Neo4j

Graph database for highly interconnected data
What is a graph database?
When I first heard about graph databases, this is what I expected to see. I had no idea how that could possibly be useful. This kind of data is easily presentable from stuff in a relational database.
Graph databases are more about graph theory than bar and line graphs.
The term “graph theory” may sound scary like we’re about to get into a discussion about quantum physics, but it’s actually pretty approachable. For a great description of what we’re talking about when we say graphs here, check out Vaidehi Joshi’s #basecs.

We’re still going to go over some of these things, but for some learn-at-your-own pace stuff, Vaidehi’s got some great material.
https://dev.to/vaidehijoshi
#basecs

The best intro to computer science you can get without a destructive quantity of student-loan debt.
Before we get into how a graph database works, let's look at how a relational database stores data.

Compare to a Relational DB
1-to-1 or 1-to-N

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>company_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jamie</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Natasha</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Ali</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Ashley</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MDLogix</td>
</tr>
<tr>
<td>2</td>
<td>Test Double</td>
</tr>
<tr>
<td>3</td>
<td>Fixt</td>
</tr>
<tr>
<td>4</td>
<td>Fractured Atlas</td>
</tr>
</tbody>
</table>

Data is stored in tables. Each row is a record. Records can reference other records with what's called a “foreign key”. For 1:1 and 1:N relationships between records, one record points at another. If many records can point to a single record, this makes it a 1:N relation.
Many-to-Many

M:N relations are more complicated. You can’t have one record point at several other records easily. The most common way to handle it is to have a whole other table whose sole purpose is to contain references to both of the other records. In this example, we have a “people” table and a “friendships” table. The “friendships” table exists only to say person A is friends with person B.

Another downside of a relationship like this between records is that a “friendship” is bidirectional (technically speaking). That means that we need to know either when we’re querying to treat “person_id” and “friend_id” the same or when we insert the record that we need to insert another one with the values flipped.

But this is what the data looks like, even if it’s not how we think of it. We build up abstractions around it in our head and it becomes more intuitive when we see this pattern repeat itself over and over.
In a graph database, as we discussed in the slide that I totally didn’t plagiarize from Vaidehi, we have nodes that represent the concepts in our system. In this example, we have a person and a company, along with some data about those concepts. The nodes are the “vertices” in graph-theory terminology.
To join the concepts together, you add a relationship between them. This is the “edge” in graph-theory terms. Relations in Neo4j have a direction. Typically this is what you might consider a parent-child relationship between objects.

Notice that the relationship can also have attributes. That will be important later when we talk about querying.
There's no real structural difference between 1:N and 1:1 relationships. You simply add edges between vertices.
People tend to think about relational data like this, but we represent it in tables.
In Neo4j, the way we think about it *is* the way we store and query it.
We learned a few slides ago that relationships are directional. A unidirectional friendship doesn’t seem like a good time for either person, but unfortunately Neo4j doesn’t support storing bidirectional or undirected relationships. The good news is that, when we query, if we ask for a bidirectional or undirected relationship it will match a relationship in either direction, so we can effectively ignore this detail.
Many-to-Many

Jamie FRIENDS_WITH Ali

FRIENDS_WITH

FRIENDS_WITH

FRIENDS_WITH

Ashley FRIENDS_WITH Tasha

❤

M:N relationships are still easy. There are no join tables, just relationships.
This simple Cypher query is a little more verbose than the equivalent SQL query, but it's still pretty readable.
I would argue that simplifying it into this is even more expressive than the SQL query, though.
Querying

MATCH
  (p)<-[[:FRIENDS_WITH]]->(p2)
WHERE
  p.name = 'Jamie'
RETURN p2

The best part is that the query matcher looks exactly how you might write it in text in Slack or some other chat client.
MATCH
  (p)<-[FRIENDS_WITH]-(p2)
WHERE
  p.name = 'Jamie'
RETURN p2
MATCH
  (p)<-[:FRIENDS_WITH]->(p2)
WHERE
  p.name = 'Jamie'
RETURN p2
MATCH
  (p)<-[[:FRIENDS_WITH]]->(p2)
WHERE
  p.name = 'Jamie'
RETURN p2
We could shorten this by removing the WHERE clause and move the attributes we're matching the first node on into the node description itself, but that's not as easy to read.
Querying

```
MATCH
  (p)<-[::FRIENDS_WITH]->(p2)
WHERE
  p.name = 'Jamie'
RETURN p2
```

But I find this way easier to read. When I was learning about Neo4j, all I saw were the other style of query and I had NO idea what I was looking at.
Because we aren’t storing data in tables, nodes can be whatever type we like. All nodes are stored in the same space. This lets us reuse relationships between different types of objects. If we have the same type of relationship between multiple different types of nodes, we can use labels for the nodes just like we do with the relationships. The label would be the class name you’re using in your app for the object whose attributes are stored in that node.

And nodes can have more than one label, so you can store the entire object hierarchy if you want to use a single-table-inheritance-like construct.

For the sake of brevity, I’ll be omitting the node labels unless they improve clarity. Otherwise, they’re just noise in a demo. In a real app, I’d provide them just about every time.
We can also query based on attributes of the relationship. This query finds all people who I’ve been friends with since before 2016.

```
MATCH
  (p)<-[friendship:FRIENDS_WITH]--> (p2)
WHERE
  p.name = 'Jamie'
  AND friendship.created_at < '2016'
RETURN p2
```
MATCH
  (p)<-[:FRIENDS_WITH]->(p2)
WHERE
  p.name = 'Jamie'
RETURN p2
This is the type of query that really starts to show the power of a graph database. Relational databases are great for querying on how two objects are directly related, but a graph database like Neo4j lets you query on indirect relationships. This query traverses two FRIENDS_WITH relationships, which means it finds people who are not necessarily my friends, but are friends with my friends.
This is the equivalent SQL code to return the same result. In fact, I’ve been using SQL for well over a decade now and I couldn’t tell you with 100% certainty whether this would work without executing it. Indirect relationships are just not something it does well.

This would also return a whoooole mess of rows if we didn’t limit the SELECT clause to just the secondary friends, and that’s just for an INNER JOIN. Other types of joins would return even more rows. As great as relational DBs are, doing a lot of relationships can be intense.

Anecdotally, when I was working at STAQ, I converted a query to use joins like this so it wouldn’t have to send multiple queries to get all the data it needed. This ended up requiring a 21-way JOIN because of how interconnected all of the objects were. The query generated by ActiveRecord was several kilobytes long. Each join increases the number of rows almost exponentially, so this query ended up running the API server out of memory while retrieving just the metadata for any nontrivial report.

But anyway, back to this query, we had to set up 4 joins between 5 relations on 2 tables in SQL to get a single layer of indirection between a start and end point.
But in the graph query, all we had to do was say “make this a second-degree relationship”. It was a 2-character difference.
It gets even better. We can specify a minimum and maximum traversal distance. This query gets my friends and their friends. This is even harder to do with SQL in a single query.
You can omit the minimum length to tell it to traverse “up to” this length. However, omitting the minimum is the same as either 0 or 1 as the minimum and that depends solely on your server configuration, so just always specify the minimum.
You can also omit the maximum to tell it to go as far out as it can. As you can imagine, for highly interconnected nodes, an unbounded indirection like this can get unbelievably slow. This query returns everyone in the graph who is connected to me in any way whatsoever through any number of friends.

Anyone know where we’re going with this?
Six Degrees of Kevin Bacon

For those unfamiliar, Six Degrees of Kevin Bacon is a game where someone gives you the name of a screen actor, then you try to connect them to Kevin Bacon by way of co-stars of other movies in less than six moves. For example…
Six Degrees of Kevin Bacon

What this means is that there are six other co-star relationships between your starting actor and Kevin Bacon.
Six Degrees of Kevin Bacon

But a co-star relationship isn’t a direct relationship.
There's a movie in between, so a co-star relationship is actually two STARRED_IN relationships that point to the same movie.
Here's how we'd write that query.

MATCH
  (actor)-[:STARRED_IN]-(movie)<-[[:STARRED_IN]]-(co_star)
WHERE
  actor.name = 'Kevin Bacon'
RETURN co_star
MATCH
  (actor)-[:STARRED_IN]->()<-[:STARRED_IN]-(co_star)
WHERE
  actor.name = 'Kevin Bacon'
RETURN co_star

But we aren’t using the movie node, so we can get rid of its name. Now we see there’s nothing special about the movie node, so we can just call this a second-degree STARRED_IN relationship.

If one co-star relationship is 2 STARRED_IN relationships …
MATCH (actor)-[::STARRED_IN*1..12]-(co_star)
WHERE actor.name = 'Kevin Bacon'
RETURN co_star

That means that Six Degrees of Kevin Bacon is up to 12 degrees of STARRED_IN relationships.
If we get rid of the WHERE clause, this is how we'd write it. This query returns all actors who meet the criteria, though. What if we want to specify the other endpoint of this relationship?

MATCH
  (actor {name: 'Kevin Bacon'})-[:STARRED_IN*1..12]-(co_star)
RETURN co_star
Querying

MATCH
  (actor {name:'Kevin Bacon'})-[:STARRED_IN*1..12]-
  (co_star {name:'Kate Mulgrew'})
RETURN co_star

All we have to do is specify the name the same way. This could also go in the WHERE clause.
Querying

MATCH
  (actor {name: 'Kevin Bacon'})-[:STARRED_IN*1..12]-
  (co_star {name: 'Kate Mulgrew'})
RETURN co_star

But then, if we’re specifying the co-star by name, returning the co-star is kinda dumb, so what point is there to this exercise?

Well, the point of the game is to find the path between them.
MATCH
  p=(actor {name:'Kevin Bacon'})-[STARRED_IN*1..12]-
  (co_star {name:'Kate Mulgrew'})
RETURN p

Not only can we return information about the nodes and relationships or the nodes and relationships themselves, but we can also return the entire path it matched.
One actually useful thing a graph is good for is finding all posts that are part of the same conversation thread as another. To be part of the same thread means that post A is a reply to post B.
Conversation threads online are trees. Each post is a reply to the one above it. They’re all part of the same tree, but I wouldn’t say they’re all the same conversation. If I’m querying for replies to the original post, I would expect to get the whole tree, but if I’m querying for context for one of the later replies, I wouldn’t expect something in a completely different subtree.

I bring up trees in a conversation about graphs because trees are just a specific type of graph.
I began thinking about this problem as it relates to Mastodon, the open-source social-networking site.
I noticed that when you fetch the context of a single post on Mastodon, it sometimes takes several seconds to fetch it, its ancestors, and its descendants.
Relational threading is usually a matter of storing the parent id of a particular post. Then if you want to get all the replies to a particular post, you query for all posts whose parent_id is the id of that post. But we're not looking for replies to an individual post, we want its replies, and their replies, and their replies, and so on. We also want to go the other direction and query up the tree. It's difficult to do when you only have parent_id to match on.
Mastodon uses Postgres as its data store, so it had to pigeonhole its queries into a relational model.

In fact, this is how Mastodon does that. It uses two methods which each run two separate queries. Two of these queries use recursive subqueries. The fact that Postgres can do this is a pretty amazing testament to how flexible it is, but it’s a slow query and the server has to run two of them on each one of these API requests.

The second query in each method uses primary keys so its response time will be negligible in comparison.
Again, it took 5 seconds to run that query to return 5 results — 2 ancestors and 3 descendants.
Graph Threading

MATCH
  p=(descendant:Post)-[:IN_REPLY_TO*]->(ancestor:Post)
WHERE
  descendant.uuid = '43b501b4-073c-4747-94fe-8131d1a04829'
  OR ancestor.uuid = '43b501b4-073c-4747-94fe-8131d1a04829'
RETURN p

This query does the exact same thing. We return the path from all ancestors and descendants to the original post. All we do is specify that the original post can be either the ancestor or the descendant and tell it to go as far as it needs to get all posts.
Recommendation engines are another amazing use of graph databases.
You’ve probably seen recommendation engines on most e-commerce sites. Amazon relies on theirs heavily to upsell.
If you were designing your e-commerce site using a relational model, you might have an Order model that belongs to a Customer and LineItems that belong to both orders and Products.
In Neo4j, the relationships might look like this.
If you want to find out what other products were on an order with a particular one, you can run a query like this. You’d probably want to group the related products by their own ids and count how many times each related product appeared ...
But if we query all the way from the product to the customer and back, we can get recommendations on the customers buying stuff across multiple orders, which could increase our upsales.
This query in Neo4j got only marginally more complicated. In Postgres, this would’ve involved two more inner joins, which is definitely not a marginal change.
Questions?