Tutorial: Import data from a relational database into Neo4j

Introduction

This tutorial shows the process for exporting data from a relational database (PostgreSQL) and importing into a graph database (Neo4j). You will learn how to take data from the relational system and to the graph by translating the schema and using import tools.

Alternatively, you can:

- Create AuraDB cloud instance.
- Start a blank Neo4j Sandbox.
- Download and install Neo4j Desktop.

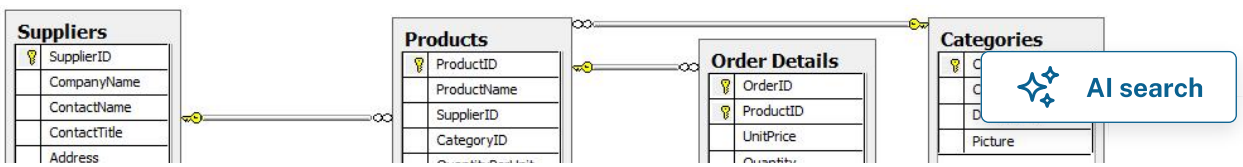
This guide uses a specific dataset, but its principles can be applied and reused with any data domain.

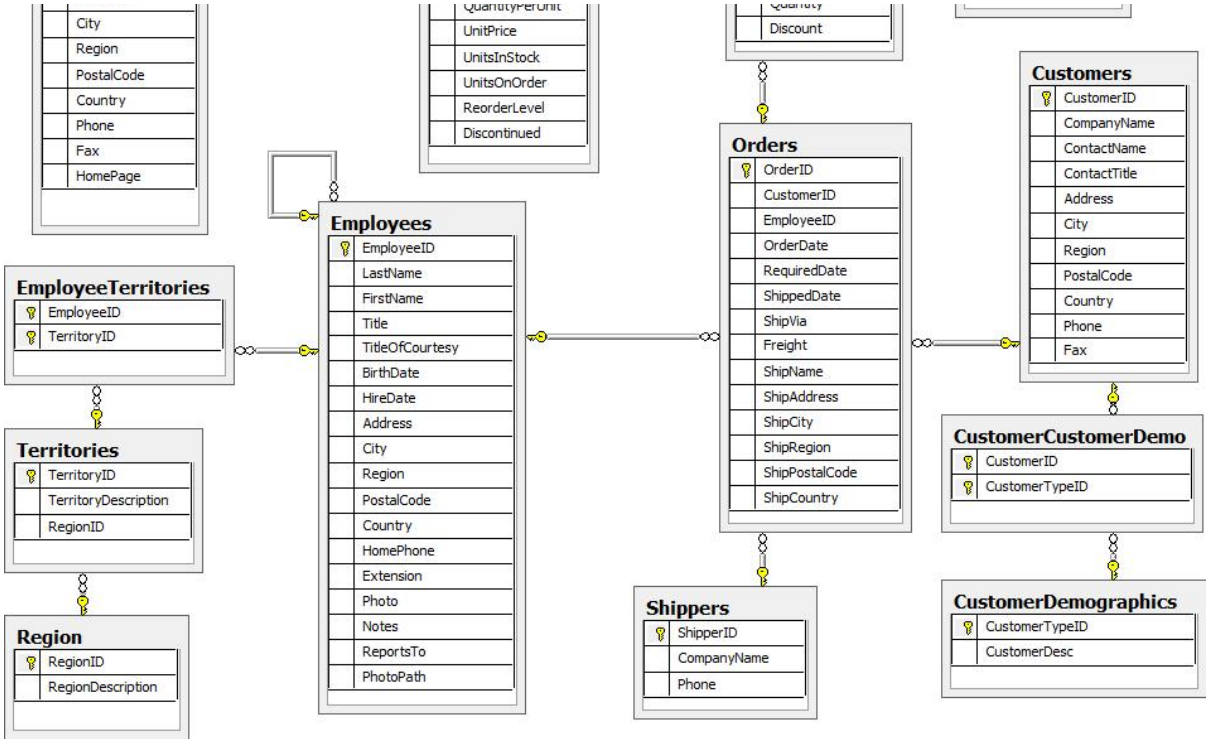
You should have a basic understanding of the property graph model and know how to model data as a graph.

About the data domain

In this guide, we will be using the Northwind dataset → , an often-used SQL dataset. This data depicts a product sale system - storing and tracking customers, products, customer orders, warehouse stock, shipping, suppliers, and even employees and their sales territories. Although the NorthWind dataset is often used to demonstrate SQL and relational databases, the data also can be structured as a graph.

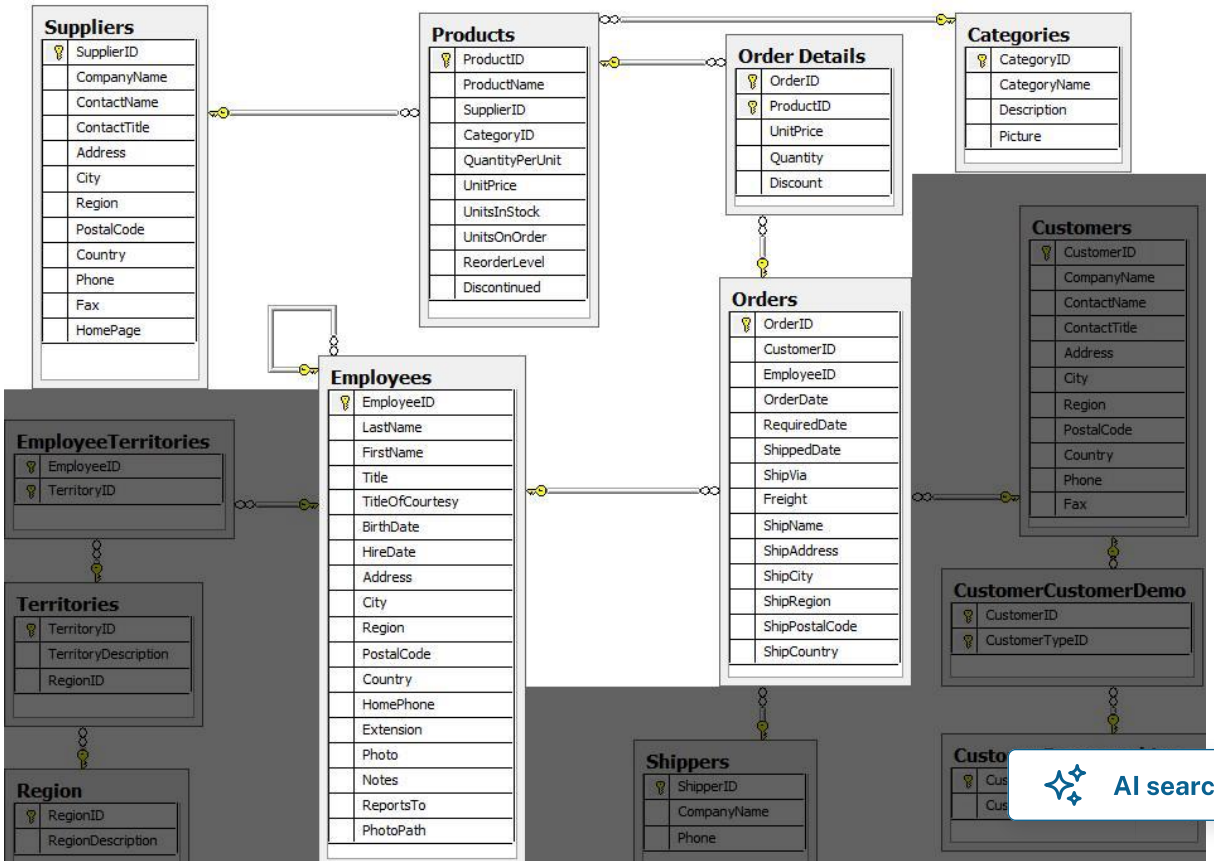
An entity-relationship diagram (ERD) of the Northwind dataset is shown below.





First, this is a rather large and detailed model. We can scale this down a bit for our example and choose the entities that are most critical for our graph - in other words, those that might benefit most from seeing the connections. For our use case, we really want to optimize the relationships with orders - what products were involved (with the categories and suppliers for those products), which employees worked on them and those employees' managers.

Using these business requirements, we can narrow our model down to these essential entities.



Developing a graph model

The first thing you will need to do to get data from a relational database into a graph is to translate the relational data model to a graph data model. Determining how you want to structure tables and rows as nodes and relationships may vary depending on what is most important to your business needs.

> **NOTE**
>
> For more information on adapting your graph model to different scenarios, check out our modeling designs guide.

When deriving a graph model from a relational model, you should keep a couple of general guidelines in mind.

1. A *row* is a *node*.
2. A *table name* is a *label name*.
3. A *join or foreign key* is a *relationship*.

With these principles in mind, we can map our relational model to a graph with the following steps:

Rows to nodes, table names to labels

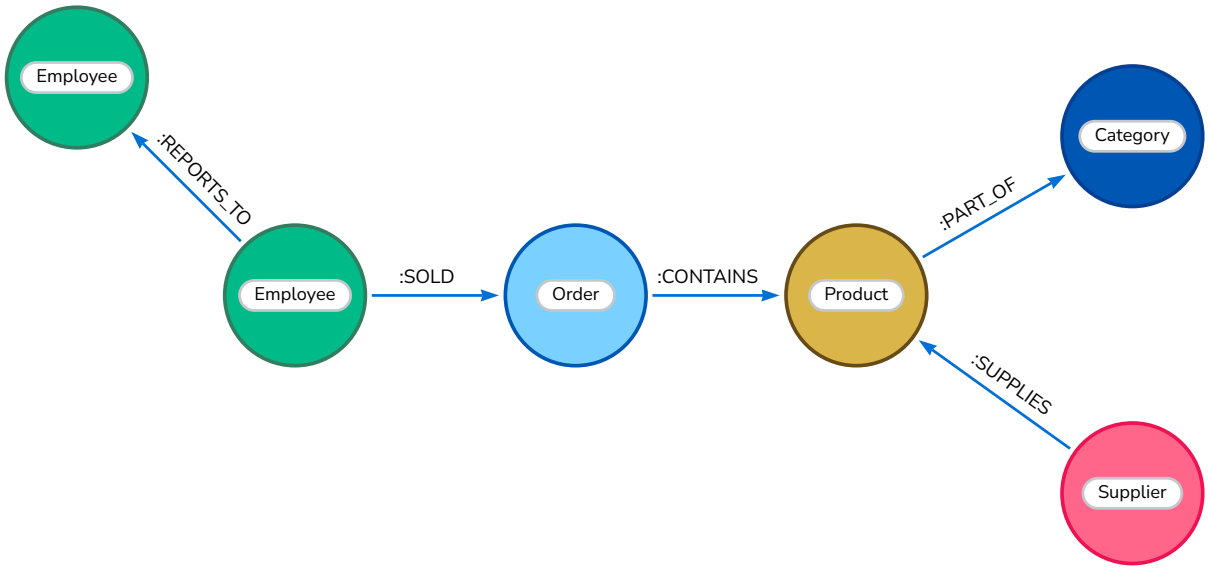
1. Each row on our `Orders` table becomes a node in our graph with `Order` as the label.
2. Each row on our `Products` table becomes a node with `Product` as the label.
3. Each row on our `Suppliers` table becomes a node with `Supplier` as the label.
4. Each row on our `Categories` table becomes a node with `Category` as the label.
5. Each row on our `Employees` table becomes a node with `Employee` as the label.

Joins to relationships

1. Join between `Suppliers` and `Products` becomes a relationship named `SUPPLIES` (where supplier supplies product).
2. Join between `Products` and `Categories` becomes a relationship named `PART_OF` (where product is part of a category).
3. Join between `Employees` and `Orders` becomes a relationship named `SO` (where employee sold an order).

4. Join between `Employees` and itself (unary relationship) becomes a relationship named `REPORTS_TO` (where employees have a manager).
5. Join with join table (`Order Details`) between `Orders` and `Products` becomes a relationship named `CONTAINS` with properties of `unitPrice`, `quantity`, and `discount` (where order contains a product).

If we draw our translation out on the whiteboard, we have this graph data model.



Now, we can, of course, decide that we want to include the rest of the entities from our relational model, but for now, we will keep to this smaller graph model.

How does the graph model differ from the relational model?

- There are no nulls. Non-existing value entries (properties) are just not present.
- It describes the relationships in more detail. For example, we know that an employee SOLD an order rather than having a foreign key relationship between the Orders and Employees tables. We could also choose to add more metadata about that relationship, should we wish.
- Either model can be more normalized. For example, addresses have been denormalized in several of the tables, but could have been in a separate table. In a future version of our graph model, we might also choose to separate addresses from the `Order` (or `Supplier` or `Employee`) entities and create separate `Address` nodes.

Exporting relational tables to CSV

Thankfully, this step has already been done for you with the Northwind data you will use later on in this guide.

However, if you are working with another data domain, you need to take the data from the relational tables and put it in another format for loading to the graph. A common format that many systems can handle a flat file of comma-separated values (CSV).

Here is an example script we already ran to export the northwind data into CSV files for you.

export_csv.sql

```
COPY (SELECT * FROM customers) TO '/tmp/customers.csv' WITH CSV header;
COPY (SELECT * FROM suppliers) TO '/tmp/suppliers.csv' WITH CSV header;
COPY (SELECT * FROM products)  TO '/tmp/products.csv' WITH CSV header;
COPY (SELECT * FROM employees) TO '/tmp/employees.csv' WITH CSV header;
COPY (SELECT * FROM categories) TO '/tmp/categories.csv' WITH CSV header;

COPY (SELECT * FROM orders
      LEFT OUTER JOIN order_details ON order_details.OrderID = orders.OrderID) TO '/tmp/
orders.csv' WITH CSV header;
```

If you want to create the CSV files yourself using your own northwind RDBMS, you can run this script against your RDBMS with the command `psql -d northwind < export_csv.sql`.

Note: You need not run this script unless you want to execute it against your own northwind RDBMS.

Importing the data using Cypher

You can use Cypher®'s `LOAD CSV` command to transform the contents of the CSV file into a graph structure.

When you use `LOAD CSV` to create nodes and relationships in the database, you have two options for where the CSV files reside:

- In the **import** folder for the Neo4j instance that you can manage.
- From a publicly-available location such as an S3 bucket or a github location. You must use this option if you are using Neo4j AuraDB or Neo4j Sandbox.

If you want to use the CSV files for the Neo4j instance you manage, you can copy the CSV files from Northwind files on GitHub → and place them in the **import** folder for your Neo4j DBMS.

You use use Cypher's `LOAD CSV` statement to read each file and add Cypher c

take the row/column data and transform it to the graph.

Next you will run Cypher code to:

1. Load the nodes from the CSV files.
2. Create the indexes and constraints for the data in the graph.
3. Create the relationships between the nodes.

Creating Order nodes

Execute this Cypher block to create the Order nodes in the database:

```
// Create orders
LOAD CSV WITH HEADERS FROM 'https://gist.githubusercontent.com/
jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f419b5a0cb8ac4f/
orders.csv' AS row
MERGE (order:Order {orderID: row.OrderID})
  ON CREATE SET order.shipName = row.ShipName;
```

If you have placed the CSV files in to the **import** folder, you should use this code syntax to load the CSV files from a local directory:

```
// Create orders
LOAD CSV WITH HEADERS FROM 'file:///orders.csv' AS row
MERGE (order:Order {orderID: row.OrderID})
  ON CREATE SET order.shipName = row.ShipName;
```

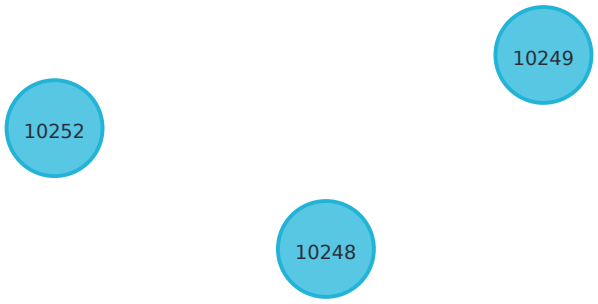
This code creates 830 `Order` nodes in the database.

You can view some of the nodes in the database by executing this code:

```
MATCH (o:Order) return o LIMIT 5;
```

The graph view is:

AI search



The table view contains these values for the node properties:

o
{"shipName":Vins et alcools Chevalier,"orderID":10248}
{"shipName":Toms Spezialitäten,"orderID":10249}
{"shipName":Hanari Carnes,"orderID":10250}
{"shipName":Victuailles en stock,"orderID":10251}
{"shipName":Suprêmes délices,"orderID":10252}

You might notice that you have not imported all of the field columns in the CSV file. With your statements, you can choose which properties are needed on a node, which can be left out, and which might need imported to another node type or relationship. You might also notice that you used the `MERGE` keyword, instead of `CREATE`. Though we feel pretty confident there are no duplicates in our CSV files, we can use `MERGE` as good practice for ensuring unique entities in our database.

Creating Product nodes

Execute this code to create the Product nodes in the database:

```
// Create products
LOAD CSV WITH HEADERS FROM 'https://gist.githubusercontent.com/
jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f419b5a0cb8ac4f/
products.csv' AS row
MERGE (product:Product {productID: row.ProductID})
  ON CREATE SET product.productName = row.ProductName, product.unitPrice =
toFloat(row.UnitPrice);
```

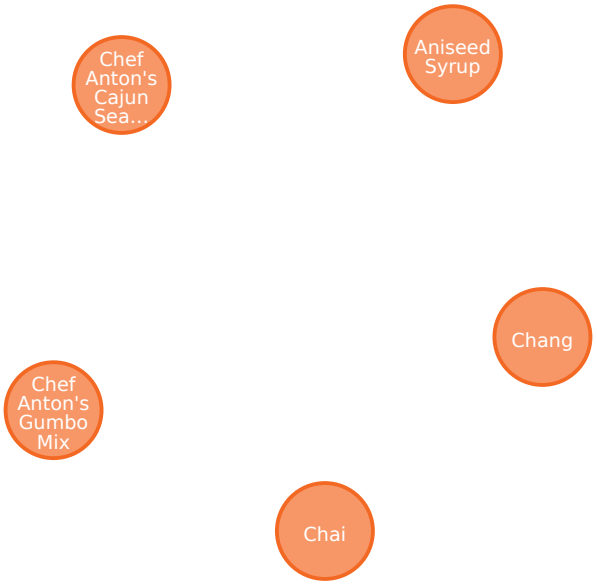
AI search

This code creates 77 `Product` nodes in the database.

You can view some of these nodes in the database by executing this code:

```
MATCH (p:Product) return p LIMIT 5;
```

The graph view is:



The table view contains these values for the node properties:

p
{"unitPrice":18.0,"productID":1,"productName":Chai}
{"unitPrice":19.0,"productID":2,"productName":Chang}
{"unitPrice":10.0,"productID":3,"productName":Aniseed Syrup}
{"unitPrice":22.0,"productID":4,"productName":Chef Anton’s Cajun Seasoning}
{"unitPrice":21.35,"productID":5,"productName":Chef Anton’s Gumbo Mix}

Creating Supplier nodes

Execute this code to create the Supplier nodes in the database:

```
// Create suppliers
LOAD CSV WITH HEADERS FROM 'https://gist.githubusercontent.com/
```

AI search

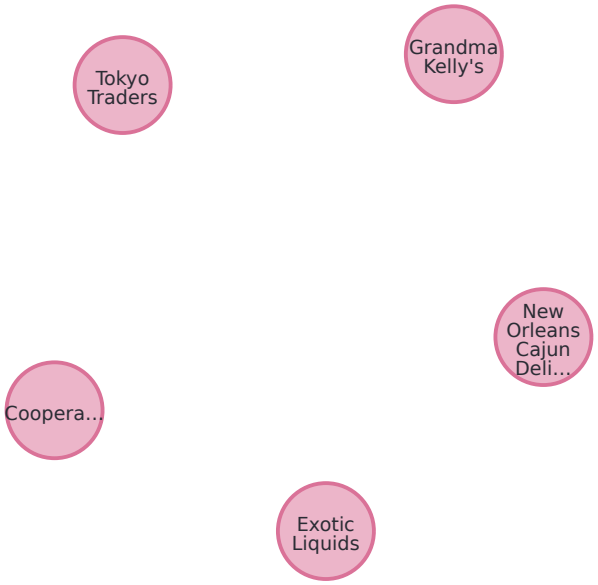
```
jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f419b5a0cb8ac4f/
suppliers.csv' AS row
MERGE (supplier:Supplier {supplierID: row.SupplierID})
  ON CREATE SET supplier.companyName = row.CompanyName;
```

This code creates 29 `Supplier` nodes in the database.

You can view some of these nodes in the database by executing this code:

```
MATCH (s:Supplier) return s LIMIT 5;
```

The graph view is:



The table view contains these values for the node properties:

s
{"supplierID":1,"companyName":Exotic Liquids}
{"supplierID":2,"companyName":New Orleans Cajun Delights}
{"supplierID":3,"companyName":Grandma Kelly’s Homestead}
{"supplierID":4,"companyName":Tokyo Traders}
{"supplierID":5,"companyName":Cooperativa de Quesos 'Las Cabras'}

Creating Employee nodes

Execute this code to create the Supplier nodes in the database:

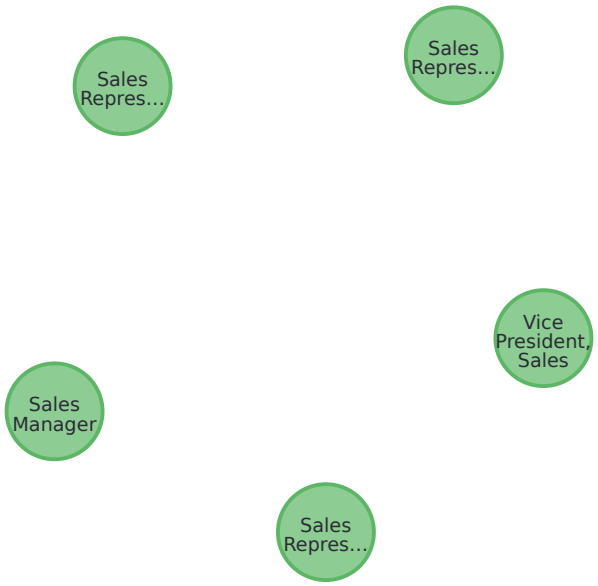
```
// Create employees
LOAD CSV WITH HEADERS FROM 'https://gist.githubusercontent.com/
jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f419b5a0cb8ac4f/
employees.csv' AS row
MERGE (e:Employee {employeeID:row.EmployeeID})
  ON CREATE SET e.firstName = row.FirstName, e.lastName = row.LastName, e.title = row.Title;
```

This code creates 9 `Employee` nodes in the database.

You can view some of these nodes in the database by executing this code:

```
MATCH (e:Employee) return e LIMIT 5;
```

The graph view is:



The table view contains these values for the node properties:

e
{"lastName":Davolio,"firstName":Nancy,"employeeID":1,"title":Sales Representative}
{"lastName":Fuller,"firstName":Andrew,"employeeID":2,"title":Vice President, Sales}
{"lastName":Leverling,"firstName":Janet,"employeeID":3,"title":Sales Representative}
{"lastName":Peacock,"firstName":Margaret,"employeeID":4,"title":Sales Representative}

e
{"lastName":Buchanan,"firstName":Steven,"employeeID":5,"title":Sales Manager}

Creating Category nodes

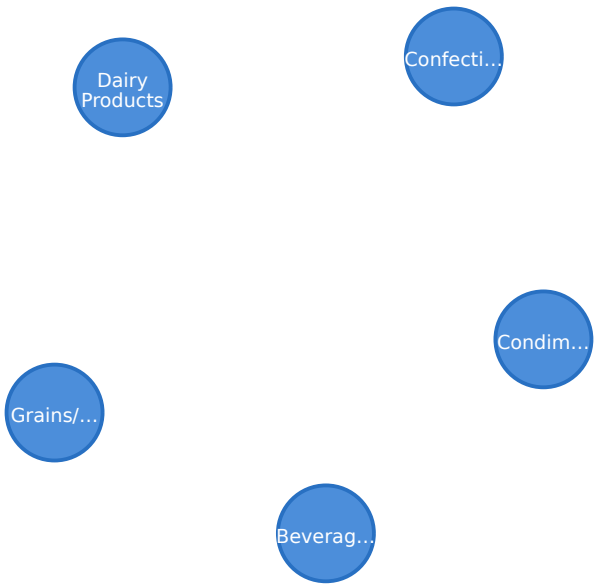
```
// Create categories
LOAD CSV WITH HEADERS FROM 'https://gist.githubusercontent.com/
jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f419b5a0cb8ac4f/
categories.csv' AS row
MERGE (c:Category {categoryID: row.CategoryID})
  ON CREATE SET c.categoryName = row.CategoryName, c.description = row.Description;
```

This code creates 8 `Category` nodes in the database.

You can view some of these nodes in the database by executing this code:

```
MATCH (c:Category) return c LIMIT 5;
```

The graph view is:



The table view contains these values for the node properties:

c
{"description":Soft drinks, coffees, teas, beers, and ales,"categoryName":Beverages,"categor

AI search

c
{"description":Sweet and savory sauces, relishes, spreads, and seasonings,"categoryName":Condiments,"categoryID":2}
{"description":Desserts, candies, and sweet breads,"categoryName":Confections,"categoryID":3}
{"description":Cheeses,"categoryName":Dairy Products,"categoryID":4}
{"description":Breads, crackers, pasta, and cereal,"categoryName":Grains/Cereals,"categoryID":5}

NOTE

For very large commercial or enterprise datasets, you may find out-of-memory errors, especially on smaller machines. To avoid these situations, you can use `CALL IN {…} TRANSACTIONS` subquery to commit data in batches. Don't forget to prepend this query with `:auto` in Neo4j Browser. This practice is not standard recommendation for smaller datasets, but is only recommended when memory issues are threatened. More information on this subquery can be found in the Cypher manual → Subqueries in transactions.

Creating the indexes and constraints for the data in the graph

After the nodes are created, you need to create the relationships between them. Importing the relationships will mean looking up the nodes you just created and adding a relationship between those existing entities. To ensure the lookup of nodes is optimized, you will create indexes for any node properties used in the lookups (often the ID or another unique value).

We also want to create a constraint (also creates an index with it) that will disallow orders with the same id from getting created, preventing duplicates. Finally, as the indexes are created after the nodes are inserted, their population happens asynchronously, so we call `db.awaitIndexes()` to block until they are populated.

Execute this code block:

```
CREATE INDEX product_id FOR (p:Product) ON (p.productID);
CREATE INDEX product_name FOR (p:Product) ON (p.productName);
CREATE INDEX supplier_id FOR (s:Supplier) ON (s.supplierID);
CREATE INDEX employee_id FOR (e:Employee) ON (e.employeeID);
CREATE INDEX category_id FOR (c:Category) ON (c.categoryID);
CREATE CONSTRAINT order_id FOR (o:Order) REQUIRE o.orderID IS UNIQUE;
CALL db.awaitIndexes();
```

AI search

After you execute this code, you can run the following Cypher command to view the indexes in the database:

```
SHOW INDEXES;
```

Two token lookup indexes (one for node labels and one for relationship types) are present by default when creating a Neo4j database. They exclusively solve node label and relationship type predicates and assist with the population of other indexes. Deleting them may have negative performance implications. You should see these indexes (and constraint) in the database:

```
+-----------------------------------------------------------------------------------------------
-------------------------------------------------------------------------+
|id|name          |state |populationPercent|type  |entityType  |labelsOrTypes|properties   |
indexprovider    |owningConstraint|lastRead                         |readCount|
+-----------------------------------------------------------------------------------------------
-------------------------------------------------------------------------+
|7 |category_id    |ONLINE|100.0            |RANGE |NODE        |[Category]   |[categoryID] |
range-1.0        |null            |null                             |0        |
+-----------------------------------------------------------------------------------------------
-------------------------------------------------------------------------+
|6 |employee_id    |ONLINE|100.0            |RANGE |NODE        |[Employee]   |[employeeID] |
range-1.0        |null            |null                             |0        |
+----------------------------------                                  ----------------------------------
----------------------------------  View all (5 more lines)  --------------+
|1 |index_343aff4e|ONLINE|100.0            |LOOKUP|NODE        |null         |null         |
```

For more information on indexes and their use in Neo4j, go to the Cypher Manual → The use of indexes.

Creating the relationships between the nodes

Next you have to create relationships:

1. Between Orders and Employees.
2. Between Products and Suppliers and between Products and Categories.
3. Between Employees.

Creating relationships between Orders and Employees

With the initial nodes and indexes in place, you can now create the relationships for orders to products and orders to employees.

AI search

Execute this code block:

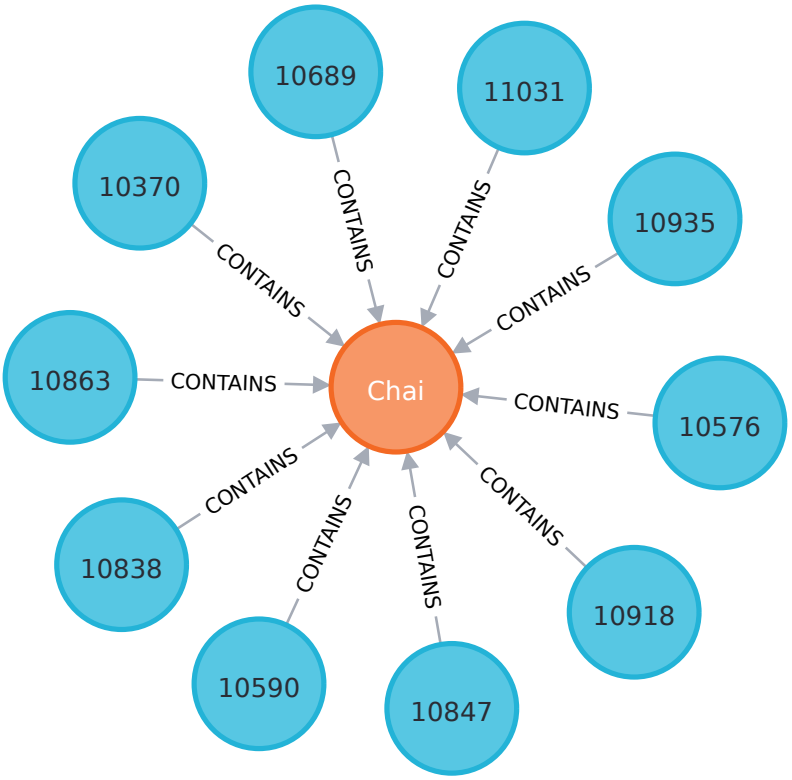
```
// Create relationships between orders and products
LOAD CSV WITH HEADERS FROM 'https://gist.githubusercontent.com/
jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f419b5a0cb8ac4f/
orders.csv' AS row
MATCH (order:Order {orderID: row.OrderID})
MATCH (product:Product {productID: row.ProductID})
MERGE (order)-[op:CONTAINS]->(product)
  ON CREATE SET op.unitPrice = toFloat(row.UnitPrice), op.quantity = toFloat(row.Quantity);
```

This code creates 2155 relationships in the graph.

You can view some of them by executing this code:

```
MATCH (o:Order)-[]-(p:Product)
RETURN o,p LIMIT 10;
```

Your graph view should look something like this:



Then, execute this code block:

```
// Create relationships between orders and employees
```

AI search

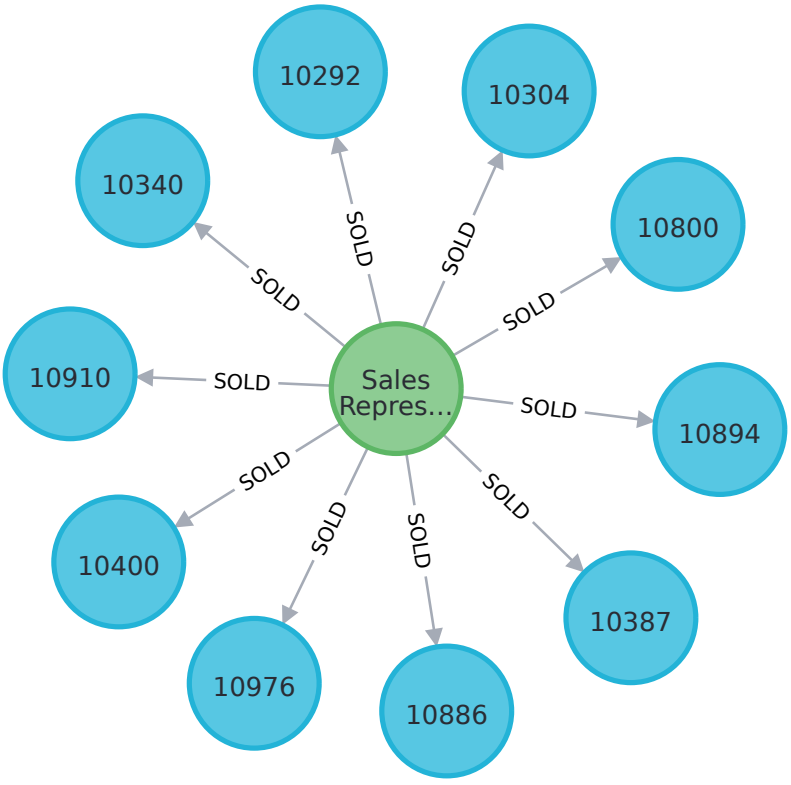
```
LOAD CSV WITH HEADERS FROM 'https://gist.githubusercontent.com/
jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f419b5a0cb8ac4f/
orders.csv' AS row
MATCH (order:Order {orderID: row.OrderID})
MATCH (employee:Employee {employeeID: row.EmployeeID})
MERGE (employee)-[:SOLD]->(order);
```

This code creates 830 relationships in the graph.

You can view some of them by executing this code:

```
MATCH (o:Order)-[]-(e:Employee)
RETURN o,e LIMIT 10;
```

Your graph view should look something like this:



Creating relationships between Products and Suppliers and between Products and Categories

Next, create relationships between Products, Suppliers, and Categories:

Execute this code block:

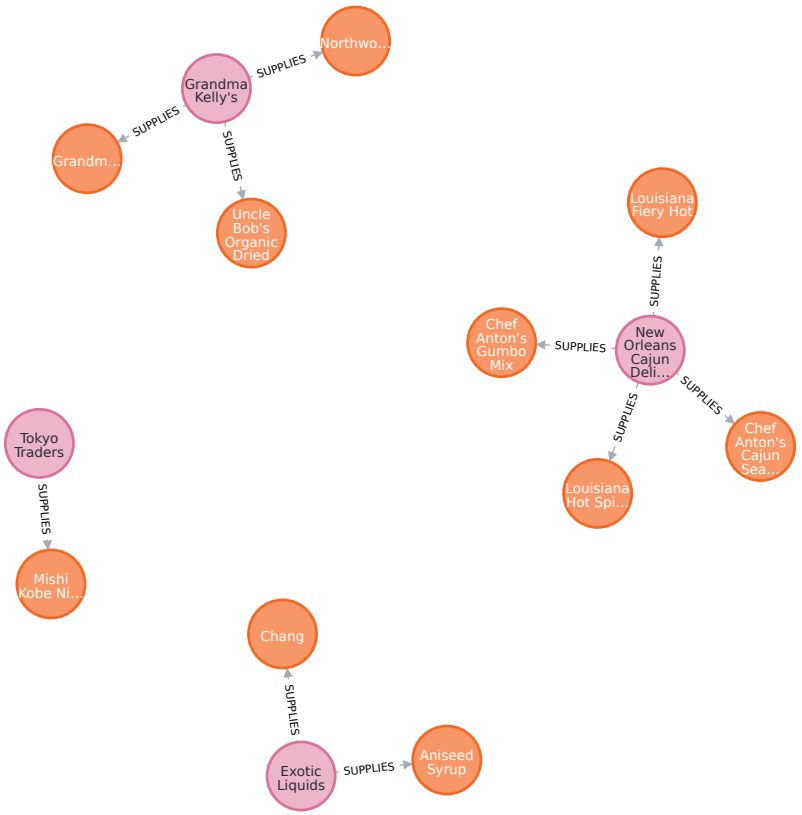
```
// Create relationships between products and suppliers
LOAD CSV WITH HEADERS FROM 'https://gist.githubusercontent.com/
jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f419b5a0cb8ac4f/
products.csv
' AS row
MATCH (product:Product {productID: row.ProductID})
MATCH (supplier:Supplier {supplierID: row.SupplierID})
MERGE (supplier)-[:SUPPLIES]->(product);
```

This code creates 77 relationships in the graph.

You can view some of them by executing this code:

```
MATCH (s:Supplier)-[]-(p:Product)
RETURN s,p LIMIT 10;
```

Your graph view should look something like this:



Then, execute this code block:

```
// Create relationships between products and categories
LOAD CSV WITH HEADERS FROM 'https://gist.githubusercontent.com/
jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f419b5a0cb8ac4f/
```

AI search

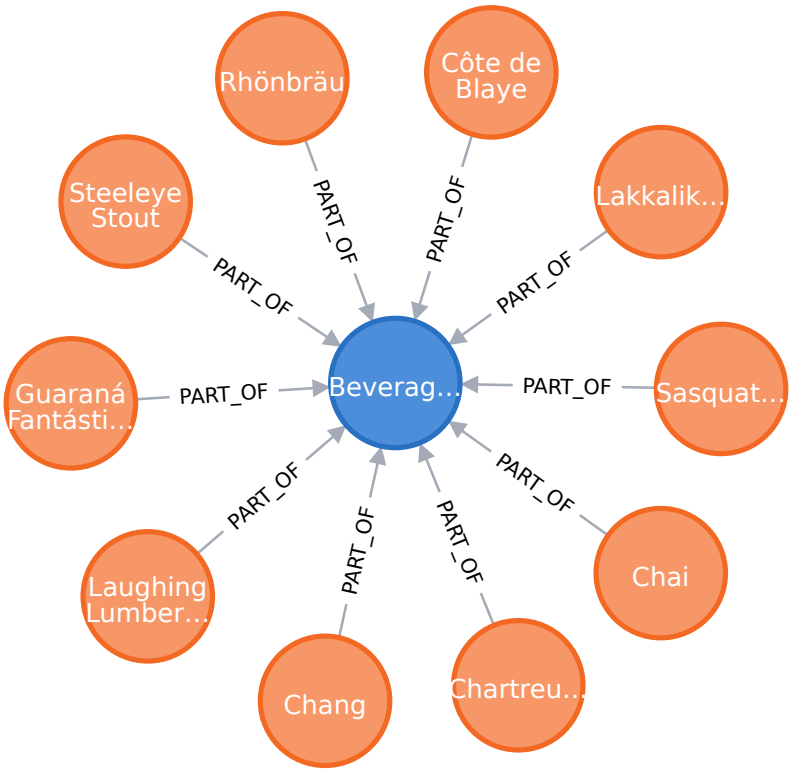
```
products.csv
' AS row
MATCH (product:Product {productID: row.ProductID})
MATCH (category:Category {categoryID: row.CategoryID})
MERGE (product)-[:PART_OF]->(category);
```

This code creates 77 relationships in the graph.

You can view some of them by executing this code:

```
MATCH (c:Category)-[]-(p:Product)
RETURN c,p LIMIT 10;
```

Your graph view should look something like this:



Creating relationships between Employees

Lastly, you will create the 'REPORTS_TO' relationship between Employees to represent the reporting structure:

Execute this code block:

```
// Create relationships between employees (reporting hierarchy)
```

AI search

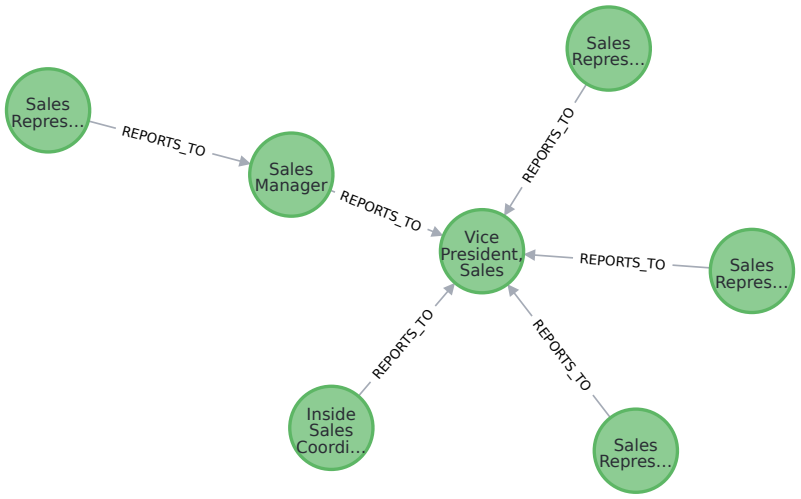
```
LOAD CSV WITH HEADERS FROM 'https://gist.githubusercontent.com/
jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f419b5a0cb8ac4f/
employees.csv' AS row
MATCH (employee:Employee {employeeID: row.EmployeeID})
MATCH (manager:Employee {employeeID: row.ReportsTo})
MERGE (employee)-[:REPORTS_TO]->(manager);
```

This code creates 8 relationships in the graph.

You can view some of them by executing this code:

```
MATCH (e1:Employee)-[]-(e2:Employee)
RETURN e1,e2 LIMIT 10;
```

Your graph view should look something like this:



Next, you will query the resulting graph to find out what it can tell us about our newly-imported data.

Querying the graph

We might start with a couple of general queries to verify that our data matches the model we designed earlier in the guide. Here are some example queries.

Execute this code block:

AI search

```
//find a sample of employees who sold orders with their ordered products
MATCH (e:Employee)-[rel:SOLD]->(o:Order)-[rel2:CONTAINS]->(p:Product)
RETURN e, rel, o, rel2, p LIMIT 25;
```

Execute this code block:

```
//find the supplier and category for a specific product
MATCH (s:Supplier)-[r1:SUPPLIES]->(p:Product {productName: 'Chocolade'})-[r2:PART_OF]-
>(c:Category)
RETURN s, r1, p, r2, c;
```

Once you are comfortable that the data aligns with our data model and everything looks correct, you can start querying to gather information and insights for business decisions.

Which Employee had the highest cross-selling count of 'Chocolade' and another product?

Execute this code block:

```
MATCH (choc:Product {productName:'Chocolade'})<-[:CONTAINS]-(:Order)<-[:SOLD]-(employee),
      (employee)-[:SOLD]->(o2)-[:CONTAINS]->(other:Product)
RETURN employee.employeeID as employee, other.productName as otherProduct, count(distinct o2)
as count
ORDER BY count DESC
LIMIT 5;
```

Looks like employee No. 4 was busy, though employee No. 1 also did well! Your results should look something like this:

employee	otherProduct	count
4	Gnocchi di nonna Alice	14
4	Pâté chinois	12
1	Flotemysost	12
3	Gumbär Gummibärchen	12
1	Pavlova	11

How are Employees organized? Who reports to whom?

Execute this code block:

```
MATCH (e:Employee)<-[:REPORTS_TO]-(sub)
RETURN e.employeeID AS manager, sub.employeeID AS employee;
```

Your results should look something like this:

manager	employee
2	3
2	4
2	5
2	1
2	8
5	9
5	7
5	6

Notice that employee No. 5 has people reporting to them but also reports to employee No. 2.

Next, let's investigate that a bit more.

Which Employees report to each other indirectly?

Execute this code block:

```
MATCH path = (e:Employee)<-[:REPORTS_TO*]-(sub)
WITH e, sub, [person in NODES(path) | person.employeeID][1..-1] AS path
RETURN e.employeeID AS manager, path as middleManager, sub.employeeID AS employee
ORDER BY size(path);
```

Your results should look something like this:

manager	middleManager	employee
2	[]	3
2	[]	4
2	[]	5
2	[]	1
2	[]	8
5	[]	9
5	[]	7
5	[]	6
2	[5]	9
2	[5]	7
2	[5]	6

How many orders were made by each part of the hierarchy?

Execute this code block:

```
MATCH (e:Employee)
OPTIONAL MATCH (e)<-[:REPORTS_TO*0..]-(sub)-[:SOLD]->(order)
RETURN e.employeeID as employee, [x IN COLLECT(DISTINCT sub.employeeID) WHERE x <>
e.employeeID] AS reportsTo, COUNT(distinct order) AS totalOrders
ORDER BY totalOrders DESC;
```

Your results should look something like this:

employee	reportsTo	totalOrders
2	[8,1,5,6,7,9,4,3]	830
5	[6,7,9]	224
4	[]	156
3	[]	127
1	[]	123

employee	reportsTo	totalOrders
8	[]	104
7	[]	72
6	[]	67
9	[]	43

What's next?

If you followed along with each step through this guide, then you might want to explore the data set with more queries and try to answer additional questions you came up with for the data. You may also want to apply these same principles to your own or another data set for analysis.

If you used this as a process flow to apply to a different data set or you would like to do that next, feel free to start at the top and work through this guide again with another domain. The steps and processes still apply (though, of course, the data model, queries, and business questions will need adjusted).

If you have data that needs additional cleansing and manipulation than what is covered in this guide, the APOC library may be able to help. It contains hundreds of procedures and functions for handling large amounts of data, translating values, cleaning messy data sources, and more!

If you are interested in doing a one-time initial dump of relational data to Neo4j, then the Neo4j ETL Tool might be what you are looking for. The application is designed with a point-and-click user interface with the goal of fast, simple relational-to-graph loads that help new and existing users gain faster value from seeing their data as a graph without Cypher, import procedures, or other code.

Resources

- Northwind SQL, CSV and Cypher data files →, also as zip → file
- LOAD CSV: Cypher's command for importing CSV files
- APOC library: Neo4j's utility library
- Neo4j ETL Tool: Loading relational data without code
- Importing Data with Neo4j
- Graph Data Modeling

Prev
‹ Import CSV data with Neo4j Desktop

Next
Resources ›

Contents