Show Submission Credentials

P3. Neo4j Primer An Introduction on Graph Database and its Query Language.

16 days 2 hours left

- ✔ Introduction to Graph Databases
- ✔ Cypher Query Language
- ✔ Conclusion

Introduction to Graph Databases

Introduction to Graph Databases

Why Graph Databases?

The massive commercial success of companies such as Facebook, LinkedIn, and Twitter have centered their business models around their own proprietary graphs. For example, Facebook built an empire on the insight to capture these relationships in the social graph.

Graph databases natively embrace relationships to store, process, and query connections efficiently. While other databases compute relationships at query time through expensive and cumbersome JOIN operations, a graph database stores connections alongside the data in the model.

Graph Databases

A graph database is a database designed to treat the relationships between data as equally important to the data itself (without relying on "foreign keys" and "JOIN" operations). The data is stored like drawing a picture showing how each individual entity connects with or is related to others. As a result, the execution time for each query is only proportional to the size of the part of the graph traversed, rather than the size of the overall graph.

Graphs are naturally additive, which means we can add new nodes, new labels, and new relationships to an existing structure without disturbing existing queries and application functionality.

The Property Graph Model

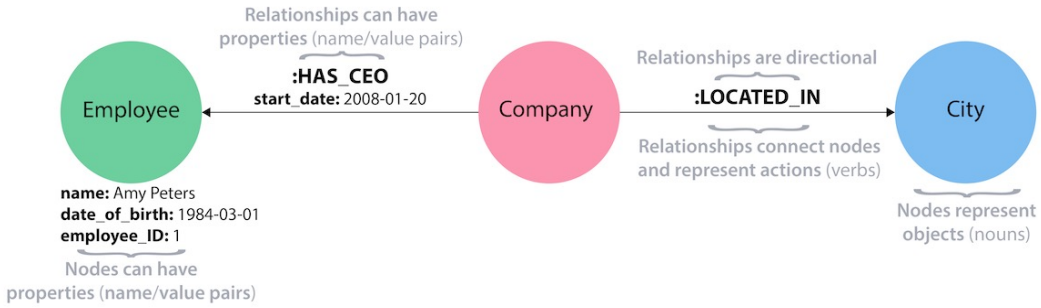

Figure 10: An example of the elements of the property graph model

Neo4j uses the property graph model to organize the data as nodes, relationships, and properties.

Nodes are the entities in the graph. Nodes can be tagged with **labels** to represent their roles in your domain, e.g., `Employee` or `Company` . A node can hold any number of attributes (key-value pairs) called **properties**, e.g., `name` , `date_of_birth` .

Relationships provide directed, named, semantically relevant connections between two node entities (e.g., `Company is LOCATED_IN City`). A relationship always has a direction, a type, a start node, and an end node. Like nodes, relationships can also have properties, e.g., `Company is LOCATED_IN City` since `year:2015` . Due to the efficient way relationships are stored, two nodes can share any number or type of relationships without sacrificing performance. Although they are stored in one direction, relationships can always be navigated efficiently in either direction.

Limitations of Graph Databases

Despite many use cases where graph databases shine, graph databases have their own limitations. Graph databases are not as efficient at processing high volumes of transactions. Besides, graph databases are not good at handling queries that span the entire database.

Cypher Query Language

Cypher Query Language

Cypher is Neo4j's open graph query language. Cypher's syntax provides a familiar way to match patterns of nodes and relationships in the graph.

Introduction

Cypher is a declarative, SQL-inspired language for describing patterns in graphs visually using an ASCII-art syntax. As a declarative language, Cypher allows us to state **what** we want to select, insert, update or delete from our graph data without requiring us to describe exactly **how** to do it.

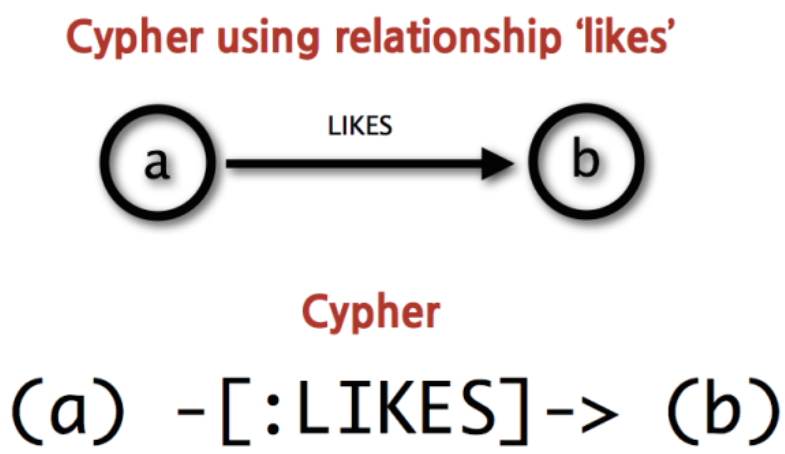

Figure 11: The ASCII-art syntax of Cypher

Nodes

Cypher uses ASCII-Art to represent patterns. We surround nodes with parentheses that look like circles, e.g., `(n)`. For example, you can use expressive variable names like `(user)` or `(post)`.

Usually, the relevant labels of the node are provided to distinguish between entities and to optimize execution, e.g., `(n)` will match any node (e.g., a user node or a post node) but `(post:Post)` will only match the nodes with the label `Post`.

Relationships

Relationships are basically an arrow `-->` between two nodes, e.g., `(user:User)-->(user:User)`, `(user:User)-->(post:Post)`. Additional information can be placed in square brackets inside of the arrow to select the relationships by type, e.g., `(user:User)-[rel:FOLLOWS]->(user:User)` and `(user:User)-[rel:BLOCKS]->(user:User)`.

Please note that node-labels, relationship-types, and property-names are **case-sensitive** in Cypher.

From SQL to Cypher

Cypher is inspired by SQL. Cypher consists of clauses, keywords, and expressions like predicates and functions, many are familiar to SQL (e.g., `WHERE`, `ORDER BY`, `AND`, `p.unitPrice > 10`).

Unlike SQL, Cypher is all about expressing graph **patterns**. `MATCH` clause is used for matching those patterns. These patterns are what you would usually draw on a whiteboard, just converted into text using ASCII-art symbols, e.g., `MATCH (follower:User)-[r:FOLLOWS]->(followee:User)`.

Here we will provide you with some SQL query examples and the equivalent Cypher queries.

Select and Return Records

```
# SQL: Select everything from the users table.
SELECT u.*
FROM users as u;

# Cypher: match a simple pattern: all nodes with the label `User` and RETURN the
m.
MATCH (u:User)
RETURN u;
```

Field Access, Ordering, and Paging

```
# SQL
SELECT p.ProductName, p.UnitPrice
FROM products as p
ORDER BY p.UnitPrice DESC
LIMIT 10;

# Cypher
MATCH (p:Product)
RETURN p.productName, p.unitPrice
ORDER BY p.unitPrice DESC
LIMIT 10;
```

Filter by Equality

```
# SQL
SELECT p.ProductName, p.UnitPrice
FROM products AS p
WHERE p.ProductName = 'Chocolade';

# Cypher
MATCH (p:Product)
WHERE p.productName = "Chocolade"
RETURN p.productName, p.unitPrice;

# There is a shortcut in Cypher if you match for a labeled node with a certain a
ttribute.
MATCH (p:Product {productName:"Chocolade"})
RETURN p.productName, p.unitPrice;
```

Filter by List/Range

```
# SQL
SELECT p.ProductName, p.UnitPrice
FROM products as p
WHERE p.ProductName IN ('Chocolade','Chai');

# Cypher
MATCH (p:Product)
WHERE p.productName IN ['Chocolade','Chai']
RETURN p.productName, p.unitPrice;
```

Filter by Multiple Numeric and Textual Predicates

```
# SQL
SELECT p.ProductName, p.UnitPrice
FROM products AS p
WHERE p.ProductName LIKE 'C%' AND p.UnitPrice > 100;

# Cypher
MATCH (p:Product)
WHERE p.productName STARTS WITH "C" AND p.unitPrice > 100
RETURN p.productName, p.unitPrice;
```

Query by Connected Records

```
# SQL
SELECT DISTINCT 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)
JOIN products AS p ON (od.ProductID = p.ProductID)
WHERE p.ProductName = 'Chocolade';

# Cypher
MATCH (p:Product {productName:"Chocolade"})<-[:PRODUCT]-(:Order)<-[:PURCHASED]-
(c:Customer)
RETURN distinct c.companyName;
```

Indexes

A database index is a redundant copy of some of the data in the database to improve the efficiency of search-related data. This comes at the cost of additional storage space and slower writes.

Similar to SQL, Cypher enables the creation of indexes on one or more properties for all nodes that have a given label.

- An index that is created on a single property for any given label is called a single-property index.
- An index that is created on more than one property for any given label is called a composite index.

Query Examples

```
# SQL
CREATE INDEX person_single_property_index ON person (firstname)
CREATE INDEX person_composite_index ON person (firstname, lastname)

# Cypher
CREATE INDEX ON :Person(firstname)
CREATE INDEX ON :Person(firstname, lastname)
```

Unique Constraint

In SQL, you can add a constraint to enforce the uniqueness of the values in a column or a group of columns. Similarly, you can enforce the uniqueness of a specific label with one property or a group of properties.

```
# SQL
ALTER TABLE user ADD CONSTRAINT user_unique_constraint UNIQUE (username)

# Cypher
CREATE CONSTRAINT ON (user:User) ASSERT user.username IS UNIQUE
```

With both SQL and Cypher, adding a unique property constraint on a property will also implicitly add a single-column/single-property index, so there is no need to index the column/property separately again.

The linguistic efficiency of Cypher

SQL runs up against major performance challenges when it tries to navigate connected data. For data-relationship questions, a single query in SQL can be many lines longer than the same query in a graph database query language like Cypher.

Lengthy SQL queries not only take more time to run, but they are also more likely to include human coding mistakes because of their complexity. In addition, shorter queries increase the ease of understanding and maintenance across your team of developers.

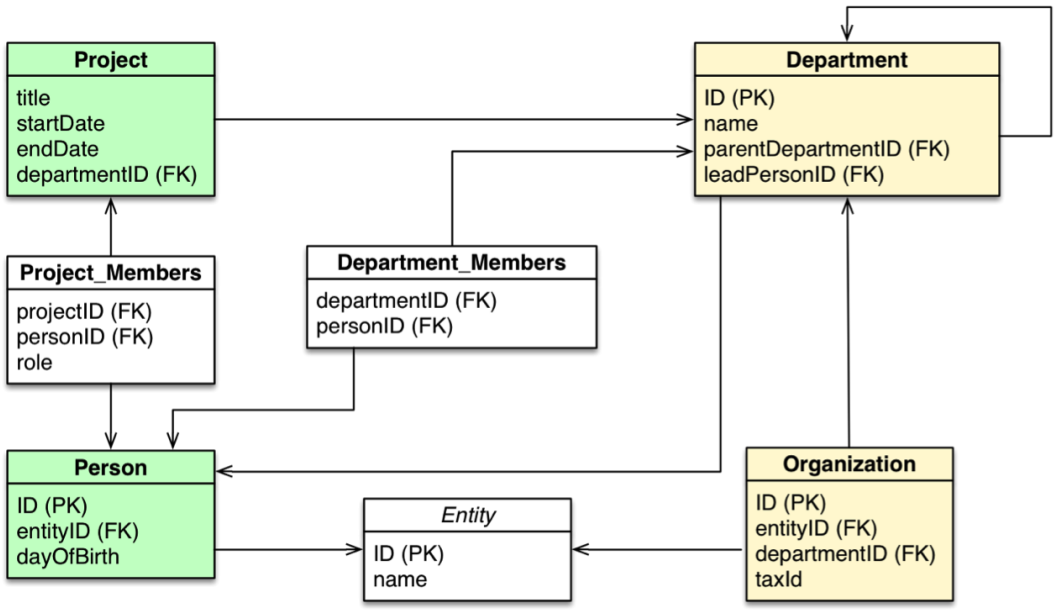

Figure 12: A relational data model of an organizational domain

In the organizational domain depicted in the model above, what would a SQL statement that lists the employees in the IT Department look like? And how does that statement compare to a Cypher statement?

```
# SQL query
# To maintain a many-to-many mapping of Department and Person,
# you have to rely on a table named as `Department_Members`.
# To search the employees given a department name,
# you have to join three tables together (Person, Department, Department_Member
s).
SELECT name FROM Person

LEFT JOIN Department_Members

  ON Person.Id = Department_Members.PersonId

LEFT JOIN Department

  ON Department.Id = Department_Members.DepartmentId

WHERE Department.name = "IT Department"

# Cypher statement
MATCH (p:Person)<-[:EMPLOYS]-(d:Department)
WHERE d.name = "IT Department"
RETURN p.name
```

When many-to-many relationships occur in the model with SQL, you must introduce a JOIN table that holds foreign keys of both the participating tables, further increasing join operation costs. The figure below shows this concept of connecting a Person (from Person table) to a Department (in Department table) by creating a Person-Department join table that contains the ID of the person in one column and the ID of the associated department in the next column.

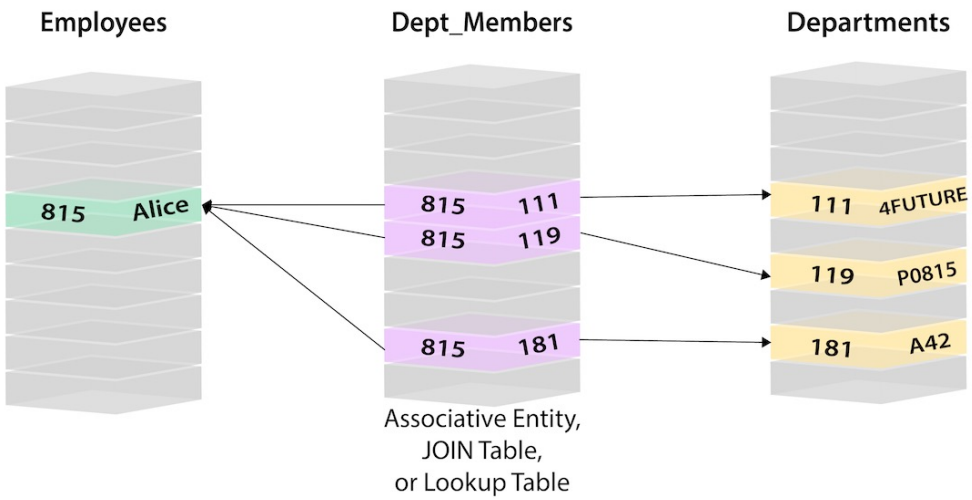

Figure 13: Relational model in a SQL database

The connections are very cumbersome because you must know the person ID and department ID values which require additional lookups in order to know which person connects to which departments. Here are the steps for SQL to query the departments that an employee `Alice` works for.

1. Search the `Employees` table to get the employee ID of `Alice`, which is `815`.
2. Search `815` in the `Dept_Members` table to find the "connected" department IDs (`111`, `119`, `181`).
3. Search `111`, `119`, `181` in the `Departments` table to find the department names.

Hence, in SQL, traversing the links are expensive using JOIN operations. Even with the usage of indexes (at a cost of space and write performance penalty), each hop still requires an index lookup.

Those types of costly join operations are often tempered by denormalizing the data to reduce the number of joins needed, e.g., by duplicating the data such as create an all-in-one table that contains columns such as `Employee Name` and `Department Name` at the same time, at a cost of breaking the data integrity of a relational database. This will not solve all the connection queries either, e.g., it is still hard (or even harder) to find if `Alice` and `Bob` are working in the same department.

With a Graph database, one single hop with no lookups will solve the problem.

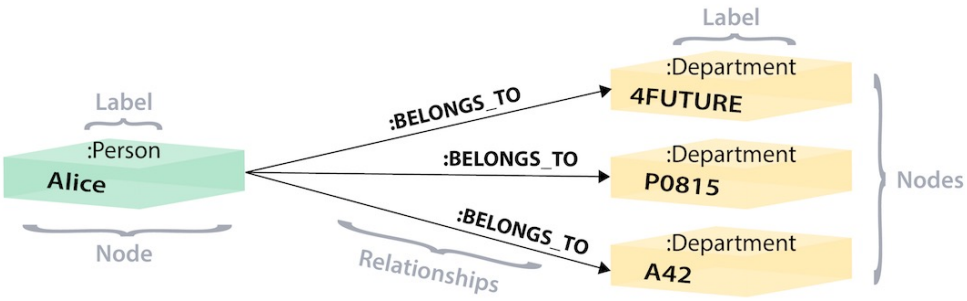

Figure 14: Relational Model in a graph database

The node `Alice`, at the storage level, stores the locations of the 3 connected `Department` nodes, and this query only traverses 4 nodes in total, regardless of the size of the whole graph.

Here is one more example. For a customer named `John Doe`, check all the products `John Doe` has bought, then find all the peer customers who bought any of these products and list all the other products that the peer customers bought (that have not been bought by `John Doe`) as the product recommendation to `John Doe`.

```
# Cypher query
MATCH (u:Customer {customer_id:'John Doe'})-[:BOUGHT]->(p:Product)<-[:BOUGHT]-(p
eer:Customer)-[:BOUGHT]->(recommendation:Product)
WHERE not (u)-[:BOUGHT]->(recommendation)
RETURN recommendation as Recommendation, count(*) as Frequency
ORDER BY Frequency DESC LIMIT 5;
```

Here is the equivalent SQL query which suffers from bad performance. You will find the query extremely hard to read not to mention the difficulty to maintain it.

```
SELECT product.product_name as Recommendation, count(1) as Frequency
FROM product, customer_product_mapping, (SELECT cpm3.product_id, cpm3.customer_i
d
FROM Customer_product_mapping cpm, Customer_product_mapping cpm2, Customer_produ
ct_mapping cpm3
WHERE cpm.customer_id = 'John Doe'
and cpm.product_id = cpm2.product_id
and cpm2.customer_id != 'John Doe'
and cpm3.customer_id = cpm2.customer_id
and cpm3.product_id not in (select distinct product_id
FROM Customer_product_mapping cpm
WHERE cpm.customer_id = 'John Doe')
) recommended_products
WHERE customer_product_mapping.product_id = product.product_id
and customer_product_mapping.product_id in recommended_products.product_id
and customer_product_mapping.customer_id = recommended_products.customer_id
GROUP BY product.product_name
ORDER BY Frequency desc
```

It is a very common problem in the real world when you need to build complex connected data query with RDBMS. You will easily end up with these lengthy SQL queries. As a solution, you want to treat connectedness as a first-class citizen and adopt a Graph database.

Conclusion

We hope that this primer has introduced you to the basics on how to visualize the data and transfer the insight into good feature engineering.

Reference

- Neo4j - What is a Graph Database? (https://neo4j.com/developer/graph-database/)
- Neo4j - Intro to Cypher (https://neo4j.com/developer/cypher-query-language/)
- Neo4j - Concepts: Relational to Graph (https://neo4j.com/developer/graph-db-vs-rdbms/)
- From SQL to Cypher – A hands-on Guide (https://neo4j.com/developer/guide-sql-to-cypher/)
- 3.5.1. Indexes – The Neo4j Developer Manual (https://neo4j.com/docs/developer-manual/current/cypher/schema/index/)
- Neo4j Blog - RDBMS & Graphs: SQL vs. Cypher Query Languages By Michael Hunger, Ryan Boyd & William Lyon (https://neo4j.com/blog/sql-vs-cypher-query-languages/)
- Graph Databases By Ian Robinson, Jim Webber and Emil Eifrem (https://neo4j.com/graph-databases-book-sx/)