CQL vs SQL Queries

The Northwind database was selected as it was classic RDMS database which has a lot of scope for complex queries. To load the database into neo4j we used their official guide with models. Northwind database loaded from Microsoft's sample database loader script to MS SQL Server.

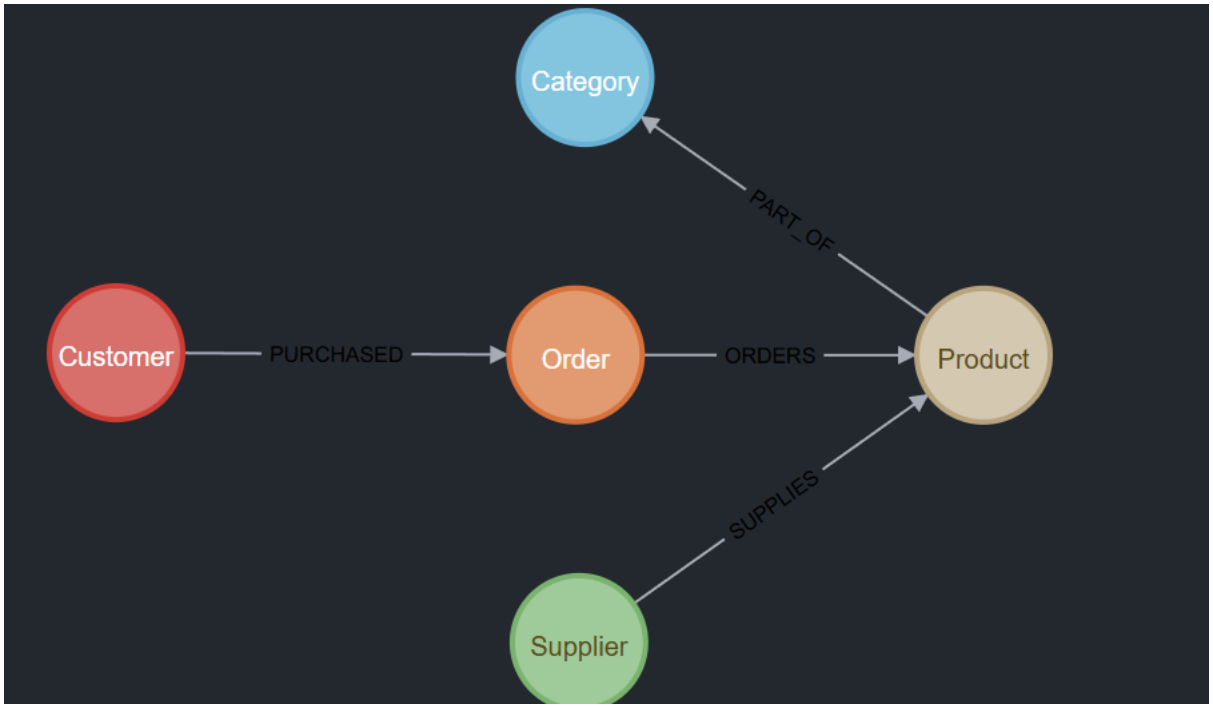

Figure 1 Model of Neo4j Database

The following queries were used to compare the performance and complexity of these two types of databases. CQL has significant performance benefits when it comes to queries which have a lot of joins.

Query	SQL (Relational)	Cypher (Graph)	Performance & Complexity
1. Customer Order History	SELECT O.OrderID, O.OrderDate, O.Freight FROM Customers AS C JOIN Orders AS O ON C.CustomerID = O.CustomerID WHERE C.CustomerID = 'ALFKI';	MATCH (c:Customer {customerID: 'ALFKI'})-[:PURCHASED]->(o:Order) RETURN o.orderID, o.orderDate, o.freight;	Both are efficient; Cypher has a slight edge due to no join overhead.
2. Products in an Order	SELECT P.ProductID, P.ProductName, OD.Quantity, OD.UnitPrice FROM Orders AS O JOIN "Order Details" AS OD ON O.OrderID = OD.OrderID JOIN	MATCH (o:Order {orderID: '10271'})-[:ORDERS]-	Both are efficient. Cypher is more intuitive as

	Products AS P ON OD.ProductID = P.ProductID WHERE O.OrderID = 10271;	>(p:Product) RETURN p.productID, p.productName;	relationships hold properties.
3. Customers who Bought a Product	SELECT DISTINCT C.CustomerID, C.CompanyName FROM Customers AS C JOIN Orders AS O ON C.CustomerID = O.CustomerID JOIN "Order Details" AS OD ON O.OrderID = OD.OrderID WHERE OD.ProductID = 9;	MATCH (p:Product {productID: '9'})<-[:ORDERS]-(:Order)<-[:PURCHASED]-(c:Customer) RETURN DISTINCT c.customerID, c.companyName;	Cypher is more efficient. SQL's multi-join is less performant.
4. Suppliers for a Customer's Products	SELECT DISTINCT S.SupplierID, S.CompanyName FROM Customers AS C JOIN Orders AS O ON C.CustomerID = O.CustomerID JOIN "Order Details" AS OD ON O.OrderID = OD.OrderID JOIN Products AS P ON OD.ProductID = P.ProductID JOIN Suppliers AS S ON P.SupplierID = S.SupplierID WHERE C.CustomerID = 'ALFKI';	MATCH (c:Customer {customerID: 'ALFKI'})-[:PURCHASED]->(:Order)-[:ORDERS]->(p:Product)<-[:SUPPLIES]-(s:Supplier) RETURN DISTINCT s.supplierID, s.companyName;	Graph wins significantly. SQL's multi-hop join is computationally expensive.
5. Top 5 Most Popular Products	SELECT TOP 5 P.ProductName, SUM(OD.Quantity) AS TotalSold FROM Products AS P JOIN "Order Details" AS OD ON P.ProductID = OD.ProductID GROUP BY P.ProductName ORDER BY TotalSold DESC;	MATCH (:Order)-[r:ORDERS]->(p:Product) RETURN p.productName, SUM(r.quantity) AS TotalSold ORDER BY TotalSold DESC LIMIT 5;	Both are well-suited for this. Performance is similar, with a slight edge to SQL for classic aggregation.
6. Customers with Shared Products	SELECT DISTINCT C2.CustomerID, C2.CompanyName FROM Customers AS C1 JOIN Orders AS O1 ON C1.CustomerID = O1.CustomerID JOIN "Order Details" AS OD1 ON O1.OrderID = OD1.OrderID JOIN "Order Details" AS OD2 ON OD1.ProductID = OD2.ProductID JOIN Orders AS O2 ON OD2.OrderID = O2.OrderID JOIN Customers AS C2 ON O2.CustomerID = C2.CustomerID WHERE C1.CustomerID = 'ALFKI' AND C2.CustomerID <> 'ALFKI';	MATCH (c1:Customer {customerID: 'ALFKI'})-[:PURCHASED]->(:Order)-[:ORDERS]->(p:Product)<-[:ORDERS]-(:Order)<-[:PURCHASED]-(c2:Customer) WHERE c1 <> c2 RETURN DISTINCT c2.customerID, c2.companyName;	Graph wins significantly. SQL is complex to write and very slow.
7. Product with Highest Price	SELECT TOP 1 ProductName, UnitPrice FROM Products ORDER BY UnitPrice DESC;	MATCH (p:Product) RETURN p.productName, p.unitPrice ORDER BY p.unitPrice DESC LIMIT 1;	Both are highly efficient for this simple sorting and retrieval.