neo4j-graph-modelling

September 3, 2024

[ ]:

##What is Graph Data Modeling? ##**Why model?** If you use a Neo4j graph to support part or all of your application, you must collaboratively work with your stakeholders to design a graph that will:

Answer the key use cases for the application.

0.1 Provide the best Cypher statement performance for the key use cases.

##**Components of a Neo4j graph** The Neo4j components that are used to define the graph data model are:

1. Nodes
2. Labels
3. Relationships
4.

0.2 Properties

##**Data modeling process** Here are the steps to create a graph data model:

Understand the domain and define specific use cases (questions) for the application.

1. Develop the initial graph data model:
2. Model the nodes (entities).
3. Model the relationships between nodes.
4. Test the use cases against the initial data model.
5. Create the graph (instance model) with test data using Cypher.
6. Test the use cases, including performance against the graph.
7. Refactor (improve) the graph data model due to a change in the key use cases or for performance reasons.
8. Implement the refactoring on the graph and retest using Cypher.

Graph data modeling is an **iterative process.** Your initial graph data model is a starting point, but as you learn more about the use cases or if the use cases change, the initial graph data model

will need to change. In addition, you may find that especially **when the graph scales, you will need to modify the graph (refactor) to achieve the best performance for your key use cases.**

Refactoring is very common in the development process. A Neo4j graph has an **optional schema which is quite flexible, unlike the schema in an RDBMS.** A Cypher developer can easily modify the graph to represent an improved data model.

##1. Neo4j elements for graph data modeling What Neo4j elements must you understand to begin the graph data modeling process?

[ ]:
```
Nodes, Labels, Properties, Relationships
```

##2. Iterative modeling What term do we use to define how we will improve an existing graph data model?

[ ]:
```
The term we use for evolving a data model is **Refactoring.** In order to␣
↪optimize your graph model for specific use cases you must Refactor it.
```

##Understanding the domain for your application Before you begin the data modeling process you must:

1. Identify the stakeholders and developers of the application.
2. With the stakeholders and developers:
3. Describe the application in detail.
4. Identify the users of the application (people, systems).
5. Agree upon the use cases for the application.
6. Rank the importance of the use cases.

##1. Understanding the domain What are some ways that stakeholders prepare for modeling?

##Solution

Stakeholders can prepare for graph data modeling by doing the following:

1. Describe the domain in detail.
2. Identify the systems and users of the application.
3. Enumerate the use cases of the application.

##Types of models When performing the graph data modeling process for an application, you will need at least two types of models:

1. Data model
2. Instance model

##Data model The data model describes the labels, relationships, and properties for the graph. It does not have specific data that will be created in the graph.

Here is an example of a data model:

There is nothing that **uniquely identifies a node with a given label.** A graph data model, however is important because **it defines the names that will be used for labels, relationship types, and properties when the graph is created and used by the application.**

##Style guidelines for modeling As you begin the graph data modeling process, it is important that you agree upon how labels, relationship types, and property keys are named. Labels, relationship types, and property keys are case-sensitive, unlike Cypher keywords which are case-insensitive.

A Neo4j best practice is to use the following when you name the elements of the graph, but you are free to use any convention for your application.

1. A label is a single identifier that begins with a **capital letter and can be CamelCase.**

Examples: Person, Company, GitHubRepo

2. A relationship type is a single identifier that is in **all capital letters with the underscore character.**

Examples: FOLLOWS, MARRIED_TO

3. A property key for a node or a relationship is a **single identifier that begins with a lower-case letter and can be camelCase.**

Examples: deptId, firstName

####Note: Property key names need not be unique. For example, a Person node and a Movie node, each can have the property key of tmdbId.

##Instance model An important part of the graph data modeling process is to **test the model against the use cases.** To do this, you need to have a set of sample data that you can use to see if the use cases can be answered with the model.

Here is an example of an instance model:

####In this instance model, we have created some instances of Person and Movie nodes, as well as their relationships. Having this type of instance model will help us to test our use cases.

##1. Purpose of models Why do we create an instance model during our modeling process?

####**Ans:To help us confirm that our data model can satisfy use cases of the application.**

##Defining labels

In the Movie domain, we use the nouns in our use cases to define the labels, for example:

1. What people acted in a movie?
2. What person directed a movie?
3. What movies did a person act in?

Here are some of the labeled nodes that we will start with.

Notice here that we use CamelCase for the names for labels.

##Node properties Node properties are used to:

1. Uniquely identify a node.
2. Answer specific details of the use cases for the application.
3. Return data.

##For example, in a Cypher statement, properties are used to:

1.Anchor (where to begin the query).

##MATCH (p:Person {name: 'Tom Hanks'})-[:ACTED_IN]-(m:Movie) RETURN m

2. Traverse the graph (navigation).

##MATCH (p:Person)-[:ACTED_IN]-(m:Movie {title: 'Apollo 13'})-[:RATED]-(u:User) RETURN p,u

3. Return data from the query.

##MATCH (p:Person {name: 'Tom Hanks'})-[:ACTED_IN]-(m:Movie) RETURN m.title, m.released

##Unique identifiers in the Movie graph In the Movie graph, we use the following properties to uniquely identify our nodes:

1. **Person.tmdbId**
2. **Movie.tmdbId**

##Properties for nodes In addition to the tmdbId that is used to uniquely identify a node, we must revisit the use cases to determine the types of data a node must hold.

Here is a list of our use cases specific to Person and Movie nodes that we will focus on. These use cases inform us about the data we need in Movie and Person nodes.

Given the details of the steps of these use cases, here are the properties we will define for the Movie nodes:

1. Movie.title (string)
2. Movie.released (date)
3. Movie.imdbRating (decimal between 0-10)
4. Movie.genres (list of strings)

Here are the properties we will define for the Person nodes:

1. Person.name (string)
2. Person.born (date)
3. Person.died (date)

####Note: The died property will be optional.

Here is the initial data model:

###And here is the initial **instance model** you will be creating:

##1. Labels Suppose you have created a list use cases for the application. What elements of the use cases are used to define labels the data model?

[ ]:
```
Nouns

Nouns represent the things or entities in your use cases.
```

##2. Defining node properties What can node properties be used for? Select the three correct answers below:

[ ]:
```
#Node properties can be used for:

Uniquely identifying a node.

Answering specific details of the use cases.

Returning data.
```

##Creating Nodes Here is the initial instance model you will be working with:

Run the Cypher code below to add the Person and Movie nodes to the graph which will serve as our initial instance model:

[ ]:
```
MATCH (n) DETACH DELETE n;

MERGE (:Movie {title: 'Apollo 13', tmdbId: 568, released: '1995-06-30',␣
 ↪imdbRating: 7.6, genres: ['Drama', 'Adventure', 'IMAX']})
MERGE (:Person {name: 'Tom Hanks', tmdbId: 31, born: '1956-07-09'})
MERGE (:Person {name: 'Meg Ryan', tmdbId: 5344, born: '1961-11-19'})
MERGE (:Person {name: 'Danny DeVito', tmdbId: 518, born: '1944-11-17'})
MERGE (:Person {name: 'Jack Nicholson', tmdbId: 514, born: '1937-04-22'})
MERGE (:Movie {title: 'Sleepless in Seattle', tmdbId: 858, released:␣
 ↪'1993-06-25', imdbRating: 6.8, genres: ['Comedy', 'Drama', 'Romance']})
MERGE (:Movie {title: 'Hoffa', tmdbId: 10410, released: '1992-12-25',␣
 ↪imdbRating: 6.6, genres: ['Crime', 'Drama']})
```

###Notice that in this code we are using the Neo4j best practice guidelines for naming labels (CamelCase) and properties (camelCase).

You can verify that the nodes have been created by running this code:

[ ]:
```
MATCH (n) RETURN n
```

[ ]:
```
MATCH (n) DETACH DELETE n;
```

[ ]:
```
MERGE (:Movie {title: 'Apollo 13', tmdbId: 568, released: '1995-06-30',␣
 ↪imdbRating: 7.6, genres: ['Drama', 'Adventure', 'IMAX']})
MERGE (:Person {name: 'Tom Hanks', tmdbId: 31, born: '1956-07-09'})
MERGE (:Person {name: 'Meg Ryan', tmdbId: 5344, born: '1961-11-19'})
MERGE (:Person {name: 'Danny DeVito', tmdbId: 518, born: '1944-11-17'})
MERGE (:Person {name: 'Jack Nicholson', tmdbId: 514, born: '1937-04-22'})
MERGE (:Movie {title: 'Sleepless in Seattle', tmdbId: 858, released:␣
 ↪'1993-06-25', imdbRating: 6.8, genres: ['Comedy', 'Drama', 'Romance']})
MERGE (:Movie {title: 'Hoffa', tmdbId: 10410, released: '1992-12-25',␣
 ↪imdbRating: 6.6, genres: ['Crime', 'Drama']})
```

[ ]:
```
MATCH (n) RETURN count(n)
```

##Adding a new label to the model What label should be added to nodes to identify any Users who have rated a movie?

[ ]:
```
The answer is User.

user is in the wrong case. Users is plural. Person is already used for actors␣
 ↪and directors.
```

##Creating More Nodes We want to add a couple of User nodes to the graph so we can test the changes to our model.

Any User node will have the following properties:

1. userId - an integer (eg. 123)
2. name - a string (eg. User's Name)

Use the sandbox window to the right to create two User nodes for:

1. 'Sandy Jones' with the userId of 534
2. 'Clinton Spencer' with the userId of 105

[ ]:
```
MERGE (sandy:User {userId: 534}) SET sandy.name = "Sandy Jones"
MERGE (clinton:User {userId: 105}) SET clinton.name = "Clinton Spencer"
```

##Relationships are connections between entities Connections are the verbs in your use cases:

What ingredients are **used** in a recipe?

Who is **married** to this person?

##Naming relationships Choosing good names (types) for the relationships in the graph is important. Relationship types need to be something that is intuitive to stakeholders and developers alike. Relationship types cannot be confused with an entity name.

So in our example use cases, we could define these relationship types:

1. USES
2. MARRIED

Note here that we use the Neo4j best practice of all capital letters/underscore characters for the name of the relationship.

##Relationship direction When you create a relationship in Neo4j, a direction must either be specified explicitly or inferred by the **left-to-right** direction in the pattern specified. **At runtime, during a query, direction is typically not required.**

In our example use cases shown above, the **USES** relationship must be created to **go from a Recipe node to an Ingredient node.**

The **MARRIED** relationship could be created to start in either node since this type of relationship is symmetric.

###**A relationship is typically between 2 different nodes, but it can also be to the same node.**

##Fanout

Here, we have entities **(Person, Residence)** represented **not as a single node,** but as a **network or linked nodes.**

This is an extreme example of **fanout,** and is almost certainly **overkill for any real-life solution,** but **some amount of fanout can be very useful.**

For example, splitting last names onto separate nodes helps answer the question, "Who has the last name Scott?". Similarly, having cities as separate nodes assists with the question, "Who lives in the same city as Patrick Scott?".

The **main risk** about fanout is that it can lead to very **dense nodes, or supernodes.** These are nodes that have **hundreds of thousands of incoming or outgoing relationships** Supernodes need to be handled carefully.

##Relationships in the Movie graph Now let's look at identifying the relationships for these use cases:

1. What people **acted** in a movie?
2. What person **directed** a movie?
3. What movies did a person **act** in?

Given these use cases, we name the relationships:

1. ACTED_IN
2. DIRECTED

Furthermore, both of these relationship types start at Person nodes and end in Movie nodes.

Here is the supporting graph data model:

And here is the **instance model** to support this **graph data model:**

Tom Hanks acted in two movies. Meg Ryan and Jack Nicholson each acted in one movie. Danny DeVito both acted in and directed the same movie. Exploring relationships with this instance model we see that the movie Apollo 13 has a single actor in the graph, but the other two movies have two actors each.

##Properties for relationships Properties for a relationship are used to enrich **how two nodes are related.** When you define a property for a relationship, it is because your use cases ask a **specific question about how two nodes are related,** not just that they are related.

For example, we saw in the Neo4j Fundamentals course that properties can be added to a relationship to further describe it.

Here we see that we have a date property on the MARRIED relationship to further describe the relationship between Michael and Sarah. Additionally, we have a roles property on the WORKS_AT relationship to describe the roles that Michael has or had when he worked at Graph Inc.

These properties are specific to the relationship between two nodes.

##Relationship properties in the Movie graph Just like you analyze the use cases for naming labels, relationship types, and node properties, you use the use cases to come up with properties for relationships.

Here is a use case:

6. What role did a person play in a movie?

The runtime operations for this use case are:

1. Retrieve the name of the person.
2. Follow the ACTED_IN relationships to movies.

3. Filter the movie by its title.
4. Return the role from the ACTED_IN relationship between the two nodes.

We know that the role for a particular ACTED_IN relationship will be necessary for this use case. So we add the role property to this relationship. Here is the data model:

And here is the instance model you will be creating:

Each ACTED_IN relationship here has a different value for the **role** property.

##2. Defining relationships When you define relationships in the graph data model, what must you define?

[ ]:
```
A relationship must have a type and direction when it is created in the graph.
↪There are three correct answers to this question.

You must specify the following when you define a relationship in the graph:

1. The starting node (with label) for the relationship.

2. The ending node (with label) for the relationship.

3. The name (type) for the relationship.
```

##1. Connections between entities Suppose you have created a list of use cases for the application. What elements of the use cases are used to define the relationships in the data model?

[ ]:
```
Verbs in your use cases are used to represent the connections between entities
↪in the graph.
```

##Creating Initial Relationships Here is the instance model you will be creating:

Each **ACTED_IN** relationship here has a different value for the role property.

Run this Cypher code to **add the ACTED_IN and DIRECTED relationships to the graph:**

[ ]:
```
MATCH (apollo:Movie {title: 'Apollo 13'})
MATCH (tom:Person {name: 'Tom Hanks'})
MATCH (meg:Person {name: 'Meg Ryan'})
MATCH (danny:Person {name: 'Danny DeVito'})
MATCH (sleep:Movie {title: 'Sleepless in Seattle'})
MATCH (hoffa:Movie {title: 'Hoffa'})
MATCH (jack:Person {name: 'Jack Nicholson'})
```

```
// create the relationships between nodes
MERGE (tom)-[:ACTED_IN {role: 'Jim Lovell'}]->(apollo)
MERGE (tom)-[:ACTED_IN {role: 'Sam Baldwin'}]->(sleep)
MERGE (meg)-[:ACTED_IN {role: 'Annie Reed'}]->(sleep)
MERGE (danny)-[:ACTED_IN {role: 'Bobby Ciaro'}]->(hoffa)
MERGE (danny)-[:DIRECTED]->(hoffa)
MERGE (jack)-[:ACTED_IN {role: 'Jimmy Hoffa'}]->(hoffa)
```

You can verify that the relationships have been created with this code:

[ ]:
```
MATCH (n) RETURN n
```

##1. Adding a new relationship to the model What Relationship Type could you use when creating a relationship between User nodes and Movie to represent a rating?

The answer here is **RATED.**

rated is in the wrong case and **SCORE** and **RATING** are nouns rather than verbs.

[ ]:
```
The answer here is RATED.

rated is in the wrong case and SCORE and RATING are nouns rather than verbs.
```

##2. Relationship Properties What properties could you add to the relationship to support this use case? (Select any that apply)

You could add a **rating** property to the relationship to store the user's rating of the movie. You format properties as lowerCamelCase.

The relationship **does not** need **userId** or **tmdbId** properties to identify the start and end nodes.

[ ]:
```
You could add a rating property to the relationship to store the user's rating␣
↪of the movie. You format properties as lowerCamelCase.

The relationship does not need userId or tmdbId properties to identify the␣
↪start and end nodes.
```

In this challenge, you demonstrated your skills in identifying the connections between the entities of the domain and defining the relationships for the graph data model. In the next challenge, you will create new relationships in the graph for our instance model.

##Creating More Relationships We want to add some relationships between User nodes and Movie nodes so we can test our model.

In this challenge, you will create RATED relationships that include the rating property.

The sandbox window on the right creates one relationship between Sandy Jones and Apollo 13 with a rating of 5. Add additional MERGE code in the query edit pane to create the remaining 4

relationships per the table below:

####Use **MATCH** to find the User and Movie nodes,

####then use **MERGE** to create the relationships between the two nodes.

####Remember that you must **specify or infer(left-to-right) a direction** when you create a relationship. You should create a total of 5 RATED relationships in the graph, each with a property, rating.

[ ]:
```
MATCH (sandy:User {name: 'Sandy Jones'})
MATCH (clinton:User {name: 'Clinton Spencer'})
MATCH (apollo:Movie {title: 'Apollo 13'})
MATCH (sleep:Movie {title: 'Sleepless in Seattle'})
MATCH (hoffa:Movie {title: 'Hoffa'})
MERGE (sandy)-[:RATED {rating:5}]->(apollo)
MERGE (sandy)-[:RATED {rating:4}]->(sleep)
MERGE (clinton)-[:RATED {rating:3}]->(apollo)
MERGE (clinton)-[:RATED {rating:3}]->(sleep)
MERGE (clinton)-[:RATED {rating:3}]->(hoffa)
```

##1. Outcomes of testing During your testing of the use cases, what might you need to do?

[ ]:
```
Add more data to the graph to test scalability.

 Test and modify any Cypher code used to test the use cases.

 Refactor the data model if a use case cannot be answered.
```

[ ]:
```
#Proper use of Cypher Code.

#Scalability: If graph grows, how your Cypher code retrive and behave.

#Refactor as per need.
```

##Use case #1: What people acted in a movie? Run this Cypher code to test this use case using the movie, Sleepless in Seattle.

[ ]:
```
MATCH (p:Person)-[:ACTED_IN]-(m:Movie)
WHERE m.title = 'Sleepless in Seattle'
RETURN p.name AS Actor
```

##Use case #2: What person directed a movie? Run this Cypher code to test this use case using the movie, Hoffa.

[ ]:
```
MATCH (p:Person)-[:DIRECTED]-(m:Movie)
WHERE m.title = 'Hoffa'
RETURN  p.name AS Director
```

##Use case #3: What movies did a person act in? Run this Cypher code to test this use case using the person, Tom Hanks.

[ ]:
```
MATCH (p:Person)-[:ACTED_IN]-(m:Movie)
WHERE p.name = 'Tom Hanks'
RETURN m.title AS Movie
```

##Use case #4: How many users rated a movie? Run this Cypher code to test this use case using the movie, Apollo 13.

[ ]:
```
MATCH (u:User)-[:RATED]-(m:Movie)
WHERE m.title = 'Apollo 13'
RETURN count(*) AS `Number of reviewers`
```

##Use case #5: Who was the youngest person to act in a movie? Run this Cypher code to test this use case using the movie, Hoffa.

[ ]:
```
MATCH (p:Person)-[:ACTED_IN]-(m:Movie)
WHERE m.title = 'Hoffa'
RETURN  p.name AS Actor, p.born as `Year Born` ORDER BY p.born DESC LIMIT 1
```

##Use case #6: What role did a person play in a movie? Run this Cypher code to test this use case using the movie, Sleepless in Seattle and the person, Meg Ryan.

[ ]:
```
MATCH (p:Person)-[r:ACTED_IN]-(m:Movie)
WHERE m.title = 'Sleepless in Seattle' AND
p.name = 'Meg Ryan'
RETURN  r.role AS Role
```

##Use case #7: What is the highest rated movie in a particular year according to imDB? Run this Cypher code to test this use case using movies in the year 1995

[ ]:
```
MATCH (m:Movie)
WHERE m.released STARTS WITH '1995'
RETURN  m.title as Movie, m.imdbRating as Rating ORDER BY m.imdbRating DESC␣
↪LIMIT 1
```

Run the Cypher code to add another Movie node and its director to the graph:

[ ]:
```
MERGE (casino:Movie {title: 'Casino', tmdbId: 524, released: '1995-11-22',␣
↪imdbRating: 8.2, genres: ['Drama','Crime']})
MERGE (martin:Person {name: 'Martin Scorsese', tmdbId: 1032})
MERGE (martin)-[:DIRECTED]->(casino)
```

##Use case #8: What drama movies did an actor act in? Run this Cypher code to test this use case using the person, Tom Hanks.

[ ]:
```
MATCH (p:Person)-[:ACTED_IN]-(m:Movie)
WHERE p.name = 'Tom Hanks' AND
'Drama' IN m.genres
RETURN m.title AS Movie
```

##Use case #9: What users gave a movie a rating of 5? Run this Cypher code to test this use case using the movie, Apollo 13.

[ ]:
```
MATCH (u:User)-[r:RATED]-(m:Movie)
WHERE m.title = 'Apollo 13' AND
r.rating = 5
RETURN u.name as Reviewer
```

MATCH (n) RETURN n

##1. Person Node Count How many Person nodes are in the graph?

[ ]:
```
MATCH (:Person) RETURN count(*)
```

##2. Movie Node Count How many Movie nodes are in the graph?

[ ]:
```
MATCH (:Movie) RETURN count(*)
```

##3. User Node Count How many User nodes are in the graph?

[ ]:
```
MATCH (:User) RETURN count(*)
```

##4. Acted In Relationship Count How many ACTED_IN relationships are in the graph?

[ ]:
```
MATCH ()-[:ACTED_IN]->() RETURN count(*)
```

##5. Directed Relationship Count How many DIRECTED relationships are in the graph?

[ ]:
```
MATCH ()-[:DIRECTED]->() RETURN count(*)
```

##6. Ratings Count How many RATED relationships are in the graph?

[ ]:
```
MATCH ()-[:RATED]->() RETURN count(*)
```

##Refactoring the Graph

###Why refactor? Refactoring is the process of **changing the data model and the graph.**

There are three reasons why you would refactor:

1. **The graph as modeled does not answer all of the use cases.**
2. **A new use case has come up that you must account for in your data model.**

3. **The Cypher for the use cases does not perform optimally, especially when the graph scales.**

##Steps for refactoring To refactor a graph data model and a graph, you must:

1. Design the new data model.
2. Write Cypher code to transform the existing graph to implement the new data model.
3. Retest all use cases, possibly with updated Cypher code.

##1. Why refactor? Why do you refactor a graph data model and graph?

[ ]:
```
You may want to refactor:----------------->

1. Any of the use cases cannot be answered by the graph.
2. Another use case has been created that needs to be accounted for.
3. The data model does not scale.

if any of the use cases cannot be answered, or if another use case has been␣
 ↪added.
It it also important to ensure that the data model scales effectively and still␣
 ↪provides data in a timely fashion.

As long as the data model stays the same, you do not need to refactor the graph␣
 ↪specifically when updating or adding new data.
```

##Labels at runtime Node labels serve as an **anchor point for a query.** By specifying **a label,** we are specifying **a subset of one or more nodes with which to start a query.** Using a label helps to **reduce the amount of data that is retrieved.**

For example:

###**MATCH (n) RETURN n** returns all nodes in the graph.

###**MATCH (n:Person) RETURN n** returns all Person nodes in the graph.

Your goal in modeling should be to **reduce the size of the graph** that is touched by a query.

In Cypher, you can produce a query plan that shows what operations occur during the query. This figure **shows a query plan by the number of db hits for the query to retrieve all Person nodes:**

If Person nodes also had a label which is the country that a person is from, then you could used this Cypher code to retrieve all people from the US:

###**MATCH (n:US) RETURN n** returns all US nodes in the graph which happen to be Person nodes.

But having **a label** that is specific like this might be overkill, especially if the query could be:

###**MATCH (n:Person) WHERE n.country = 'US' RETURN n**

In Cypher, you **cannot** parameterize labels so keeping the country as a **property** makes the Cypher code more flexible.

But if you have a strong use case for having **multiple labels for a node,** you should do so.

##Do not overuse labels You should use labels wisely in your data model. They should be used if it will help with most of your use cases. A best practice is **to limit the number of labels for a node to 4.**

Here is an example of **overuse of labels** in the data model:

Here we see **Person nodes** that have **a label** representing the **country** that a Person is from like we described earlier.

In addition, we see **multiple labels for Movie nodes.** The label represents the languages available for a movie.

This is another similar scenario where you must decide if an important use case is related to the language of a movie.

###Again if the **use of a property** for a node will **suffice**, then it is best to **not have the label.**

##New use case Here is an example where **adding a label** will help our **queries at runtime.**

What if we added a new use case:

###**Use case #10: What actors were born before 1950?**

Here is the Cypher statement to test this use case:

[ ]:
```
MATCH (p:Person)-[:ACTED_IN]-()
WHERE p.born < '1950'
RETURN p.name
```

Here is what this Cypher statement does:

1. A node by label scan to retrieve all Person nodes.
2. Tests the born property for the nodes retrieved to filter them.
3. Determines which of these filtered nodes have the outgoing ACTED_IN relationship.
4. Returns the name property values.

##Profiling a query You can use the **PROFILE** keyword to see the **performance for a query.**

[ ]:
```
PROFILE MATCH (p:Person)-[:ACTED_IN]-()
WHERE p.born < '1950'
RETURN p.name
```

This is the result of the profile:

Because the cache is automatically populated, it is sometimes hard to measure performance with a small dataset. That is, db hits and elapsed time may not be comparable. What you can see, however, is the number of rows that are retrieved in the query and this number can be compared.

In the first step of this query, we see that 5 rows are returned. You can imagine that if this were a fully-loaded graph with millions of nodes, in step 1, it would need to retrieve a lot of Person nodes, some of which are not actors. One way that you can optimize this retrieval is to change the data model to include an Actor label for a Person node.

##Refactoring the model If we refactor, the initial node by label scan would only retrieve the Actor nodes.

Here is the refactored instance model we will create in the graph:

##Refactor the graph With Cypher, you can easily transform the graph. With this code, that you will execute in the next Challenge, we **find all Person nodes that have an ACTED_IN relationship.** We then **set a label for the node.**

[ ]:
```
MATCH (p:Person)
WHERE exists ((p)-[:ACTED_IN]-())
SET p:Actor
```

##1. Why add labels? What is the primary reason why you would add labels to nodes?

[ ]:
```
To reduce the number of data accessed at runtime.
```

##2. Number of labels As a best practice, what is the maximum number of labels a node should have?

[ ]:
```
4
```

##Adding the Actor Label Here is the refactored instance model we will create in the graph where we add an Actor label to some of the Person nodes:

###Profile the query

[ ]:
```
PROFILE MATCH (p:Person)-[:ACTED_IN]-()
WHERE p.born < '1950'
RETURN p.name
```

In the first step of this query, we see that 5 Person rows are returned.

###Refactor the graph With Cypher, you can easily transform the graph to add Actor labels.

Execute this Cypher code to add the Actor label to the appropriate nodes:

[ ]:
```
MATCH (p:Person)
WHERE exists ((p)-[:ACTED_IN]-())
SET p:Actor
```

There are 5 Person nodes in the graph, but only 4 have an :ACTED_IN relationship. Therefore, the query above should apply the Actor label to four of the five Person nodes.

##Profile the query Now that we have refactored the graph, we must change our query and profile again.

[ ]:
```
PROFILE MATCH (p:Actor)-[:ACTED_IN]-()
WHERE p.born < '1950'
RETURN p.name
```

In the first step of this query, we see that 4 Actor rows are returned.

##Retesting After Refactoring After you have refactored the graph, you should **revisit all queries** for your use cases.

You should first determine if any of the queries need to be **rewritten** to take advantage of the refactoring.

Next, we rewrite some of our queries to take advantage of the refactoring.

During your testing on your real application and especially with a fully-scaled graph, you can also **profile the new queries to see if it improves performance.** On the small instance model we are using, you will not see significant improvements, but you may see **differences in the number of rows retrieved.**

##Use case #1: What people acted in a movie? We rewrite this query to use the Actor label.

Original code:

[ ]:
```
MATCH (p:Person)-[:ACTED_IN]-(m:Movie)
WHERE m.title = 'Sleepless in Seattle'
RETURN p.name AS Actor
```

New code:

[ ]:
```
MATCH (p:Actor)-[:ACTED_IN]-(m:Movie)
WHERE m.title = 'Sleepless in Seattle'
RETURN p.name AS Actor
```

##Profiling Queries For the query that uses the Person label we see this result that first retrieves the 5 Person nodes:

For the query that uses the Actor label we see this result that first retrieves the 4 Actor nodes, a slight improvement for this small graph:

##Use case #3: What movies did a person act in? We rewrite this query to use the Actor label.

Original code:

[ ]:
```
MATCH (p:Person)-[:ACTED_IN]-(m:Movie)
WHERE p.name = 'Tom Hanks'
RETURN m.title AS Movie
```

New code:

[ ]:
```
MATCH (p:Actor)-[:ACTED_IN]-(m:Movie)
WHERE p.name = 'Tom Hanks'
RETURN m.title AS Movie
```

##Use case #5: Who was the youngest person to act in a movie? We rewrite this query to use the Actor label.

Original code:

[ ]:
```
MATCH (p:Person)-[:ACTED_IN]-(m:Movie)
WHERE m.title = 'Hoffa'
RETURN  p.name AS Actor, p.born as `Year Born` ORDER BY p.born DESC LIMIT 1
```

New code:

[ ]:
```
MATCH (p:Actor)-[:ACTED_IN]-(m:Movie)
WHERE m.title = 'Hoffa'
RETURN  p.name AS Actor, p.born as `Year Born` ORDER BY p.born DESC LIMIT 1
```

##Use case #6: What role did a person play in a movie? We rewrite this query to use the Actor label.

Original code:

[ ]:
```
MATCH (p:Person)-[r:ACTED_IN]-(m:Movie)
WHERE m.title = 'Sleepless in Seattle' AND
p.name = 'Meg Ryan'
RETURN  r.role AS Role
```

New code:

[ ]:
```
MATCH (p:Actor)-[r:ACTED_IN]-(m:Movie)
WHERE m.title = 'Sleepless in Seattle' AND
p.name = 'Meg Ryan'
RETURN  r.role AS Role
```

##Use case #8: What drama movies did an actor act in? We rewrite this query to use the Actor label.

Original code:

[ ]:
```
MATCH (p:Person)-[:ACTED_IN]-(m:Movie)
WHERE p.name = 'Tom Hanks' AND
```

```
'Drama' IN m.genres
RETURN m.title AS Movie
```

New code:

[ ]:
```
MATCH (p:Actor)-[:ACTED_IN]-(m:Movie)
WHERE p.name = 'Tom Hanks' AND
'Drama' IN m.genres
RETURN m.title AS Movie
```

##Use case #10: What actors were born before 1950? For completeness of our testing, we add this new use case.

We rewrite this query to use the Actor label.

Original code:

[ ]:
```
MATCH (p:Person)
WHERE p.born < '1950'
RETURN p.name
```

New code:

[ ]:
```
MATCH (p:Actor)
WHERE p.born < '1950'
RETURN p.name
```

##Profile queries If you have a scaled graph (with more nodes/relationships than what we have been using in this course), you should also use the **PROFILE** keyword to compare the **performance** of the queries after the refactoring.

##Steps after refactoring After you have refactored the graph, what must you do?

[ ]:
```
Rewrite any Cypher queries for use cases that are affected by the refactoring.

Retest all use cases that are affected by the refactoring.
```

##Adding the Director Label

Here is the query required to add the Director label to all Person nodes connected to a Movie by at least one DIRECTED relationship.

[ ]:
```
MATCH (p:Person)
WHERE exists ((p)-[:DIRECTED]-())
SET p:Director
```

##Avoid These Labels ###Semantically orthogonal labels "Semantically orthogonal" is a fancy term that means that labels should have nothing to do with one another. You should be careful

not to use the same type of label in different contexts. For example, using the region for all types of nodes is not useful for most queries.

Here is an example where both Person nodes and User nodes are labeled with regions. If there are no use cases where the region is significant for both types of nodes, it is not helpful to use these same labels for Person nodes and User nodes.

Don't do this:

##Representing class hierarchies You also want to avoid labeling your nodes to represent hierarchies.

Suppose we have this hierarchy of Screen Actors Guild memberships:

This is often called "inheritance" or "IS-A" relationships. You should not do this where nodes have multiple labels that represent a hierarchy such as this:

Instead, you should do this:

##Do not do this! What label practices should you avoid when creating your graph data model?

[ ]:
```
The two practices that you should avoid are Semantically orthogonal labels and␣
↪labels that represent hierarchies.

Adding labels to nodes when no use case exists.
Labels to represent class hierarchies for your data.
```

##Eliminating Duplicate Data

###Duplicate data You should take care to **avoid duplicating data** in your graph. Where some databases require a form of **denormalization** to **improve the speed of a set of queries,** this is **not always** the case with a graph database. De-duplicating data gives you the added benefit of allowing you to query through a node - for example, finding other customers who have purchased a particular product, or finding similar movies based on the rating of other users.

In addition, duplicating data in the graph increases the size of the graph and the amount of data that may need to be retrieved for a query.

##New use case We have a new use case that we must account for.

###Use case #11: What movies are available in a particular language?

Our current instance model looks like this:

As you can see we **do not account for languages in the data model so we will have to add this data.**

##Duplicate data example Suppose we **add a property** to each Movie node in the graph named languages that represents the languages in which a movie is available.

Here is what the instance model would look like:

Here we see that **all Movie nodes have English in the list of languages.** This is **duplicate data** and for a scaled database, would represent a lot of duplication.

##Why eliminate duplication? Why would you refactor a graph to eliminate duplication?

[ ]:
```
Improve query performance.
Reduce the amount of storage required for the graph.
```

##Adding Language Data In the previous lesson, you learned that **duplicating data in the graph can be expensive.** To illustrate duplication of data, you will add a languages property to each Movie node in the instance model.

Execute this Cypher code to add a languages property to the Movie nodes of the graph:

[ ]:
```
MATCH (apollo:Movie {title: 'Apollo 13', tmdbId: 568, released: '1995-06-30',␣
 ↪imdbRating: 7.6, genres: ['Drama', 'Adventure', 'IMAX']})
MATCH (sleep:Movie {title: 'Sleepless in Seattle', tmdbId: 858, released:␣
 ↪'1993-06-25', imdbRating: 6.8, genres: ['Comedy', 'Drama', 'Romance']})
MATCH (hoffa:Movie {title: 'Hoffa', tmdbId: 10410, released: '1992-12-25',␣
 ↪imdbRating: 6.6, genres: ['Crime', 'Drama']})
MATCH (casino:Movie {title: 'Casino', tmdbId: 524, released: '1995-11-22',␣
 ↪imdbRating: 8.2, genres: ['Drama','Crime']})
SET apollo.languages = ['English']
SET sleep.languages =  ['English']
SET hoffa.languages =  ['English', 'Italian', 'Latin']
SET casino.languages =  ['English']
```

And here is the associated instance model:

###Querying languages Here is a query to support our new use case:

Use case #11: What movies are available in a particular language?

With this query, we find all movies in Italian.

[ ]:
```
MATCH (m:Movie)
WHERE 'Italian' IN m.languages
RETURN m.title
```

##Querying duplicate data Here is our current instance model where each Movie node has a languages property:

##For our latest use case:

###Use case #11: What movies are available in a particular language?

This query finds all movies in Italian:

[ ]:
```
MATCH (m:Movie)
WHERE 'Italian' IN m.languages
RETURN m.title
```

What this query does is retrieve all Movie nodes and then test whether the languages property contains Italian. There are two issues with the data model, especially if the graph scales:

1. **The name of the language is duplicated in many Movie nodes.**
2. **In order to perform the query, all Movie nodes must be retrieved.**

A solution here is to model properties as nodes.

##Refactor properties as nodes Here are the steps we use to refactor:

1. We take the property values for each Movie node and create a Language node.
2. Then we create the IN_LANGUAGE relationship between that Movie node and the Language node.
3. Finally, we remove the languages property from the Movie node.

This is the code to refactor the graph to turn the property values into nodes:

[ ]:
```
MATCH (m:Movie)
UNWIND m.languages AS language
WITH  language, collect(m) AS movies
MERGE (l:Language {name:language})
WITH l, movies
UNWIND movies AS m
WITH l,m
MERGE (m)-[:IN_LANGUAGE]->(l);
MATCH (m:Movie)
SET m.languages = null
```

This code iterates through all Movie nodes and creates a Language node for each language it finds and then creates the relationship between the Movie node and Language node using the

IN_LANGUAGE relationship. It uses the Cypher UNWIND clause to separate each element of the languages property list into a separate row value that is processed later in the query.

This is what the instance model looks like after the refactoring:

There will only be one node with the language value of English and we remove the languages property from all Movie nodes. This eliminates a lot of duplication in the graph.

##Separating lists What Cypher clause do you use to separate list elements?

[ ]:
```
UNWIND
```

##Adding Language nodes This is what the instance model will be refactored to:

##Creating Language Nodes Execute this code to refactor the graph to turn the languages property values into Language nodes:

[ ]:
```
MATCH (m:Movie)
UNWIND m.languages AS language
WITH  language, collect(m) AS movies
MERGE (l:Language {name:language})
WITH l, movies
UNWIND movies AS m
WITH l,m
MERGE (m)-[:IN_LANGUAGE]->(l);
MATCH (m:Movie)
SET m.languages = null
```

##Modifying the Cypher statement This is the Cypher code for what our use case used to be before the refactoring.

[ ]:
```
MATCH (m:Movie)
WHERE 'Italian' IN m.languages
RETURN m.title
```

This query can now be modified to instead use the newly-created Language node.

[ ]:
```
MATCH (m:Movie)-[:IN_LANGUAGE]-(l:Language)
WHERE  l.name = 'Italian'
RETURN m.title
```

This is the only use case that deals with languages so we need **not retest all of our queries** after the refactor.

##Adding Genre nodes In the previous Challenge, you eliminated duplication by taking the data in the languages property and creating Language nodes that are related to movies.

This challenge has three steps:

1. Modify and run the query in the sandbox query pane to use the data in the genres property for the Movie nodes and create Genre nodes using the IN_GENRE relationship to connect Movie nodes to Genre nodes.
2. Delete the genres property from the Movie nodes.
3. Rewrite the query for the use case: What drama movies did an actor act in?

[ ]:
```
//refactor code
MATCH (m:Movie)
UNWIND m.genres AS genre
MERGE (g:Genre {name: genre})
MERGE (m)-[:IN_GENRE]->(g)
SET m.genres = null;

//revised query
MATCH (p:Actor)-[:ACTED_IN]-(m:Movie)--(g:Genre)
WHERE p.name = 'Tom Hanks' AND
g.name = 'Drama'
RETURN m.title AS Movie
```

##Eliminating Complex Data in Nodes

###Example: Complex data Since nodes are used to store data about specific entities, you may have initially modeled, for example, a Production node to contain the details of the address for the production company.

####Storing complex data in the nodes like this may not be beneficial for a couple of reasons:

1. Duplicate data. Many nodes may have production companies in a particular location and the data is repeated in many nodes.
2. Queries related to the information in the nodes require that all nodes be retrieved.

##Refactoring complex data If there is a high amount of duplicate data in the nodes or if key questions of your use cases would perform better if all nodes need not be retrieved to get at the complex data, then you might consider refactoring the graph as shown here.

In this refactoring, if there are queries that need to filter production companies by their state, then it will be **faster to query based upon the State.name value,** rather than evaluating all of the

state properties for the Production nodes.

How you refactor your graph to handle complex data will depend upon the performance of the queries when your graph scales.

##1. Refactoring complex data? Why do you refactor a graph that has complex data in nodes?

[ ]:
```
Eliminate duplication of data in multiple nodes.
 Improve query performance.
```

##Relationships in the graph Neo4j as a native graph database is implemented to **traverse relationships quickly.** In some cases, it is **more performant to query the graph based upon relationship types, rather than properties** in the nodes.

Let's look at a new use case:

Use case #12: What movies did an actor act in for a particular year?

We can execute this query with the current graph:

[ ]:
```
MATCH (p:Actor)-[:ACTED_IN]-(m:Movie)
WHERE p.name = 'Tom Hanks' AND
m.released STARTS WITH '1995'
RETURN m.title AS Movie
```

It returns the movie, Apollo 13:

What if Tom Hanks acted in 50 movies in the year 1995? The query would need to retrieve all movies that Tom Hanks acted in and then check the value of the released property. What if Tom Hanks acted in a total of 1000 movies?** All of these Movie nodes would need to be evaluated.**

And here is another new use case:

###Use case #13: What actors or directors worked in a particular year?

Again, we can execute this query with the current graph:

[ ]:
```
MATCH (p:Person)--(m:Movie)
WHERE  m.released STARTS WITH '1995'
RETURN DISTINCT p.name as `Actor or Director`
```

It returns Tom Hanks and Martin Scorsese:

This query is even worse for performance because in order to return results, it must retrieve all Movie nodes. You can imagine, if the graph contained millions of movies, it would be a **very expensive query.**

##Refactoring to specialize relationships Relationships are fast to traverse and they do not take up a lot of space in the graph. In the previous two queries, the data model would benefit from having **specialized relationships between the nodes.**

So, for example, in addition to the ACTED_IN and DIRECTED relationships, we add relationships that have year information.

ACTED_IN_1992

ACTED_IN_1993

ACTED_IN_1995

DIRECTED_1992

DIRECTED_1995

At first, it seems like a lot of relationships for a large, scaled movie graph, but if the latest two new queries are important use cases, it is worth it.

This is what our instance model will now look like:

In most cases where we specialize relationships, we keep the original generic relationships as existing queries still need to use them.

The code to refactor the graph to add these specialized relationships uses the APOC library.

This is the code to refactor the ACTED_IN relationships in the graph that you will execute in the next Challenge:

[ ]:
```
MATCH (n:Actor)-[:ACTED_IN]->(m:Movie)
CALL apoc.merge.relationship(n,
  'ACTED_IN_' + left(m.released,4),
  {},
  {},
  m ,
  {}
) YIELD rel
RETURN count(*) AS `Number of relationships merged`;
```

It has a apoc.merge.relationship procedure that allows you to dynamically create relationships in the graph. It uses the 4 leftmost characters of the released property for a Movie node to create the name of the relationship.

As a result of the refactoring, the previous two queries can be rewritten and will definitely perform better for a large graph:

Here is the rewrite of the first query:

[ ]:
```
MATCH (p:Actor)-[:ACTED_IN_1995]-(m:Movie)
WHERE p.name = 'Tom Hanks'
RETURN m.title AS Movie
```

For this query the specific relationship is traversed, but fewer Movie nodes are retrieved.

And here is how we rewrite the second query:

[ ]:
```
MATCH (p:Person)-[:ACTED_IN_1995|DIRECTED_1995]-()
RETURN p.name as `Actor or Director`
```

For this query, because the year is in the relationship type, we do not have to retrieve any Movie nodes.

##1. Why specialize relationships? Why do you refactor a graph to specialize relationships?

[ ]:
```
Reduce the number of nodes that need to be retrieved.
Improve query performance.
```

##2. How do you create dynamic relationships? What do you do to create a dynamic relationship in Cypher?

[ ]:
```
Use the APOC library.
```

##Specializing ACTED_IN and DIRECTED Relationships In this Challenge, you will modify the instance model to match the following diagram. This diagram uses specialized ACTED_IN and DIRECTED relationships.

This Challenge has 2 steps:

1. Refactor all ACTED_IN relationships
2. Refactor all DIRECTED relationships

Refactor all ACTED_IN relationships Execute the following code to create a new set of relationships based on the year of the released property for each Node.

For example, Apollo 13 was released in 1995, so an additional ACTED_IN_1995 will be created between Apollo 13 and any actor that acted in the movie.

[ ]:
```
MATCH (n:Actor)-[:ACTED_IN]->(m:Movie)
CALL apoc.merge.relationship(n,
  'ACTED_IN_' + left(m.released,4),
  {},
  {},
  m ,
  {}
) YIELD rel
RETURN count(*) AS `Number of relationships merged`;
```

It should create 5 relationships.

With this refactoring, we can now confirm that our rewritten query works for the use case:

Use case #12: What movies did an actor act in for a particular year?

To verify the query has run successfully, we can attempt to use the newly created ACTED_IN_1995 relationship to see which Movies Tom Hanks acted in that were released in 1995.

[ ]:
```
MATCH (p:Actor)-[:ACTED_IN_1995]->(m:Movie)
WHERE p.name = 'Tom Hanks'
RETURN m.title AS Movie
```

It should return one movie from our dataset, Apollo 13.

##Refactor all DIRECTED relationships We can use the same method to create DIRECTED_{year} relationships between the Director and the Movie.

Modify the code you have just run to match the following pattern.

MATCH (n:Director)-[:DIRECTED]→(m:Movie)

Then modify the procedure call change the prefix of the relationship to DIRECTED_.

It should create 2 relationships.

[ ]:
```
MATCH (n:Director)-[:DIRECTED]->(m:Movie)
CALL apoc.merge.relationship(n,
  'DIRECTED_' + left(m.released,4),
  {},
  {},
  m ,
  {}
) YIELD rel
RETURN count(*) AS `Number of relationships merged`;
```

##Testing the Model With this refactoring and the previous refactoring, we can now confirm that our rewritten query works for the use case:

###Use case #12: What movies did an actor act in for a particular year?

[ ]:
```
MATCH (p:Person)-[:ACTED_IN_1995|DIRECTED_1995]->()
RETURN p.name as `Actor or Director`
```

It should return Tom Hanks and Martin Scorsese.

##Specializing RATED Relationships In the previous Challenge, you added a number of specialized relationships to the graph for the ACTED_IN and DIRECTED relationships. This Challenge has 3 steps.

In our current graph, there are RATED relationships between User nodes and Movie nodes.

Suppose we wanted to improve the performance of this query:

Use case #9: What users gave a movie a rating of 5?

##Why create specialized relationships? Let's take a practical example. Run this Cypher code to test this use case with the movie, Apollo 13.

[ ]:
```
MATCH (u:User)-[r:RATED]->(m:Movie)
WHERE m.title = 'Apollo 13' AND
r.rating = 5
RETURN u.name as Reviewer
```

It should return one User, Sandy Jones.

What if there were thousands of Users in the graph. This query would need to traverse all RATED relationships and evaluate the rating property. For a large graph, more evaluations mean longer query processing time.

In this challenge, you will specialize the RATED relationships to reflect the rating. Unlike the refactoring where we removed the genres and languages properties from the nodes, we will not remove the rating property from the RATED relationship. This is because we may need it for a query that has a reference to the relationship and needs to return the rating value.

This is the instance model you will refactor toward:

Creating specialized RATED_{rating} Relationships To pass this challenge, you must use the knowledge you gained in the previous lesson to merge a relationship between the graph using apoc.merge.relationship.

The pattern you must search for is:

MATCH (u:User)-[r:RATED]→(m:Movie)

The relationship type passed as the second parameter should be:

'RATED_'+ r.rating

[ ]:
```
MATCH (n:User)-[r:RATED]->(m:Movie)
CALL apoc.merge.relationship(n,
  'RATED_' + r.rating,
  {},
  {},
  m,
  {}
) YIELD rel
RETURN COUNT(*) AS `Number of relationships added`;
```

##Refactoring with Intermediate nodes Why do you refactor to create intermediate nodes?

[ ]:
```
Creating intermediate nodes allows for two or more nodes to connect in a single␣
 ↪context, for example, a User may rate a movie via the company's website or␣
 ↪mobile app.

That information can also be shared, allowing you to query through an element␣
 ↪that used to be siloed within a relationship.
```

```
Extracting a relationship into a node is also referred to as relating something␣
↪to a relationship.

This, however, will not reduce the number of relationships in the graph. The␣
↪size of the graph will only get bigger.
```

[ ]:
```
Connect more than two nodes in a single context.
Share data in the graph.
Relate something to a relationship.
```

###Write and run refactor code to:

1. Find an actor that acted in a Movie (MATCH (a:Actor)-[r:ACTED_IN]→(m:Movie))
2. Create (using MERGE) a Role node setting it's name to the role in the ACTED_IN relationship.
3. Create (using MERGE) the PLAYED relationship between the Actor and the Role nodes.
4. Create (using MERGE) the IN_MOVIE relationship between the Role and the Movie nodes.

[ ]:
```
// Find an actor that acted in a Movie
MATCH (a:Actor)-[r:ACTED_IN]->(m:Movie)

// Create a Role node
MERGE (x:Role {name: r.role})

// Create the PLAYED relationship
// relationship between the Actor and the Role nodes.
MERGE (a)-[:PLAYED]->(x)

// Create the IN_MOVIE relationship between
// the Role and the Movie nodes.
MERGE (x)-[:IN_MOVIE]->(m)
```

[ ]:
```

```