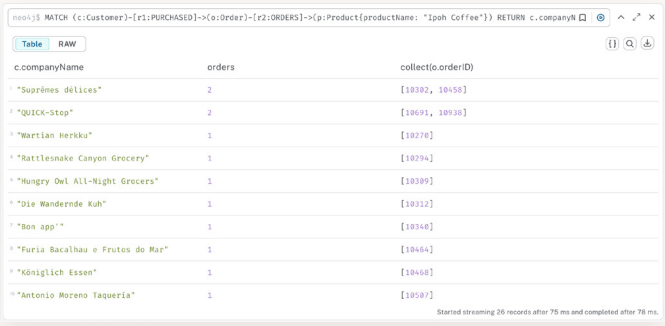

Figure 42. Out-of-stock product impacts

Next, produce might be having weather that creates a higher or lower average crop. To see how that might affect your suppliers, customers, and inventory, you could run a query like the following:

```
MATCH (cust:Customer)-[r1:PURCHASED]-
  >(o:Order)-[r2:ORDERS]->(p:Product)-
  [r3:PART_OF]->(c:Category {category-
  Name:"Produce"}),
  (p)<-[r4:SUPPLIES]-(s:Supplier)
RETURN *;
```

This query finds customers who purchased orders containing products that are part of the "Produce" category, as well as the product suppliers, and returns all the data.

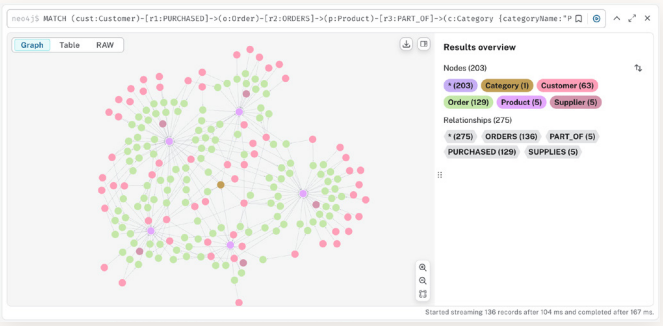

Figure 43. Produce graph network

The results show us that there are a few products in the "Produce" category (in purple), and the related suppliers (in blue) are connected to a single product. This tells us that while we don't rely on one supplier for multiple different products, we also do not have backup suppliers if the existing (and only) supplier of a product is impacted in some way.

So far, we've focused on how to query data that was previously loaded. But what if you want to create new data? You can use Cypher's `MERGE` and `CREATE` clauses to add new data to the graph.

CREATE and MERGE Clauses

The `CREATE` clause adds new nodes, relationships, and properties to the graph. It always creates new data, even if identical data already exists. It's similar to the `INSERT` statement in SQL.

The `MERGE` clause combines the functionality of `MATCH` and `CREATE`. It first attempts to find the specified pattern in the graph. If the pattern exists, it behaves like `MATCH` and returns the existing data. If the pattern doesn't exist, it behaves like `CREATE` and saves the pattern.

When using these clauses, it's important to understand that Cypher operates on entire patterns rather than individual elements. When you `MATCH` or `MERGE`, Cypher looks for the complete pattern specified. For example, if you `MERGE` a pattern and the nodes exist but the relationship does not, Cypher will create the entire pattern new, producing duplicate nodes.

To prevent data duplication, match and then merge individual parts of the pattern separately to ensure

that only the new elements get created.

Here's an example where the product category "Grains/Cereals" already exists, but the product and relationship are new:

```
MERGE (p:Product {productID: 78, pro-
  ductName: "Organic Quinoa"})
MERGE (c:Category {categoryID: 9, ca-
  tegoryName: "Grains/Cereals"})
MERGE (p)-[r:PART_OF]->(c)
RETURN *;
```

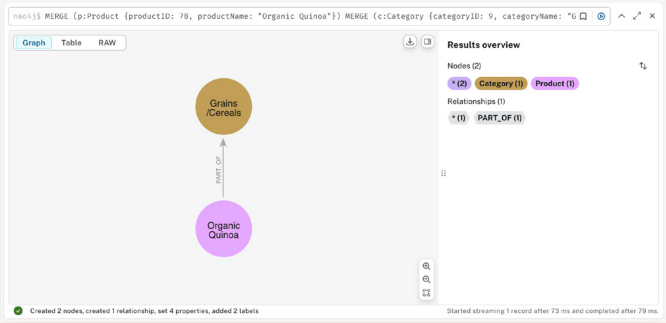

Figure 44. MERGE pattern screen

The message at the bottom of the image confirms that the Cypher statement created one new node (quinoa product) and one new relationship.

`MERGE` clauses for each node and relationship do a find-or-create operation to ensure that you only add a new product, category, or "PART_OF" relationship when each does not already exist. You can run this statement multiple times without creating duplicates because the merges will find the data that already exists.

Next Steps

Now that you have your initial version of a knowledge graph, what can you do next? Remember that a knowledge graph, especially one built on a graph database with a flexible schema like Neo4j, can expand and grow to answer more questions and serve more business needs. Here are a few ideas for expanding the utility of your knowledge graph:

- Expand your knowledge graph with additional data sources
- Load unstructured data
- Enrich the knowledge graph using graph algorithms

Rather than walk through each of these approaches step by step, this guide will provide you with some suggestions for how you can explore the next steps on your own.

Expand Your Knowledge Graph With Unstructured Data

You've built a knowledge graph with customer, product, and order information, with product categories as an organizing principle. You can broaden the types of questions the knowledge graph can answer by widening the scope of data it contains or by adding more organizing principles. You can load the data using the graphical data importer you already learned about or explore other approaches. At the time of this guide's publication, AuraDB's data importer can also connect directly to PostgreSQL, MySQL, and SQL Server databases, so you could load structured data directly from a relational database. Some ideas for other types of data and organizing principles include:

- Adding an organizing principle for the customers to help you answer questions about different customer segments. You could include location, industry, revenue, or other principles depending on the types of questions being asked by the business.
- Loading additional data about your customers (such as web clickstream activity), which would enable you to tie customers' behavior to their purchases and use the knowledge graph to offer recommendations.
- Adding supply chain information for the products in the knowledge graph, which his would enable you to use the knowledge graph to optimize the supply chain and mitigate the risk of disruption.

These ideas are, of course, just a starting point. You can follow the process outlined in this guide to come up with your own ideas.

Load Unstructured Data

One of the things that a knowledge graph built with the Neo4j graph database can do is combine structured and unstructured or semi-structured data in a single knowledge graph. Integrating these types of data enables you to answer questions that wouldn't otherwise be possible. GenAI use cases, in particular, can benefit from this capability by using vector embeddings and similarity searches to apply GraphRAG techniques for building applications that enable end users to interact with and ask questions of the knowledge graph using plain language.

You can use Neo4j's LLM Knowledge Graph Builder to load unstructured data (such as PDFs) into your existing knowledge graph. To experiment with this approach, you can use the sample PDF invoices from the Northwind order purchases to get started. The tool uses an LLM to extract nodes and relationships from unstructured content and bridge the unstructured data (with vector representations) and structured data in a single knowledge graph.

If you want to use the LLM Knowledge Graph Builder with an existing knowledge graph, you can take a couple of steps to help the Builder integrate with your existing graph. First, the Builder uses a specific label **Entity** (with two underscores each at the beginning and end) and property *id* to merge information into an existing graph; you can set this label and property by running the following Cypher:

```
MATCH (p:Product) SET p.id=p.product-
  Name, p:__Entity__
```

This query will match all existing nodes with the Product label, set the *id* property to the productName and add an **Entity** label. The Builder will use this label and property when it finds Product information in the unstructured data and use the existing Products instead of creating new ones. The other step is to provide the Builder with suggested labels and relationship types to use. Do this by clicking the **Graph Enhancement** button after you connect the Builder to your database and provide the suggested node labels and relationship types as shown in the image below:

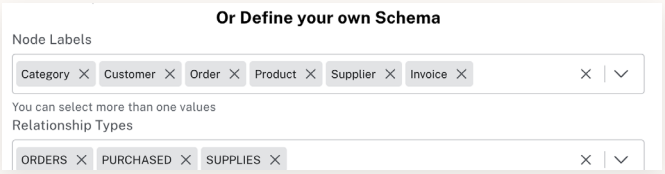

Figure 45. Providing the Builder with suggested labels and relationship types screen

Now when you load the PDF invoices, the LLM Knowledge Graph Builder will connect the invoice information to the existing products to create a single integrated knowledge graph built from both structured and unstructured data.

Enrich Your Knowledge Graph Using Graph Algorithms

In addition to making the knowledge graph more useful with more data or organizing principles, you can also use graph algorithms to uncover more insight from your knowledge graph. Algorithms such as node similarity (to use similar customers' behavior to recommend products) or pathfinding (to optimize supply chains) bring advanced capabilities that unlock deeper or previously hidden patterns in the data.

Use Cases and Design Patterns

The schema of a knowledge graph makes it straightforward to represent complex business relationships without extensive preplanning. You can incorporate additional information without having to make disruptive changes, just as we explored in the previous section.

A knowledge graph's context-rich data structure also enhances the explainability of insights since a knowledge graph stores relationships between data and its sources. Most importantly, it produces more accurate and relevant insights than siloed data systems, as it combines data from multiple systems into a single view.

Figure 46. Seven graphs of the enterprise

The next section explores a few use cases of knowledge graphs to illustrate these benefits in practice.

Supply Chain

Effective supply chain management requires understanding relationships between suppliers, distributors, warehouses, transportation logistics, raw materials, products, etc. A knowledge graph is a natural way to model and store this kind of information because the connections between different pieces of data are numerous, complex, and (often) constantly changing. Because of these characteristics, a knowledge graph provides a strong foundation for supply chain optimization, contingency planning, and risk management.

The benefits of using a knowledge graph for supply chain insights include:

- The impact of a supply chain disruption can be easily found by following the relationships downstream from the disruption.
- Graph algorithms, such as shortest path, can help to optimize delivery routes and sourcing strategies for time, cost, or other metrics.
- Graph queries can quickly identify choke points in a supply chain, which provides an opportunity to find alternative suppliers, transportation routes, etc. to mitigate the risk at that critical point in the network.

Supply chains work well as knowledge graphs because they consist of multiple complex stages, inputs, outputs, and connection points. Working with this data as a graph rather than in tables is much more intuitive and allows for better insights.

To learn more about using a knowledge graph for supply chain, check out the article series on graph data science for supply chains.

Entity Resolution

Entity resolution is the process of identifying whether multiple records are referencing the same real-world entity. In its simplest form, you can perform entity resolution with hand-crafted queries to compare key identifying attributes according to a company's business rules. However, this approach takes a lot of effort to write code and a lot of time to run the comparisons.

With a knowledge graph, you can accelerate the development and runtime requirements of entity resolution. Storing data as a knowledge graph has several advantages over other approaches:

- Shared identifiers or attributes can be easily discovered by modeling them as separate nodes in the knowledge graph. Modeling in this way makes it clear when two entities share common information and are a candidate for merging.
- Graph algorithms, such as weakly connected components, can segment the knowledge graph into separated communities, where there are no shared connections between the data. This helps reduce the number of comparisons needed in entity resolution because nodes in separate communities don't need to be compared.
- Knowledge graphs speed up transitive comparisons, which are needed to identify when more than two digital entities represent the same real-world entity.

To learn more about using a knowledge graph for entity resolution, see "Graph Data Science Use Cases: Entity Resolution."

GenAI

Despite LLMs' impressive ability to produce contextually relevant outputs, they have significant weaknesses. They lack access to real-time data, and they can't incorporate private or proprietary information not included in their training set. Furthermore, responses are unverified and don't

include the source(s) on which the LLM has based an answer. This can lead to outdated, incomplete, or incorrect responses in rapidly evolving fields or when dealing with company-specific knowledge.

Graph-based retrieval-augmented generation (GraphRAG) addresses these limitations by integrating knowledge graph data sources with LLMs. Using a knowledge graph as the data source produces better results than using a traditional database. A knowledge graph captures the context inherent in the data relationships and can provide a more complete and explainable answer than other types of data stores.

This approach results in more nuanced and accurate responses and verifiable source data, which is especially for important applications that require complex reasoning or where accuracy is critical, such as healthcare and finance.

For more information about GraphRAG techniques and implementation details, refer to "What Is GraphRAG".

Concluding Thoughts and Further Learning

Because knowledge graphs represent information as an interconnected network of entities and relationships, they reflect the complex, context-dependent nature of real-world information.

Structuring data in this way allows you to model reality with remarkable fidelity, capturing nuances across and within domains that siloed data structures often miss. You can highlight connections and insights that aren't possible with traditional data structures. A flexible structure also makes it easy to integrate new data from various sources without disrupting existing relationships.

In this guide, you learned how to create a knowledge graph from scratch and how to obtain insights from it using the Cypher graph query language. You also learned about the role the knowledge graph plays in certain domains, like supply chains, entity resolution, and GenAI.

Here are some immediate steps you can take to build upon this foundational knowledge:

- Use your Neo4j instance to experiment with different data models and explore complex queries.
- Complete some of the free self-paced courses on graph concepts and techniques in GraphAcademy.
- Join the Neo4j Community to get support and insights from fellow graph developers and enthusiasts.

Acknowledgements

This guide was developed with contributions from technical subject matter experts who helped ensure accuracy and clarity of the content. Special thanks to Jennifer Reif, John Stegeman, and Damaso Sanoja for their technical expertise and contributions to this developer guide.