→ Developer
→ Data Import

Goals

This guide explores how to export data from a PostgreSQL database (RDBMS) for import into Neo4j (GraphDB). You'll learn how to take a relational database schema and model it as a graph for import into Neo4j.

Prerequisites

You should have a basic understanding of the property graph model and have completed the modeling guide. If you download and install the Neo4j server you'll be able to follow along with the examples.

Beginner

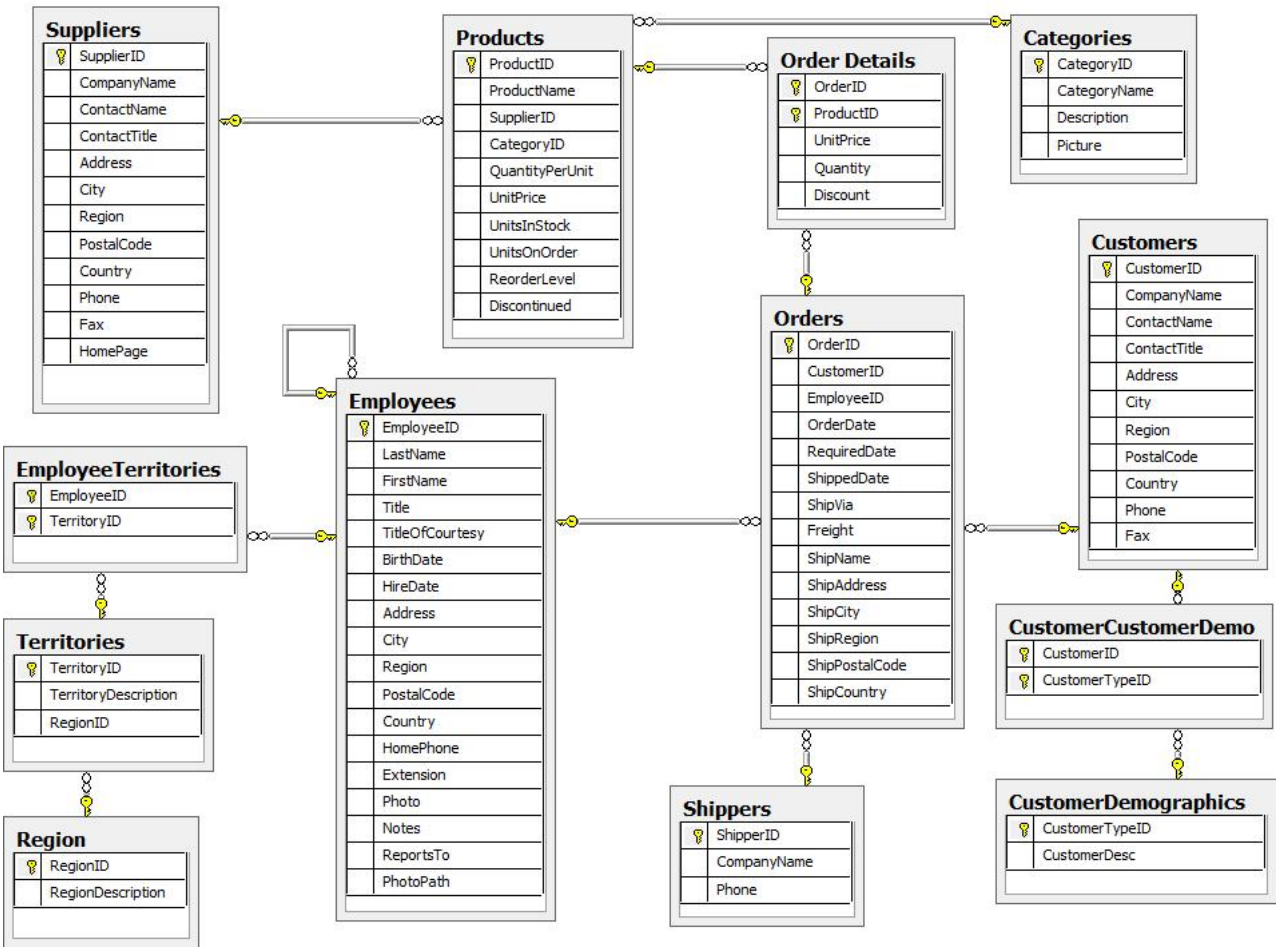
Overview

- NorthWind Introduction
- Developing a Graph Model
- Exporting the Data to CSV
- Importing the Data using Cypher
- Querying the Graph
- Updating the Graph

NorthWind Introduction

In this guide we'll be using the NorthWind dataset, a commonly used SQL dataset. Although the NorthWind dataset is often used to demonstrate SQL and relational databases, it is graphy enough to be interesting for us.

The following is an entity relationship diagram of the NorthWind dataset:

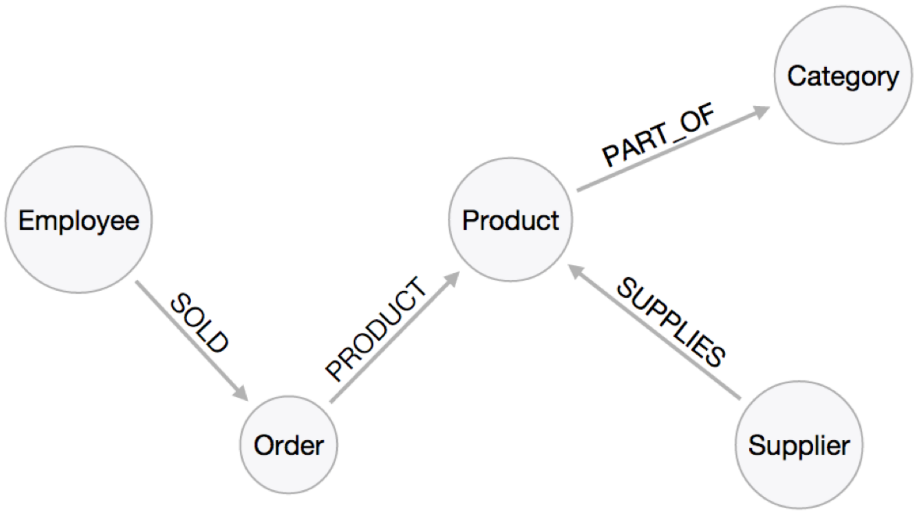


Developing a Graph Model

When deriving a graph model from a relational model, we should keep the following guidelines in mind:

- A *row* is a *node*
- A *table name* is a *label name*

In this dataset, the following graph model serves as a first iteration:



How does the Graph Model Differ from the Relational Model?

- There are no nulls.
  - In the relational version, to track the employee hierarchy we have a null entry in the 'ReportsTo' column if they don't report to anybody. In the graph version we just don't define a relationship.
  - 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.
- It will often be more normalised. For example, addresses have been denormalised in several of the tables but in a future version of the graph model we might choose to make addresses nodes in their own rights.

Exporting the Data to CSV

Now that we know what we'd like our graph to look like, we need to extract the data from PostgreSQL so we can create it as a graph. The easiest way to do that is to export the appropriate tables in CSV format. The PostgreSQL 'copy' command lets us execute a SQL query and write the result to a CSV file, e.g. with `psql -d northwind < export_csv.sql`:

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;
```

Importing the Data using Cypher

After we've exported our data from PostgreSQL, we'll use Cypher's LOAD CSV command to transform the contents of the CSV file into a graph structure.

First, create the nodes:

import_csv.cypher

```
// Create customers
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:customers.csv" AS row
CREATE (:Customer {companyName: row.CompanyName, customerID: row.CustomerID, fax: row.Fax, phone: row.Phone});

// Create products
```

```
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:products.csv" AS row
CREATE (:Product {productName: row.ProductName, productID: row.ProductID, unitPrice: toFloat(row.UnitPrice)});

// Create suppliers
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:suppliers.csv" AS row
CREATE (:Supplier {companyName: row.CompanyName, supplierID: row.SupplierID});

// Create employees
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:employees.csv" AS row
CREATE (:Employee {employeeID:row.EmployeeID,  firstName: row.FirstName, lastName: row.LastName, title: row.Title});

// Create categories
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:categories.csv" AS row
CREATE (:Category {categoryID: row.CategoryID, categoryName: row.CategoryName, description: row.Description});

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

Next, we'll create indexes on the just-created nodes to ensure their quick lookup when creating relationships in the next step.

```
CREATE INDEX ON :Product(productID);
CREATE INDEX ON :Product(productName);
CREATE INDEX ON :Category(categoryID);
CREATE INDEX ON :Employee(employeeID);
CREATE INDEX ON :Supplier(supplierID);
CREATE INDEX ON :Customer(customerID);
CREATE INDEX ON :Customer(customerName);

CREATE CONSTRAINT ON (o:Order) ASSERT o.orderID IS UNIQUE;
```

As the indexes are created after the nodes are inserted, their population happens asynchronously, so we use `schema await` (a shell command) to block until they are populated.

```
schema await
```

Initial nodes and indices in place, we can now create relationships of orders to products and employees.

```
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:orders.csv" AS row
MATCH (order:Order {orderID: row.OrderID})
MATCH (product:Product {productID: row.ProductID})
MERGE (order)-[pu:PRODUCT]->(product)
ON CREATE SET pu.unitPrice = toFloat(row.UnitPrice), pu.quantity = toFloat(row.Quantity);

USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:orders.csv" AS row
MATCH (order:Order {orderID: row.OrderID})
MATCH (employee:Employee {employeeID: row.EmployeeID})
MERGE (employee)-[:SOLD]->(order);

USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:orders.csv" AS row
MATCH (order:Order {orderID: row.OrderID})
MATCH (customer:Customer {customerID: row.CustomerID})
MERGE (customer)-[:PURCHASED]->(order);
```

Next, create relationships between products, suppliers, and categories:

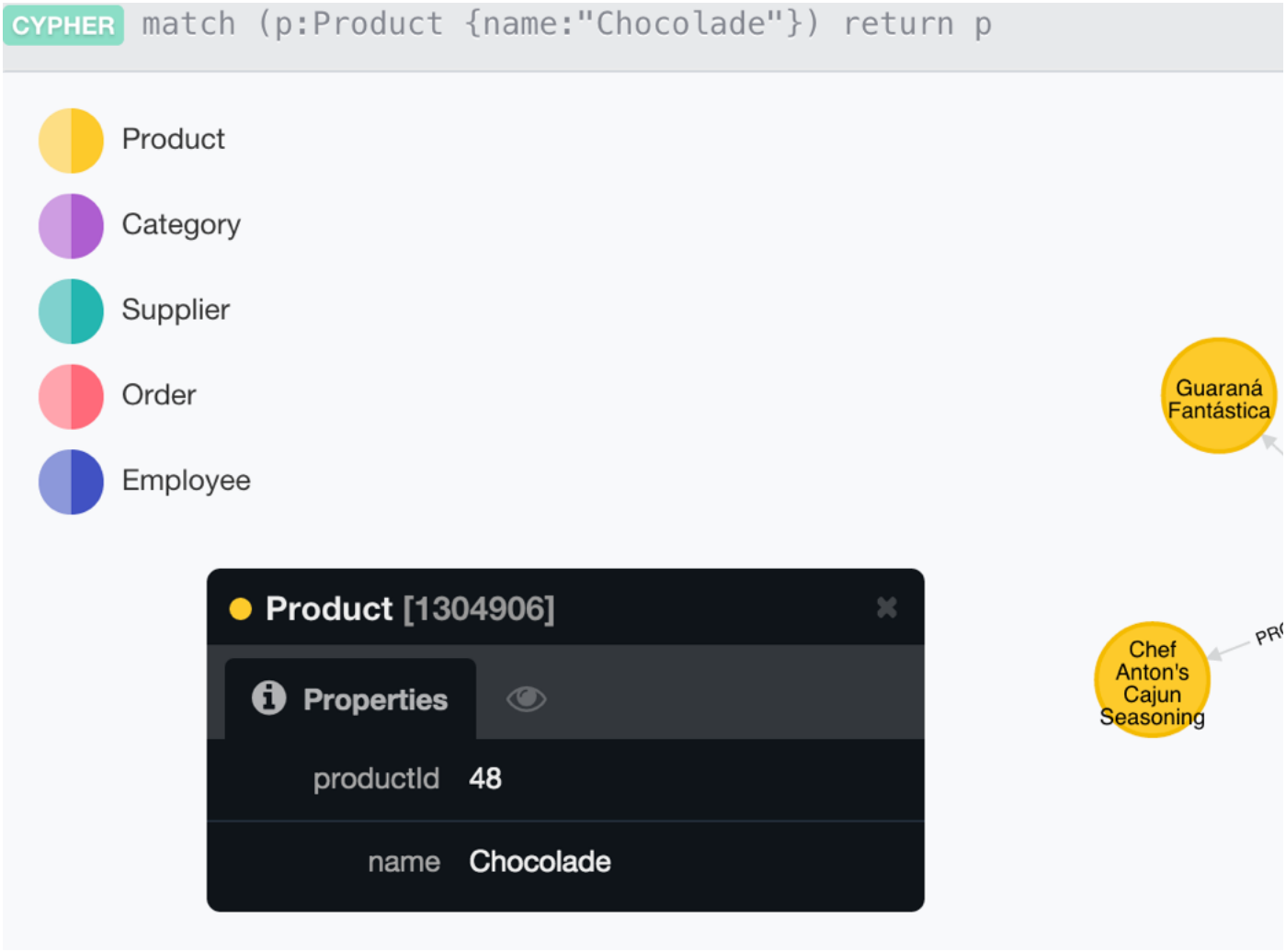
```
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:products.csv" AS row
MATCH (product:Product {productID: row.ProductID})
MATCH (supplier:Supplier {supplierID: row.SupplierID})
MERGE (supplier)-[:SUPPLIES]->(product);

USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:products.csv" AS row
MATCH (product:Product {productID: row.ProductID})
MATCH (category:Category {categoryID: row.CategoryID})
MERGE (product)-[:PART_OF]->(category);
```

Finally we'll create the 'REPORTS_TO' relationship between employees to represent the reporting structure:

```
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:employees.csv" AS row
MATCH (employee:Employee {employeeID: row.EmployeeID})
MATCH (manager:Employee {employeeID: row.ReportsTo})
MERGE (employee)-[:REPORTS_TO]->(manager);
```

You can also run the whole script at once using `bin/neo4j-shell -path northwind.db -file import_csv.cypher`.

The resulting graph should look like this:



We can now query the resulting graph.

Querying the Graph

One question we might be interested in is:

Which Employee had the Highest Cross-Selling Count of 'Chocolade' and Which Product?

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

Looks like employee No. 1 was very busy!

employee.employeeId	other.productName	count
1	Pavlova	56
1	Camembert Pierrot	56
1	Ikura	55
1	Chang	47
1	Pâté chinois	45

We might also like to answer the following question:

How are Employees Organized? Who Reports to Whom?

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

manager	employee
2	1

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

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

Let’ s investigate that a bit more:

Which Employees Report to Each Other Indirectly?

```
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, sub.employeeID AS employee, CASE WHEN LENGTH(path) = 0 THEN "Direct Report" ELSE path END AS via
ORDER BY LENGTH(path);
```

e.EmployeeID	sub.EmployeeID	via
2	1	Direct Report
2	3	Direct Report
2	4	Direct Report
2	5	Direct Report
2	8	Direct Report
5	6	Direct Report
5	7	Direct Report
5	9	Direct Report
2	6	[5]
2	7	[5]
2	9	[5]

How Many Orders were Made by Each Part of the Hierarchy?

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

e.EmployeeID	reports	totalOrders
2	[1,3,4,5,6,7,9,8]	2155
5	[6,7,9]	568
4	[]	420
1	[]	345
3	[]	321
8	[]	260
7	[]	176
6	[]	168
9	[]	107

Updating the Graph

Now if we wanted to update our graph data, we have to first find the relevant information and then update or extend the graph structures.

Janet is now reporting to Steven

We need to find Steven first, and Janet and her `REPORTS_TO` relationship. Then we remove the existing relationship and create a new one to Steven.

```
MATCH (mgr:Employee {EmployeeID:5})
MATCH (emp:Employee {EmployeeID:3})-[rel:REPORTS_TO]->()
DELETE rel
CREATE (emp)-[:REPORTS_TO]->(mgr)
RETURN *;
```

This single relationship change is all you need to update a part of the organizational hierarchy. All subsequent queries will immediately use the new structure.

- Northwind SQL, CSV and Cypher data files (zip)
- From SQL to Cypher
- Import CSV Guide
- Graph Data Modeling
- Tool: SQL to Neo4j Import