neo4j

The #1 Platform for Connected Data

Graph Databases for Beginners

Bryce Merkl Sasaki & Joy Chao

EBOOK

Ebook

TABLE OF CONTENTS

Graph Databases For Beginners

Bryce Merkl Sasaki, Joy Chao & Rachel Howard

Introduction: Welcome to the World of Graph Technology

So you've heard about graph databases and you want to know what all the buzz is about. Are they just a passing trend — here today and gone tomorrow – or are they a rising tide your business and your development team can't afford to pass up?

Whether you're a business executive or a seasoned developer, something — maybe a business challenge or a task your current database can't handle — has led you on the quest to learn more about graphs and what they can do.

In this ***Graph Databases For Beginners*** ebook, we'll take you through the basics of graph technology assuming you have little (or no) background in the technology. We'll also include some useful tips that will help you if you decide to make the switch to Neo4j.

Before we dive in, what is it that makes graphs not only relevant, but necessary in today's world? The first is its focus on ***relationships***. Collectively we are gathering more data than ever before, but more and more frequently it's ***how*** that data is related that is truly important.

Take fraud detection as an example. Financial institutions and insurance companies lose billions of dollars every year to fraudsters. But many of them now rely on Neo4j to successfully uncover fraud rings by bringing previously hidden relationships to light.

But it's not only this focus on relationships that makes Neo4j so powerful. A whiteboard data model that's flexible and scalable to evolve along with your data; an intuitive query language that makes writing queries simple; and agility that saves your company valuable time all make Neo4j stand out from other NoSQL offerings and traditional relational database (RDBMS) technologies. (Are these terms unfamiliar to you? Don't worry — we'll explain them in the chapters to follow.)

In short, graph databases are the future. And even if you're just a beginner, it's never too late to get started. We invite you to learn more about this new technology in the pages that follow. And as you read, feel free to reach out to us with your questions.

Happy Graphing,
Bryce, Joy & Rachel

Chapter 1
Graphs are the Future

Why You Should Care about Graph Databases

New tech is great, but you operate in a world of budgets, timelines, corporate standards and competitors. You don't merely replace your existing database infrastructure just because something new comes along – you only take action when an orders-of-magnitude improvement is at hand.

Graph databases fit that bill, and here's why:

Performance

Your data volume will ***definitely*** increase in the future, but what's going to increase at an even faster clip is the connections (or relationships) between your individual data points. With traditional databases, relationship queries (also known as "JOINs") will come to a grinding halt as the number and depth of relationships increase. In contrast, graph database performance stays consistent even as your data grows year over year.

Flexibility

With graph databases, your IT and data architect teams move at the speed of business because the structure and schema of a graph data model flex as your solutions and industry change. Your team doesn't have to exhaustively model your domain ahead of time; instead, they can add to the existing structure without endangering current functionality.

Agility

Developing with graph databases aligns perfectly with today's agile, test-driven development practices, allowing your graph-database-backed application to evolve alongside your changing business requirements.

What Is a Graph Database? (A Non-Technical Definition)

You don't need to understand the arcane mathematical wizardry of graph theory in order to understand graph databases. On the contrary, they're more intuitive to understand than relational database management systems (RDBMS).

A graph is composed of two elements: a node and a relationship. Each node represents an entity (a person, place, thing, category or other piece of data), and each relationship represents how two nodes are associated. For example, the two nodes "cake" and "dessert" would have the relationship "is a type of" pointing from "cake" to "dessert."

Twitter is a perfect example of a graph database connecting 313 million monthly active users. In the illustration to the right, we have a small slice of Twitter users represented in a graph data model.

Each node (labeled "User") belongs to a single person and is connected with relationships describing how each user is connected. As we can see, Billy and Harry follow each other, as do Harry and Ruth, but although Ruth follows Billy, Billy hasn't (yet) reciprocated.

If the above example makes sense to you, then you've already grasped the basics of what makes up a graph database.

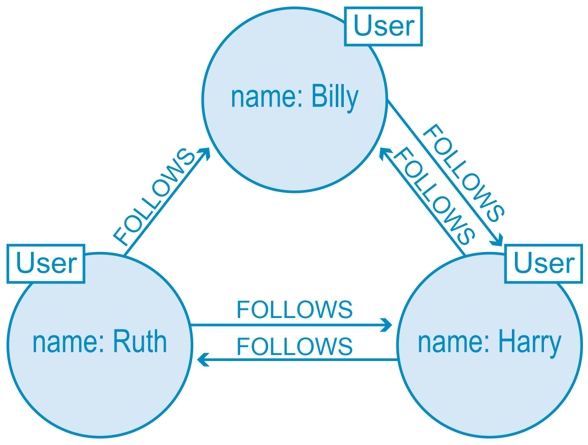


FIGURE 1.1: The connections (relationships) between different users (nodes)

Your competitors most likely aren't harnessing the power of graph technology to power their applications or analyze their big data, so this is your opportunity to step up your game and join leading companies like Walmart, eBay and Pitney Bowes.

How Graph Databases Work (In a Way You Actually Understand)

Unlike other database management systems, relationships take first priority in graph databases. This means your application doesn't have to infer data connections using things like foreign keys or out-of-band processing, like MapReduce.

The result of using graph databases instead? Your data models are simpler and more expressive than the ones you'd produce with relational databases or NoSQL (Not only SQL) stores.

There are two important properties of graph database technologies you need to understand:

1. Graph Storage

Some graph databases use "native" graph storage that is specifically designed to store and manage graphs, while others use relational or object-oriented databases instead. Non-native storage is often slower than a native approach.

2. Graph Processing Engine

Native graph processing (a.k.a. "index-free adjacency") is the most efficient means of processing data in a graph because connected nodes physically "point" to each other in the database. However, non-native graph processing engines use other means to process Create, Read, Update or Delete (CRUD) operations.

When it comes to current graph database technologies, Neo4j leads the industry as the most native when it comes to both graph storage and processing. For more information on native versus non-native graph technology, see Chapter 13.

Conclusion:
Graph Databases Are in More Places Than you Think

The real world is richly interconnected, and graph databases aim to mimic those sometimes-consistent, sometimes-erratic relationships in an intuitive way. Graph databases are extremely useful in understanding big datasets in scenarios as diverse as logistics route optimization, retail suggestion engines, fraud detection and social network monitoring.

Graph databases are on the rise, and big data is getting bigger. Your competitors most likely aren't harnessing the power of graph technology to power their applications or analyze their big data, so this is your opportunity to step up your game and join leading companies like Walmart, eBay and Pitney Bowes.

That said, it's a narrow window before your competition learns to use graphs as well. Learn to leverage graph databases today and your business retains the competitive advantage well past tomorrow.

Chapter 2
Why Data Relationships Matter

The Irony of Relational Databases

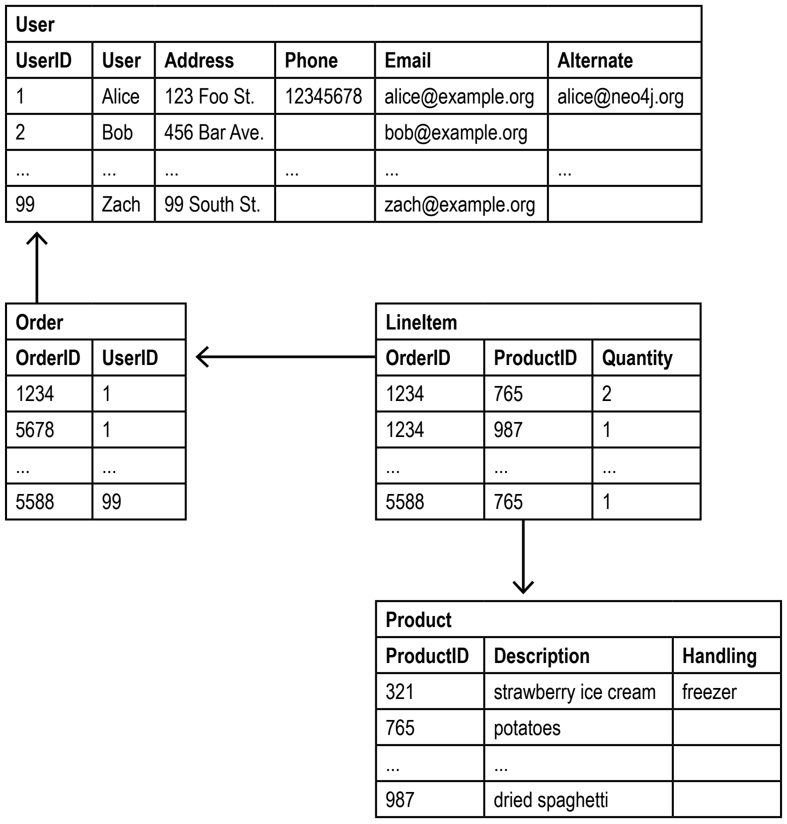
Relational databases (RDBMS) were originally designed to codify paper forms and tabular structures, and they still do this exceedingly well. Ironically, however, relational databases aren't effective at handling data relationships, especially when those relationships are added or adjusted on an ad hoc basis.

The greatest weakness of relational databases is that their schema is too inflexible. Your business needs are constantly changing and evolving, but the schema of a relational database can't efficiently keep up with those dynamic and uncertain variables.

Your business needs are constantly changing and evolving, but the schema of a relational database can't efficiently keep up with those dynamic and uncertain variables.

To compensate, your development team can try to leave certain columns empty (tech lingo: nullable), but this approach requires more code to handle the greater number of exceptions in your data. Even worse, as your data multiplies in complexity and diversity, your relational database becomes burdened with large JOIN tables that disrupt performance and hinder further development.

Consider the sample relational database below.



User

UserID	User	Address	Phone	Email	Alternate
1	Alice	123 Foo St.	12345678	alice@example.org	alice@neo4j.org
2	Bob	456 Bar Ave.		bob@example.org	
...	...	...	...	...	...
99	Zach	99 South St.		zach@example.org	

Order

OrderID	UserID
1234	1
5678	1
...	...
5588	99

LineItem

OrderID	ProductID	Quantity
1234	765	2
1234	987	1
...	...	...
5588	765	1

Product

ProductID	Description	Handling
321	strawberry ice cream	freezer
765	potatoes	
...	...	
987	dried spaghetti	

FIGURE 2.1: An example relational database where some queries are inefficient-yet-doable (e.g., "What items did a customer buy?") and other queries are prohibitively slow (e.g., "Which customers bought this product?").

In order to discover what products a customer bought, your developers would need to write several JOIN tables, which significantly slow the performance of the application. Furthermore, asking a reciprocal question like, "Which customers bought this product?" or "Which customers buying ***this*** product also bought ***that*** product?" becomes prohibitively expensive. Yet, questions like these are essential if you want to build a proper recommendation engine for your transactional application.

At a certain point, your business needs will entirely outgrow your current database schema. The problem, however, is that migrating your data to a new schema becomes incredibly effort-intensive.

Why Other NoSQL Databases Don't Fix the Problem Either

Other NoSQL (or Not only SQL) databases store sets of disconnected documents, values and columns, which in some ways gives them a performance advantage over relational databases. However, their disconnected construction makes it harder to harness data relationships properly.

Some developers add data relationships to NoSQL databases by embedding aggregate identifying information inside the field of another aggregate (tech lingo: they use foreign keys). But joining aggregates at the application level later becomes just as prohibitively expensive as in a relational database.

These foreign keys have another weak point too: they only "point" in one direction, making reciprocal queries too time-consuming to run. Developers usually work around this problem by inserting backward-pointing relationships or by exporting the dataset to an external compute structure, like Hadoop, and calculating the result with brute force. Either way, the results are slow and latent.

Graphs Put Data Relationships at the Center

When you want a cohesive picture of your big data, including the connections between elements, you need a graph database. In contrast to relational and NoSQL databases, graph databases store data relationships as relationships. This explicit storage of relationship data means fewer disconnects between your evolving schema and your actual database.

In fact, the flexibility of a graph model allows you to add new nodes and relationships without compromising your existing network or expensively migrating your data. All of your original data (and its original relationships) remain intact.

With data relationships at their center, graphs are incredibly efficient when it comes to query speeds, even for deep and complex queries. In *Neo4j in Action*, the authors performed an experiment between a relational database and a Neo4j graph database.

Their experiment used a basic social network to find friends-of-friends connections to a depth of five degrees. Their dataset included 1,000,000 people each with approximately 50 friends. The results of their experiment are listed in the table below:

Depth	RDBMS execution time(s)	Neo4j execution time(s)	Records returned
2	0.016	0.01	~2,500
3	30.267	0.168	~110,000
4	1543.505	1.359	~600,000
5	Unfinished	2.132	~800,000

FIGURE 2.2:
A performance experiment run between relational databases (RDBMS) and Neo4j shows that graph databases handle data relationships extremely efficiently.

At the friends-of-friends level (depth two), both the relational database and graph database performed adequately. However, as the depth of connectedness increased, the performance of the graph database quickly outstripped that of the relational database. It turns out data relationships are vitally important.

This comparison isn't to say other NoSQL stores or relational databases don't have a role to play (they certainly do), but they fall short when it comes to connected data relationships. Graphs, however, are extremely effective at handling connected data.

Chapter 3
Data Modeling Basics

What is Modeling Exactly?

Data modeling is an abstraction process. You start with your business and user needs (i.e., what you want your application to do). Then, in the modeling process you map those needs into a structure for storing and organizing your data. Sounds simple, right?

With traditional database management systems, modeling is far from simple. After whiteboarding your initial ideas, relational databases (RDBMS) require you to create a logical model and then force that structure into a tabular, physical model. By the time you have a working database, it looks nothing like your original whiteboard sketch (making it difficult to tell whether it's meeting user needs).

On the other hand, modeling your data for a graph database couldn't be simpler. Imagine what your whiteboard structure looks like. Probably a collection of circles and boxes connected by arrows and lines, right?

Here's the kicker: That model you drew is already a graph. Creating a graph database from there is just a matter of running a few lines of code.

A Relational Vs. Graph Data Modeling Match-Up

Let's dive into an example.

In this data center management domain (pictured below), several data centers support a few applications using infrastructure like virtual machines and load balancers. We want to create an application that manages and communicates with this data center infrastructure, so we need to create a data model that includes all relevant elements:

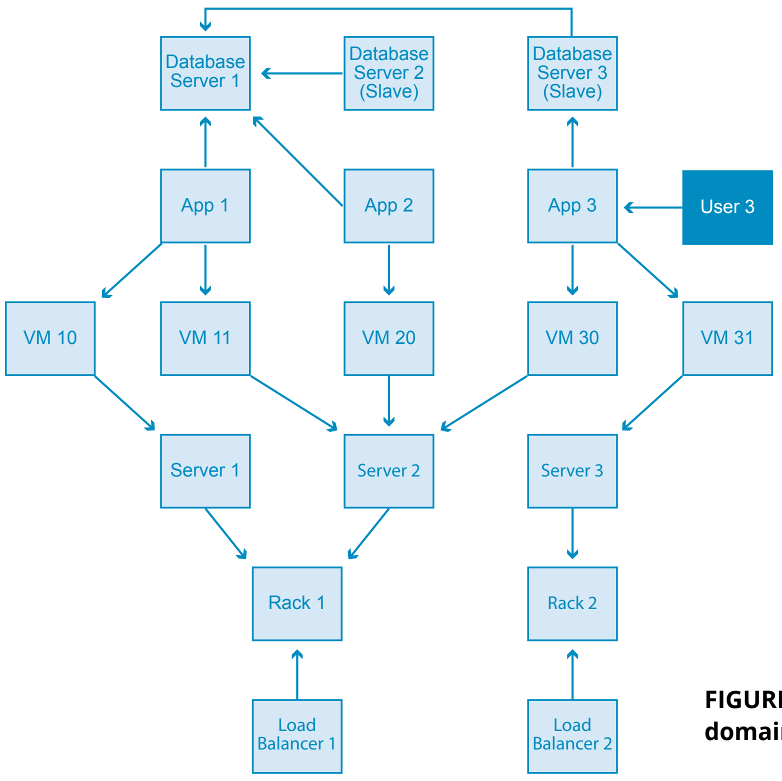


FIGURE 3.1: A sample model of a data center management domain in its initial "whiteboard" form.

"The huge advantage with Neo4j was that we were able to focus on modeling our data and how to best serve our customers instead of agonizing how to structure tables and JOINs. It also required very little coding, so we were able to keep our focus on our customers."

-- Josh Marcus, CTO, Albelli

Now, for our match-up.

If we were working with a relational database, the business leaders, subject-matter experts and system architects would convene and create a data model similar to the image above that shows the entities of this domain, how they interrelate and any rules applicable to the domain. We would then create a logical model from this initial whiteboard sketch before mapping it into the tables and relations we see below.

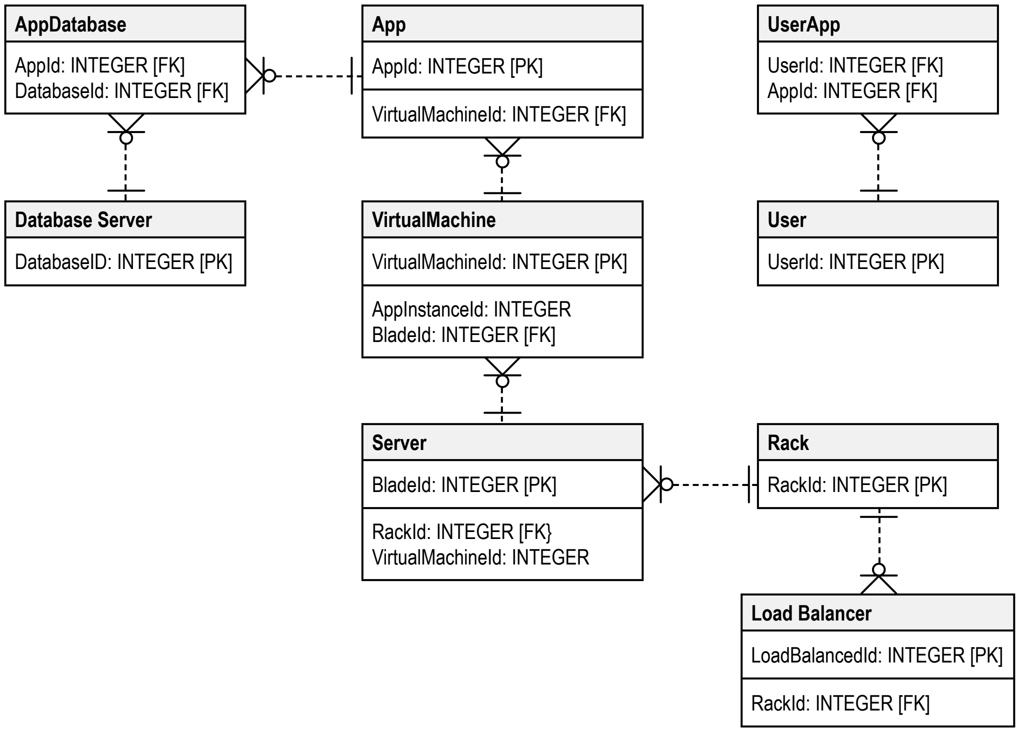


FIGURE 3.2: The relational database version of our initial "whiteboard" data model. Several JOIN tables have been added just so different tables can communicate with one another.

In the diagram above, we've had to add a lot of complexity into the system to make it fit the relational model. First, everywhere you see the annotation FK (tech lingo: foreign key) is another point of added complexity.

Second, new tables have crept into the diagram such as "AppDatabase" and "UserApp." These new tables are known as JOIN tables, and they significantly slow down the speed of a query. Unfortunately, they're also necessary in a relational data model.

Now let's look at how we would build the same application with a graph data modeling approach. At the beginning, our work is identical – decision makers convene to produce a basic whiteboard sketch of the data model (Figure 3.1).

After the initial whiteboarding step, everything looks different. Instead of altering the initial whiteboard model into tables and JOINs, we enrich the whiteboard model according to our business and user needs.

Figure 3.3 on the next page shows our newly enriched data model after adding labels, attributes and relationships:

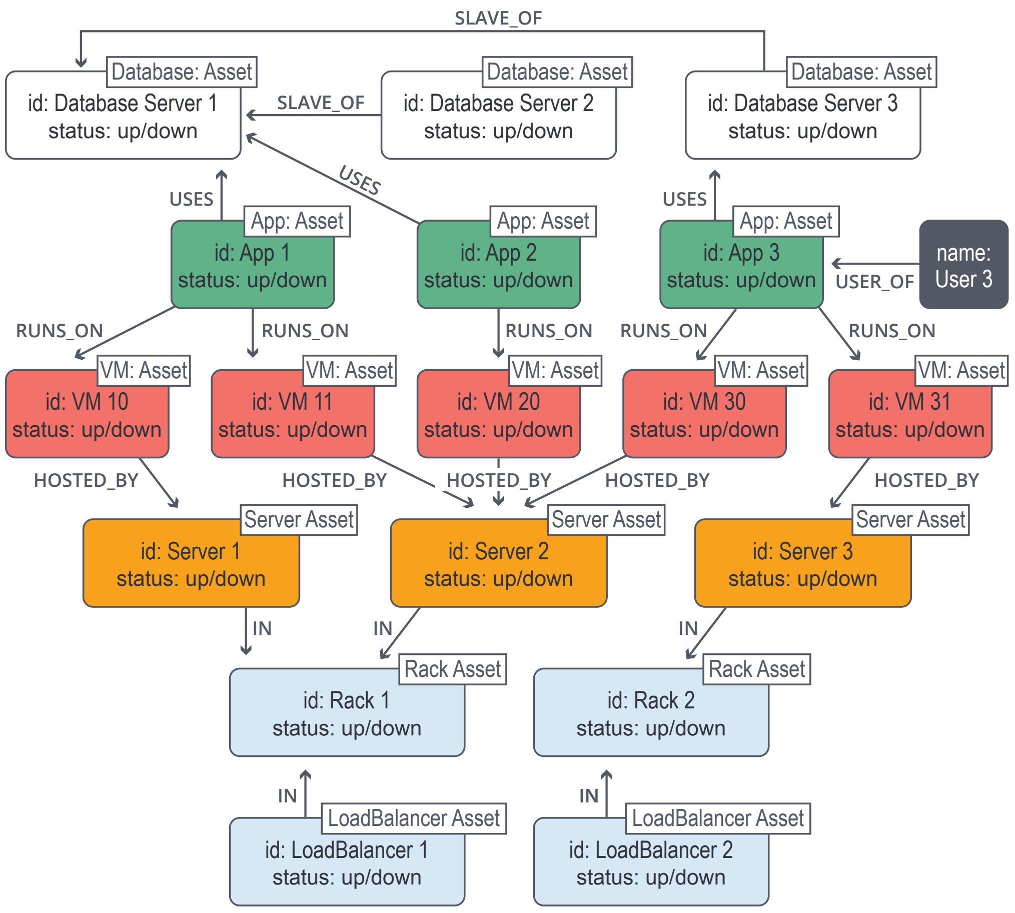


FIGURE 3.3: Our enriched data model with added labels, attributes and relationships.

Why Data Modeling Isn't a One-Off Activity

It's easy to dismiss the major differences in data modeling between relational and graph databases. After all, data modeling is just an activity you have to complete once at the beginning of your application development – right? Wrong.

Systems change, and in today's development world, they change often. In fact, your application or solution might change significantly even in mid-development. Over the lifetime of your application, your data model constantly shifts and evolves to meet changing business and user needs.

Relational databases – with their rigid schemas and complex modeling process – aren't well fit for rapid change. What you need is a data modeling approach that doesn't sacrifice performance and that supports ongoing evolution while maintaining the integrity of your data.

Now that you know the basics of data modeling, the choice is clear. You need the agile approach offered by a graph database not only to create data models quicker, but to adapt your data models to the changing needs of an uncertain future.

Systems change, and in today's development world, they change often. In fact, your application or solution might change significantly even in mid-development. Over the lifetime of your application, your data model constantly shifts and evolves to meet changing business and user needs.

Chapter 4
Data Modeling Pitfalls to Avoid

Graph databases are highly expressive when it comes to data modeling for complex problems. But expressivity isn't a guarantee that you'll get your data model right on the first try. Even graph database experts make mistakes and beginners are bound to make even more.

Let's dive into an example data model to witness the most common mistakes (and their consequences) so you don't have to learn from the same errors in your own data model.

Example Data Model: Fraud Detection in Email Communications

In this example, we'll examine a fraud detection application that analyzes users' email communications. This particular application is looking for rogue behavior and suspicious emailing patterns that might indicate illegal or unethical behavior.

We're particularly looking for patterns from past wrongdoers, such as frequently using blind-copying (BCC) and using aliases to conduct fake "conversations" that mimic legitimate interactions. In order to catch this sort of unscrupulous behavior, we'll need a graph data model that captures all the relevant elements and activities.

For our first attempt at the data model, we'll map some users, their activities and their known aliases, including a relationship describing Alice as one of Bob's known aliases. The result is a star-shaped graph with Bob in the center.

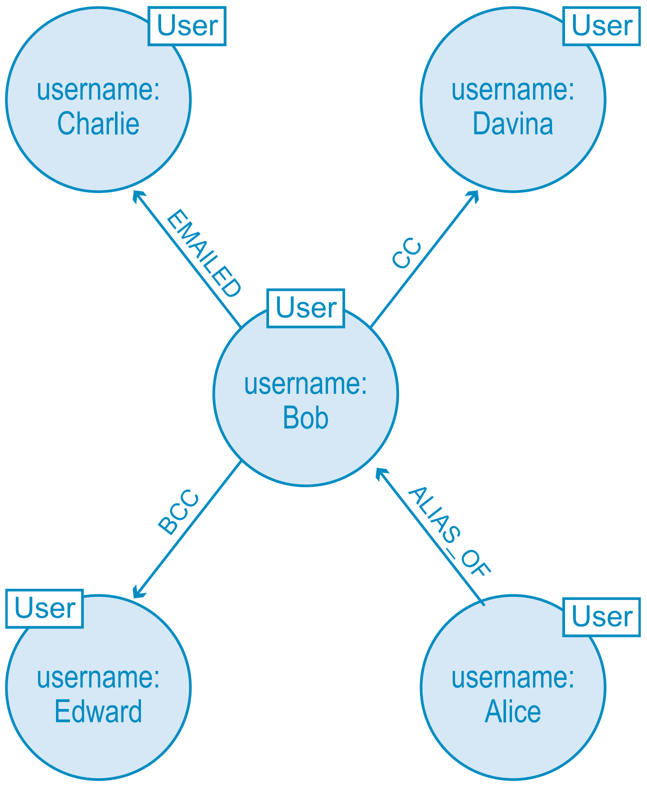


FIGURE 4.1: Our first data model attempting to map Bob's suspicious email activity with Alice as a known alias. However, this data model isn't robust enough to detect wrongful behavior.

At first blush, this initial data modeling attempt looks like an accurate representation of Bob's email activity; after all, we can easily see that Bob (an alias of Alice) emailed Charlie while BCC'ing Edward and CC'ing Davina. But we can't see the most important part of all: the email itself.

A beginning data modeler might try to remedy the situation by adding properties to the `EMAILED` relationship, representing the email's attributes as properties. However, that's not a long-term solution. Even with properties attached to each `EMAILED` relationship, we wouldn't be able to correlate connections between `EMAILED`, `CC` and `BCC` relationships – and those correlating relationships are exactly what we need for our fraud detection solution.

This is the perfect example of a common data modeling mistake. In everyday English, it's easy and convenient to shorten the phrase "Bob sent an email to Charlie" to "Bob emailed Charlie." This shortcut made us focus on the verb "emailed" rather than the email as an object itself. As a result, our incomplete model keeps us from the insights we're looking for.

The Fix: A Stronger Fraud Detection Data Model

To fix our weak model, we need to add nodes to our graph model that represent each of the emails exchanged. Then, we need to add new relationships to track who wrote the email and to whom it was sent, CC'ed and BCC'ed.

The result is another star-shaped graph, but this time the email is at the center, allowing us to efficiently track its relationship to Bob and possibly some suspicious behavior.

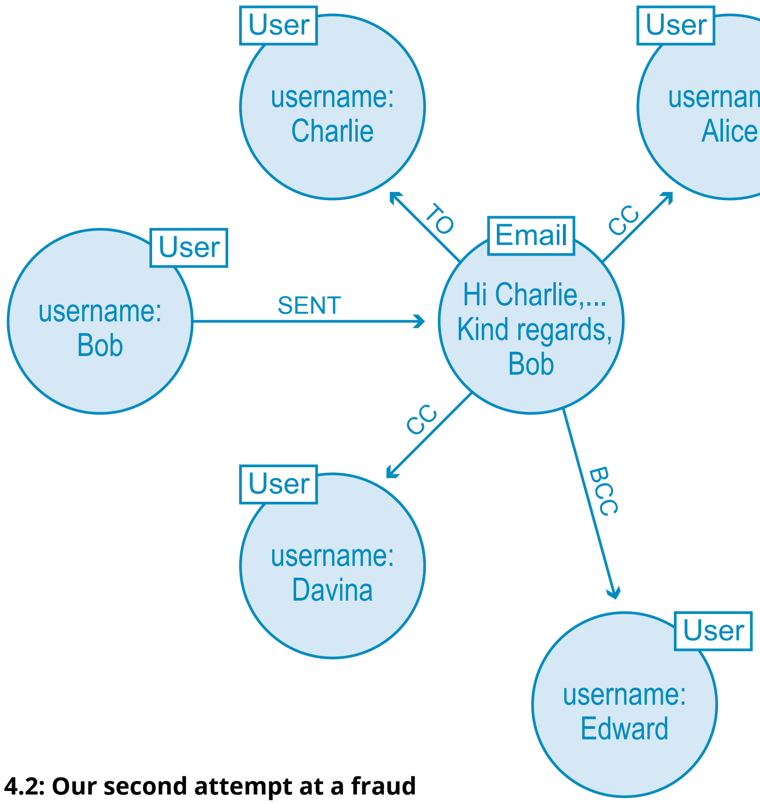


FIGURE 4.2: Our second attempt at a fraud detection data model. This iteration allows us to more easily trace the relationships of who is sending and receiving each email message.

Of course we aren't interested in tracking just one email but many, each with its own web of interactions to explore. Over time, our email server logs more interactions, giving us something like the graph below.

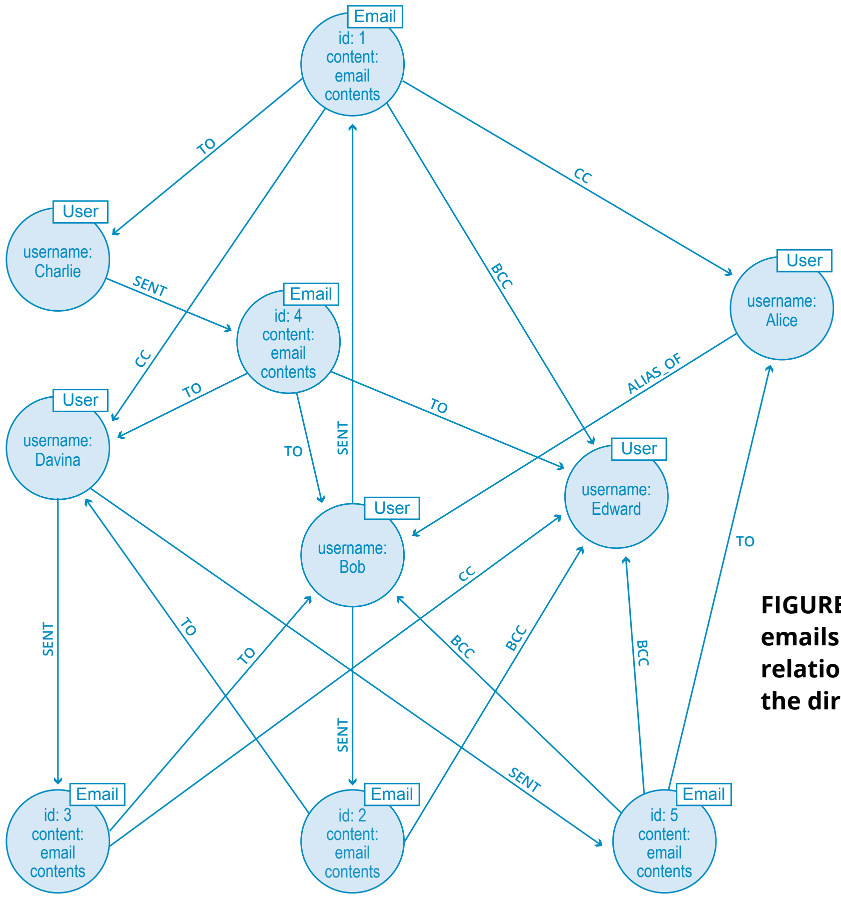


FIGURE 4.3: A data model showing many emails over time and their various relationships, including the sender and the direct, CC and BCC receivers.

The Next Step: Tracking Email Replies

At this point, our data model is more robust, but it isn't complete. We can see who sent and received emails, and we can see the content of the emails themselves. Nevertheless, we can't track any replies or forwards of our given email communications. In the case of fraud or cybersecurity, we need to know if critical business information has been leaked or compromised.

To complete this upgrade, beginners might be tempted to simply add `FORWARDED` and `REPLIED_TO` relationships to our graph model, like in the example below.

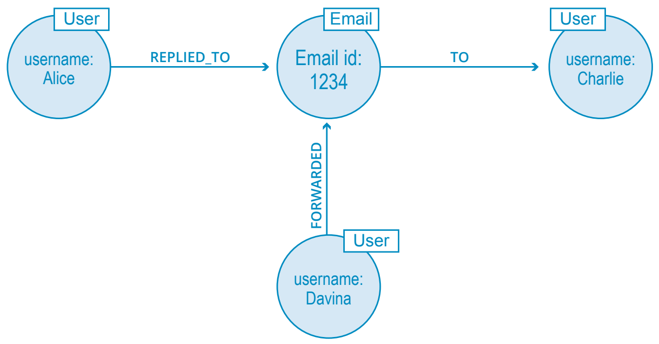


FIGURE 4.4: Our updated data model with `FORWARDED` and `REPLIED_TO` relationships in addition to the original TO relationship.

This approach, however, quickly proves inadequate. Much in the same way the **EMAILED** relationship didn't give us the proper information, simply adding **FORWARDED** or **REPLIED_TO** relationships doesn't give us the insights we're really looking for.

To build a better data model, we need to consider the fundamentals of this particular domain. A reply to an email is both a new email and a reply to the original. The two roles of a reply can be represented by attaching two labels – Email and Reply – to the appropriate node.

We can then use the same **TO**, **CC** and **BCC** relationships to map whether the reply was sent to the original sender, all recipients or a subset of recipients. We can also reference the original email with a **REPLY_TO** relationship.

The resulting graph data model is shown below.

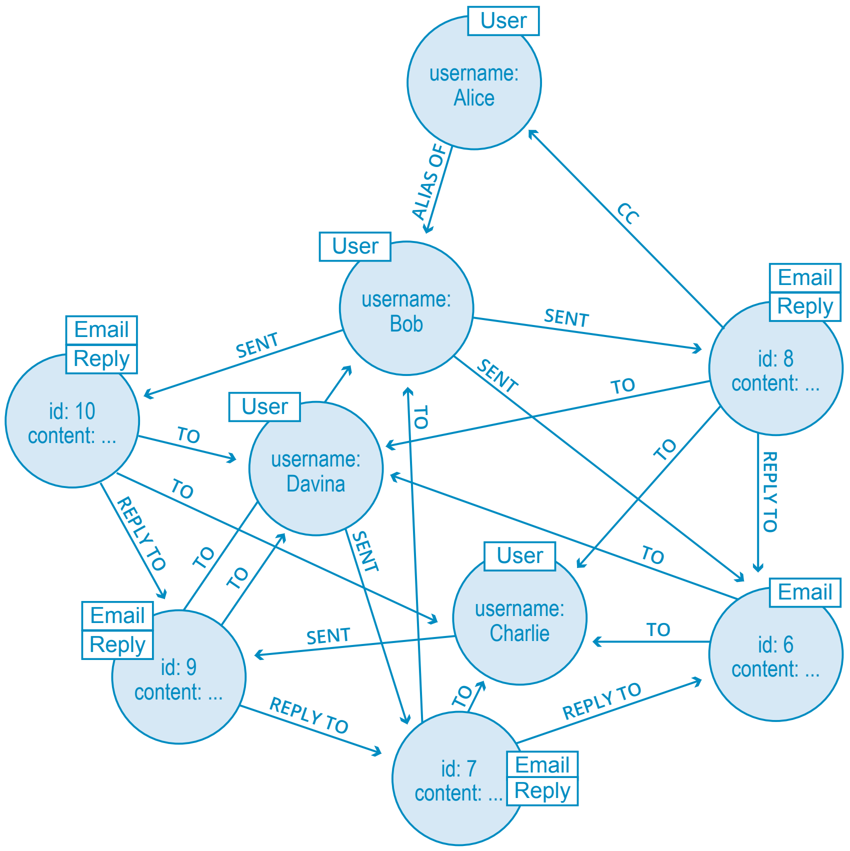


FIGURE 4.5: Our updated data model with the labels EMAIL and REPLY added to appropriate nodes.

Homework: Data Modeling for Email Forwards

Not only can we see who replied to Bob's original email, but we can track replies-to-replies and replies-to-replies-to-replies, and so on to an arbitrary depth. If we're trying to track a suspicious number of replies to known aliases, the above graph data model makes this extremely simple.

Equally important to tracking email replies is tracking email forwards, especially when it comes to leaked business information.

As a data modeling acolyte, your homework assignment is to document how you would model the forwarded email data, tracking the relationships with senders, direct recipients, CC'ed recipients, BCC'ed recipients and the original email.

Check your work on pages 61 and 62 of the O'Reilly Graph Databases book found here.

Data modeling has been made much easier with the advent of graph databases. However, while it's simpler than ever to translate your whiteboard model into a physical one, you need to ensure your data model is designed effectively for your particular use case.

There are no absolute rights or wrongs with graph data modeling, but you should avoid the pitfalls mentioned above in order to glean the most valuable insights from your data.

Up until now, the query language used by developers and data architects (i.e., SQL) was too arcane and esoteric to be understood by business decision makers. But just as graph databases have made the modeling process more understandable for the uninitiated, so has a graph database query language made it easier than ever for the common person to understand and create their own queries.

Chapter 5
Why a Database Query Language Matters

Why We Need Query Languages

Up to this point in our beginner's series, all of our database models have been in the form of diagrams like the one below.

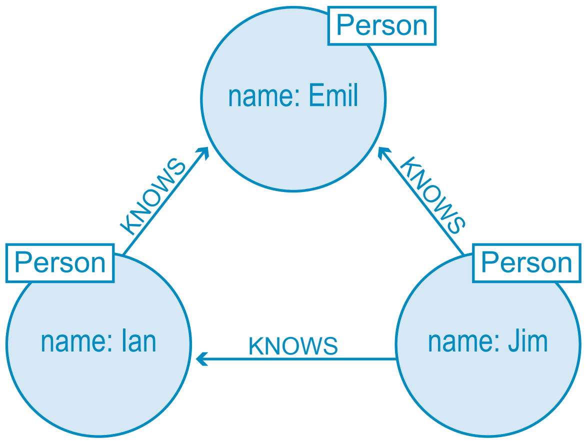


FIGURE 5.1: Example data model where Emil knows Jim and Ian, Ian knows Emil and Jim, and Jim knows Ian and Emil.

Graph diagrams like this one are perfect for describing a graph database outside of any technology context. However, when it comes to actually using a database, every developer, architect and business stakeholder needs a concrete mechanism for creating, manipulating and querying data. That is, we need a query language.

Up until now, the query language used by developers and data architects (i.e., SQL) was too arcane and esoteric to be understood by business decision makers. But just as graph databases have made the modeling process more understandable for the uninitiated, so has a graph database query language made it easier than ever for the common person to understand and create their own queries.

Why Linguistic Efficiency Matters

If you're not a techie, you might be wondering why a database query language matters at all. After all, if query languages are anything like natural human languages, then shouldn't they all be able to ultimately communicate the same point with just a few differences in phrasing? The answer is both yes and no.

Let's consider a natural language example. In English, you might say, "I used to enjoy after-dinner conversation" while reminiscing about your childhood. In Spanish, this same phrase is written as, "Disfrutaba sobremesa." Both languages express the same idea, but one is far more efficient at communicating it.

When it comes to a query language, the linguistics of efficiency are similar. A single query in SQL can be many lines longer than the same query in a graph database query language like Cypher. (Here's one great example of efficient mapping from a natural language to Cypher.)

Lengthy queries not only take more time to run, but they also are 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. For example, imagine if an outside developer had to pick through a complicated query and try to figure out the intent of the original developer – trouble would certainly ensue.

But what level of efficiency gains are we talking about between SQL queries and graph queries? How much more efficient is one versus another? The answer: Fast enough to make a significant difference to your business.

The efficiency of graph queries means they run in real time, and in an economy that runs at the speed of a single tweet, that's a bottom-line difference you can't afford to ignore.

Graph database models...not only communicate how your data is related, but they also help you clearly communicate the kinds of questions you want to ask of your data model. Graph models and graph queries are just two sides of the same coin.

The Intimate Relationship between Modeling and Querying

Before diving into the mechanics of a graph database query language below, it's worth noting that a query language isn't just about ***asking*** (a.k.a. querying) the database for a particular set of results; it's also about ***modeling*** that data in the first place.

We know from previous chapters that data modeling for a graph database is as easy as connecting circles and lines on a whiteboard. What you sketch on the whiteboard is what you store in the database.

On its own, this ease of modeling has many business benefits, the most obvious of which is that you can understand what your database developers are actually creating. But there's more to it: An intuitive model ***built with the right query language*** ensures there's no mismatch between how you built the data and how you analyze it.

A query language represents its model closely. That's why SQL is all about tables and JOINs while Cypher is all about relationships between entities. As much as the graph model is more natural to work with, so is Cypher as it borrows from the pictorial representation of circles connected with arrows that even a child can understand.

In a relational database, the data modeling process is so far abstracted from actual day-to-day SQL queries that there's a major disparity between analysis and implementation. In other words, the process of building a relational database model isn't fit for asking (and answering) questions efficiently from that same model.

Graph database models, on the other hand, not only communicate how your data is related, but they also help you clearly communicate the kinds of questions you want to ask of your data model. Graph ***models*** and graph ***queries*** are just two sides of the same coin.

The right database query language helps us traverse both sides.

An Introduction to Cypher, the Graph Database Query Language

It's time to dive into specifics. While most relational databases use a form of SQL as their query language, the graph database world is more varied so we'll look specifically at a single graph database query language: Cypher.

(While most frequently used in conjunction with Neo4j, Cypher is currently an open source, vendor-neutral and cross-platform query language, thanks to the openCypher project.

This introduction isn't meant to be a reference document for Cypher but merely a high-level overview.

Cypher is designed to be easily read and understood by developers, database professionals and business stakeholders alike. It's easy to use because it matches the way we intuitively describe graphs using diagrams.

The basic notion of Cypher is that it allows you to ask the database to find data that matches a specific pattern. Colloquially, we might ask the database to "find things like this," and the way we describe what "things like this" look like is to draw them using ASCII art.

Consider this simple pattern:

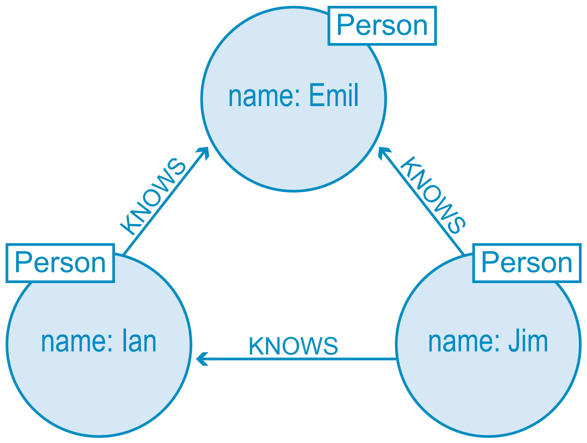


FIGURE 5.2: Example data model where Emil knows Jim and Ian, Ian knows Emil and Jim, and Jim knows Ian and Emil.

If we want to express the pattern of this basic graph in Cypher, we would write:

```
(emil)<-[:KNOWS]-(jim)-[:KNOWS]->(ian)-[:KNOWS]->(emil)
```

This Cypher statement describes a path that forms a triangle that connects a node we call **`jim`** to the two nodes we call **`ian`** and **`emil`**, and that also connects the **`ian`** node to the **`emil`** node. As you can see, Cypher naturally follows the way we draw graphs on the whiteboard.

Now, while this Cypher pattern describes a simple graph structure, it doesn't yet refer to any particular data in the database. To bind the pattern to specific nodes and relationships in an existing dataset, we first need to specify some property values and node labels that help locate the relevant elements in the dataset.

Here's our more fleshed-out query:

```
(emil:Person {name:'Emil'})
     <-[:KNOWS]-(jim:Person {name:'Jim'})
       -[:KNOWS]->(ian:Person {name:'Ian'})
       -[:KNOWS]->(emil)
```

Here we've bound each node to its identifier using its name property and Person label. The **`emil`** identifier, for example, is bound to a node in the dataset with a label **`Person`** and a name property whose value is **`Emil`**. Anchoring parts of the pattern to real data in this way is normal Cypher practice.

The simplest queries consist of a MATCH clause followed by a RETURN clause. We draw relationships using pairs of dashes with greater-than or less-than signs (--> and <--) where the < and > signs indicate relationship direction.

The Beginner's Guide to Cypher Clauses

(Disclaimer: This section is still for beginners, but it's definitely developer-oriented. If you're just curious about database query languages in general, skip to the "Other Query Languages" section below for a nice wrap-up.)

Like most query languages, Cypher is composed of clauses.

The simplest queries consist of a **`MATCH`** clause followed by a **`RETURN`** clause. Here's an example of a Cypher query that uses these three clauses to find the mutual friends of a user named Jim:

```
MATCH (a:Person {name:'Jim'})-[:KNOWS]->(b)-[:KNOWS]-
>(c), (a)-[:KNOWS]->(c)

RETURN b, c
```

Let's look at each clause in further detail:

`MATCH`
The **`MATCH`** clause is at the heart of most Cypher queries.

Using ASCII characters to represent nodes and relationships, we draw the data we're interested in. We draw nodes with parentheses, just like in these examples from the query above:

```
(a:Person {name:'Jim'})

(b)

(c)

(a)
```

We draw relationships using pairs of dashes with greater-than or less-than signs (--> and <--) where the < and > signs indicate relationship direction. Between the dashes, relationship names are enclosed by square brackets and prefixed by a colon, like in this example from the query above:

```
-[:KNOWS]->
```

Node labels are also prefixed by a colon. As you see in the first node of the query, **`Person`** is the applicable label.

```
(a:Person … )
```

Node (and relationship) property key-value pairs are then specified within curly braces, like in this example:

```
( … {name:'Jim'})
```

In our original example query, we're looking for a node labeled Person with a name property whose value is **`Jim`**. The return value from this lookup is bound to the identifier **`a`**. This identifier allows us to refer to the node that represents Jim throughout the rest of the query.

It's worth noting that this pattern

```
(a)-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-[:KNOWS]->(c)
```

could, in theory, occur many times throughout our graph data, especially in a large user set.

To confine the query, we need to anchor some part of it to one or more places in the graph. In specifying that we're looking for a node labeled Person whose name property value is **`Jim`**, we've bound the pattern to a specific node in the graph — the node representing Jim.

Cypher then matches the remainder of the pattern to the graph immediately surrounding this anchor point based on the provided information about relationships and neighboring nodes. As it does so, it discovers nodes to bind to the other identifiers. While **`a`** will always be anchored to **`Jim,`** **`b`** and **`c`** will be bound to a sequence of nodes as the query executes.

`RETURN`
This clause specifies which expressions, relationships and properties in the matched data should be returned to the client. In our example query, we're interested in returning the nodes bound to the **`b`** and **`c`** identifiers.

Cypher is intended to be easy-to-learn for SQL veterans while also being easy for beginners. At the same time, Cypher is different enough to emphasize that we're dealing with graphs, not relational sets.

Other Cypher Clauses

Other clauses you can use in a Cypher query include:

`WHERE`
Provides criteria for filtering pattern matching results.

`CREATE and CREATE UNIQUE`
Create nodes and relationships.

`MERGE`
Ensures that the supplied pattern exists in the graph, either by reusing existing nodes and relationships that match the supplied predicates, or by creating new nodes and relationships.

If your team comes from an SQL background, a query language like Cypher will be easy to learn and even easier to execute. And when it comes to your enterprise-level application, you'll be glad that the language underpinning it all is build for speed and efficiency.

DELETE/REMOVE
Removes nodes, relationships, and properties.

SET
Sets property values and labels.

ORDER BY
Sorts results as part of a **RETURN**.

SKIP LIMIT
Skip results at the top and limit the number of results.

FOREACH
Performs an updating action for each element in a list.

UNION
Merges results from two or more queries.

WITH
Chains subsequent query parts and forwards results from one to the next. Similar to piping commands in Unix.

If these clauses look familiar – especially if you're a SQL developer – that's great! Cypher is intended to be easy-to-learn for SQL veterans while also being easy for beginners. (Click here for the most up-to-date Cypher Refcard to take a deeper dive into the Cypher query language.)

At the same time, Cypher is different enough to emphasize that we're dealing with graphs, not relational sets.

Other Query Languages

Cypher isn't the only graph database query language; other graph databases have their own means of querying data as well. Many, including Neo4j, support the RDF query language SPARQL and the partially-imperative, path-based query language Gremlin.

Conclusion

Not everyone gets hands-on with their database query language on the day-to-day level; however, your down-in-the-weeds development team needs a practical way of modeling and querying data, especially if they're tackling a graph-based problem.

If your team comes from an SQL background, a query language like Cypher will be easy to learn and even easier to execute. And when it comes to your enterprise-level application, you'll be glad that the language underpinning it all is build for speed and efficiency.

Chapter 6
Imperative vs. Declarative Query Languages

The history of programming languages goes back to the beginning of computing, with new languages being created for each variation and paradigm. When it comes to databases, two well-known paradigms include imperative and declarative query languages.

All too often, the latter is often broadly defined as being any database query language that is not imperative. However, to define it in such a manner is too broad.

In this chapter, we will discuss the different features of the imperative and declarative database query languages, as each has its individual strengths and weaknesses. Selecting which language to use will depend specifically upon your individual situation.

Imperative Query Languages

If the query languages were human archetypes, imperative languages would be the micromanaging boss who gives instructions down to the final detail. In the most basic sense, imperative query languages are used to describe how you want something done specifically. This is accomplished with explicit control in a detailed, step-by step manner; the sequence and wording of each line of code plays a critical role.

Some well-known general imperative programming languages include Python, C and Java. In the world of graph databases, there aren't any purely imperative query languages. However, both Gremlin and the Java API (for Neo4j) include imperative features. These two options provide you with more detailed power over the execution of their task. If written correctly, there are no surprises – you will get exactly what you want done.

However, imperative database query languages can also be limiting and not very user-friendly, requiring an extensive knowledge of the language and deep technical understanding of physical implementation details prior to usage. Writing one part incorrectly creates faulty outcomes.

As a result, imperative languages are more prone to human error. Additionally, users must double-check the environment before and after the query and be prepared to deal with any potential erroneous scenarios.

To better illustrate the differences, imagine you have two children: Izzy and Duncan. Izzy represents an imperative query language and Duncan the declarative query language.

To get the two children to make their beds, you take differing approaches. For Duncan, it is easy. Simply instruct Duncan to make his bed and he will do it however he sees fit. Yet, he might make it slightly differently from what you had in mind, especially if you're a picky parent.

Izzy requires an entirely different process. You must first inform her that she needs both sheets and blankets to make her bed, and that those materials can be found on top of her bed. Then she requires step-by-step instructions, such as "spread the sheet over the mattress" and then "tuck in the edges."

The final result will be closely similar to Duncan's (or perhaps, exactly the same). At the end of the process, both children have their beds made.

If your project requires finer accuracy and control, imperative query languages do the job well. If the speed and productivity of the process matter more, declarative languages offer the flexibility of getting results without as much effort. Ultimately, the choice depends upon your individual use case.

Declarative Query Languages

On the other end of the spectrum, declarative database query languages let users express what data to retrieve, letting the engine underneath take care of seamlessly retrieving it. They function in a more general manner and involve giving broad instructions about what task is to be completed, rather than the specifics on how to complete it. They deal with the results rather than the process, thus focusing less on the finer details of each task.

Some well-known general declarative programming languages include Ruby, R and Haskell. SQL (Structured Query Language) is also a declarative query language and is the industry standard for relational databases. In the graph database ecosystem, several query languages are considered declarative: Cypher, SPARQL and Gremlin (which also includes some imperative features).

Using a declarative database query language may also result in better code than what can be created manually, and it is usually easier to understand the purpose of the code written in a declarative language. Declarative query languages are also easier to use, as they simply focus on what must be retrieved and do so quickly.

However, declarative languages have their own trade-offs. Users have little to no control over how inputs are dealt with; if there is a bug in the language, the user will have to rely on the providers of the language to fix the problem. Likewise, if the user wants to use a function that the query language doesn't support, they are often at a loss.

In the previous example of the children, Duncan was able to complete his task in a method faster and easier for his parent than Izzy. However, imagine now that you want them to wash the dishes.

It is the same process for Izzy: You'd need to walk through each step with her so she can learn how the process works. For Duncan, however, we have hit a snag. Duncan has never learned how to wash the dishes. You will stay in that impasse with Duncan unless his programming engineers decide to teach him how to wash the dishes. (Duncan isn't like most children.)

Conclusion

This chapter is not meant to pit the two types of database query languages against each other; it is meant to highlight the basic pros and cons to consider before deciding which language to use for your project.

You should select the best database query language paradigm for your specific use case. Neither paradigm is better than the other; they each have different strengths for software development.

If your project requires finer accuracy and control, imperative query languages do the job well. If the speed and productivity of the process matter more, declarative languages offer the flexibility of getting results without as much effort. Ultimately, the choice depends upon your individual use case.

Chapter 7
Graph Theory & Predictive Modeling

As a more developed field, graph theory (the mathematics behind graph databases) helps us gain insight into new domains. Combined with the social sciences, there are many concepts that can be straightforwardly used to gain insight from graph data.

Triadic Closures

One of the most common properties of social graphs is that of triadic closures. This is the observation that if two nodes are connected via a path with a mutual third node, there is an increased likelihood of the two nodes becoming directly connected in the future.

In a social setting, a triadic closure would be a situation where two people with a mutual friend have a higher chance of meeting each other and becoming acquainted.

The triadic closure property is most likely to be upheld when a graph has a node A with a strong relationship to two other nodes, B and C. This then gives B and C a chance of a relationship, whether it be weak or strong. Although this is not a guarantee of a potential relationship, it serves as a credible predictive indicator.

Let's take a look at this example.

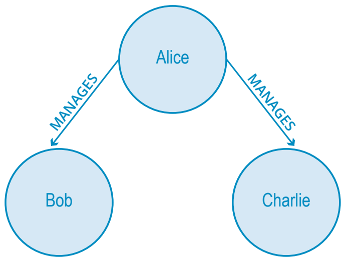


FIGURE 7.1: A graph depicting managerial relationships.

Above is an organizational hierarchy where Alice manages both Bob and Charlie. This is rather strange, as it would be unlikely for Bob and Charlie to be unacquainted with one another while sharing the same manager.

As it is, there is a strong possibility they will end up working together due to the triadic closure property. This will create either a **`WORKS_WITH`** (strong) or **`PEER_OF`** (weak) relationship between the two of them, closing the triangle – hence the term *triadic closure*.

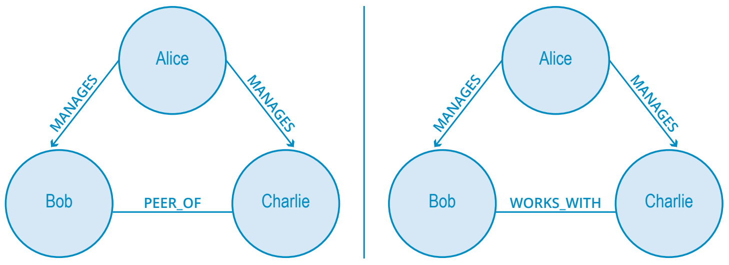


FIGURE 7.2: An example of two different triadic closures.

The triadic closure property is most likely to be upheld when a graph has a node A with a strong relationship to two other nodes, B and C. This then gives B and C a chance of a relationship, whether it be weak or strong. Although this is not a guarantee of a potential relationship, it serves as a credible predictive indicator.

Structural Balance

However, another aspect to consider in the formation of stable triadic closures is the quality of the relationships involved in the graph. To illustrate the next concept, assume that the **`MANAGES`** relationship is somewhat negative while the **`PEER_OF`** and **`WORKS_WITH`** relationships are more positive.

Based on the triadic closure property, we can assume that we can fill in the third relationship with any label, such as having everyone manage each other, like in Figure 7.3 or the weird situation in Figure 7.4, below:

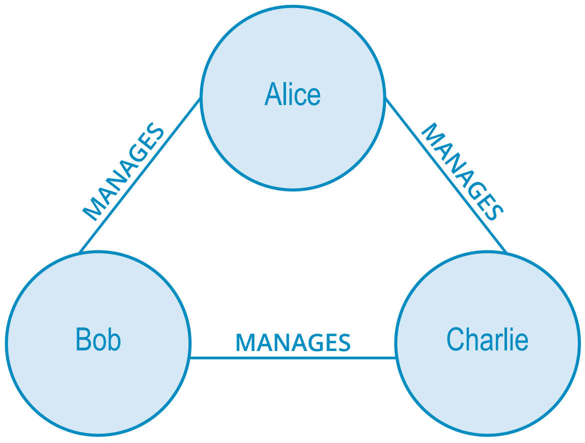


FIGURE 7.3: Example triadic closure.

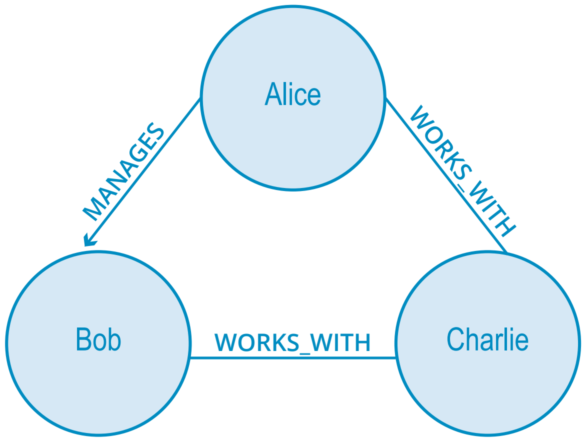


FIGURE 7.4: Example triadic closure.

However, you can see how uncomfortable those working situations would be in reality. In the second image, Charlie finds himself both the peer of a boss and a fellow worker. It would be difficult for Bob to figure out how to treat Charlie – as a fellow coworker or as the peer of his boss?

We have an innate preference for structural symmetry and rational layering. In graph theory, this is known as ***structural balance.***

A structurally balanced triadic closure is made of relationships of all strong, positive sentiments (such as in Figure 7.5) or two relationships with negative sentiments and a single positive relationship (Figure 7.6).

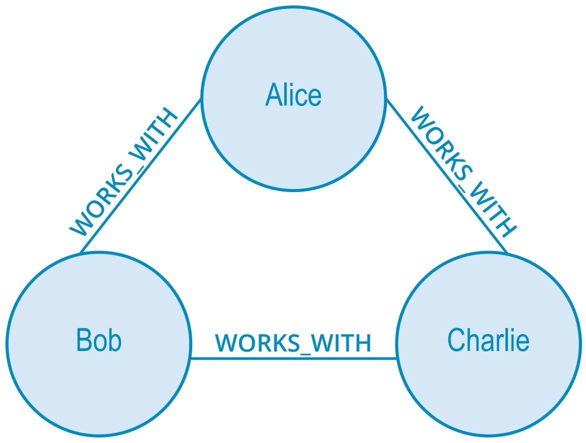


FIGURE 7.5: A structurally balanced triadic closure with relationships made of strong, positive sentiments.

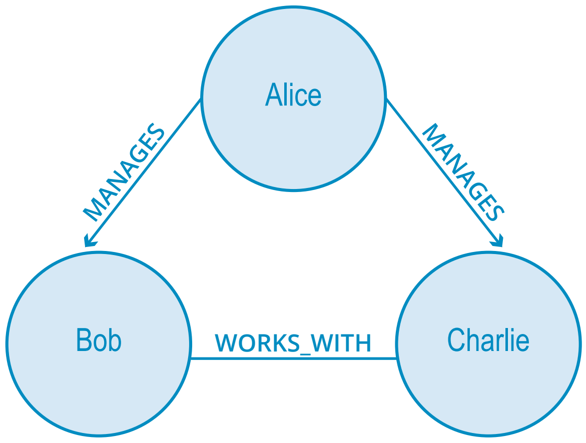


FIGURE 7.6: A structurally balanced triadic closure with two relationships with negative sentiments and a single positive relationship.

While graphs and our understanding of them are rooted in hundreds of years of study, we continue to find new ways to apply them to our personal, social and business lives. Technology today offers another method of understanding these principles in the form of the modern graph database.

Balanced closures help with predictive modeling in graphs. The simple action of searching for chances to create balanced closures allows for the modification of the graph structure for accurate predictive analysis.

Local Bridges

We can go further and gain more valuable insight into the communications flow of our organizations by looking at local bridges. These refer to a tie between two nodes where the endpoints of the local bridge are not otherwise connected, nor do they share any common neighbors. You can think of local bridges as connections between two distinct clusters of the graph. In this case, one of the ties has to be weak.

For example, the concept of weak links is relevant in algorithms for job search. Studies have shown that the best sources of jobs come from looser acquaintances rather than close friends. This is because closer friends tend to share a similar world-view (are in the same graph component) but looser friends across a local bridge are in a different social network (and are in a different graph component).

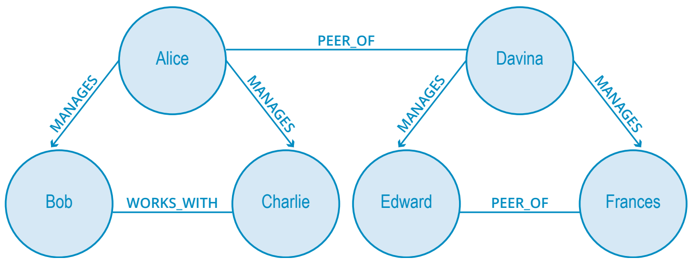


FIGURE 7.7: Alice and Davina are connected by a local bridge but belong to different graph components.

In the image above, Davina and Alice are connected by a local bridge but belong to different graph components. If Davina were to look for a new job, she would be more likely to find a successful recommendation from Alice than from Frances.

This property of local bridges being weak links is something that is found throughout social graphs. As a result, we can make predictive analyses based on empirically derived local bridges and strong triadic closure notions.

The Final Takeaway

While graphs and our understanding of them are rooted in hundreds of years of study, we continue to find new ways to apply them to our personal, social and business lives. Technology today offers another method of understanding these principles in the form of the modern graph database.

Chapter 8
Graph Search Algorithm Basics

While graph databases are certainly a rising tide in the world of technology, graph theory and graph algorithms are mature and well-understood fields of computer science.

In particular, graph search algorithms can be used to mine useful patterns and results from persisted graph data. As this is a practical introduction to graph databases, this chapter will discuss the basics of graph theory without diving too deeply into the mathematics.

Depth- and Breadth-First Search

There are two basic types of graph search algorithms: depth-first and breadth-first.

The former travel from a starting node to some end node before repeating the search down a different path from the same start node until the query is answered. Generally, depth-first is a good choice when trying to discover discrete pieces of information. They are also a good strategy for general graph traversals.

The most classic or basic level of depth-first is an uninformed search, where the algorithm searches a path until it reaches the end of the graph, then backtracks to the start node and tries a different path.

On the contrary, dealing with semantically rich graph databases allows for informed searches, which conduct an early termination of a search if nodes with no compatible outgoing relationships are found. As a result, informed searches also have lower execution times.

(For the record, Cypher queries and Java traversals generally perform informed searches.)

Breadth-first algorithms conduct searches by exploring the graph one layer at a time. They begin with nodes one level deep away from the start node, followed by nodes at depth two, then depth three, and so on until the entire graph has been traversed.

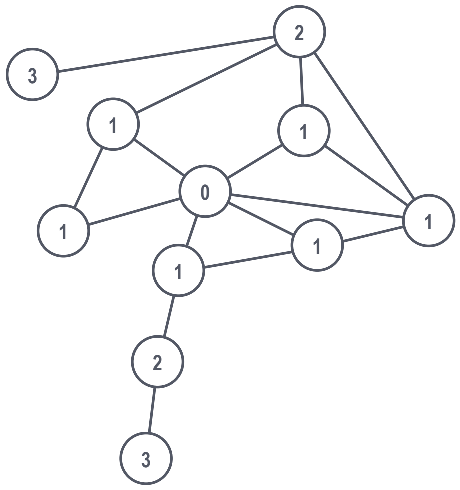


FIGURE 8.1: An example path of a breadth-first algorithm, where the search begins at the node marked "0" and then traverses first to every node marked "1" before journeying on to nodes marked "2" and then finally visiting all nodes marked "3."

There are two basic types of graph search algorithms: depth-first and breadth-first.

Dijkstra's Algorithm

The goal of Dijkstra's algorithm is to conduct a breadth-first search with a higher level of analysis in order to find the shortest path between two nodes in a graph. It does so in the following manner:

Pick the start and end nodes and add the start node to the set of solved nodes with a value of 0. Solved nodes are the set of nodes with the shortest known path from the start node. The start node has a value of 0 because it is 0 path-lengths away from itself.

Traverse breadth-first from the start node to its nearest neighbors and record the path length against each neighboring node.

Pick the shortest path to one of these neighbors and mark that node as solved. In case of a tie, Dijkstra's algorithm will pick at random.

Visit the nearest neighbors to the set of solved nodes and record the path lengths from the start node against these new neighbors. Don't visit any neighboring nodes that have already been solved, as we already know the shortest paths to them.

Repeat steps three and four until the destination node has been marked solved. Dijkstra's algorithm is very efficient as it works only with a smaller subset of the possible paths through a graph. After each node is solved, the shortest path from the start node is known and all subsequent paths build upon that knowledge.

Dijkstra's algorithm is often used to find real-world shortest paths, such as for navigation and logistics. Let's see how it would find the shortest driving route between Sydney and Perth in Australia.

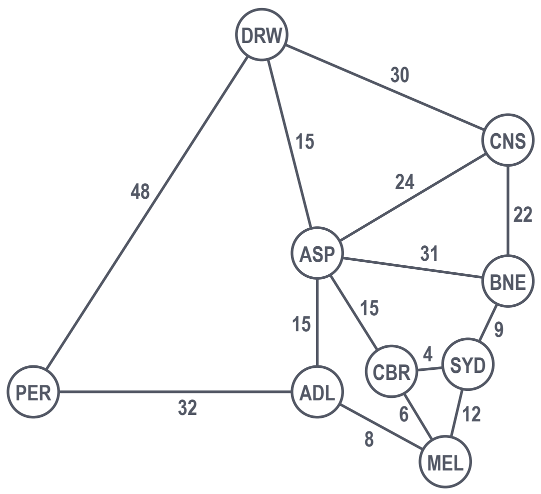


FIGURE 8.2: A graph of cities throughout Australia.

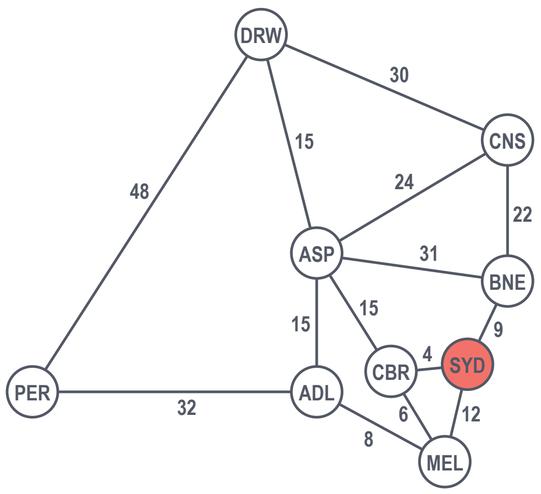


FIGURE 8.3: As the start node, Sydney has a value of 0 as we are already there.

Moving in a breadth-first manner, we look at the next cities one hop away from Sydney: Brisbane, Canberra and Melbourne.

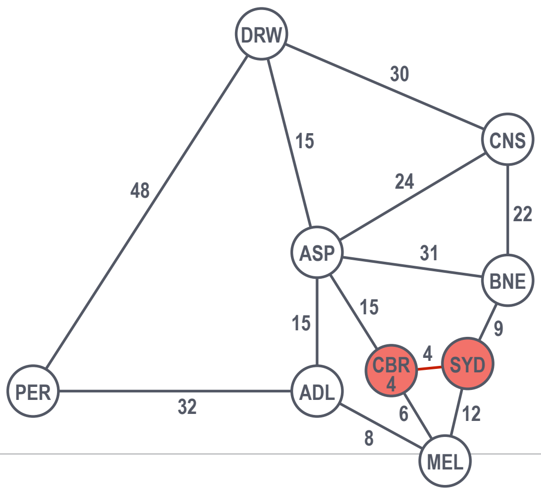


FIGURE 8.4: The algorithm has found the shortest path as being between Sydney and Canberra.

Canberra is the shortest path at 4 hours, so we count that as solved. We continue onto the next level, considering the next nodes out from our solved nodes and selecting the shortest paths. From there, Brisbane at 9 hours is the next solved node (Figure 8.5).

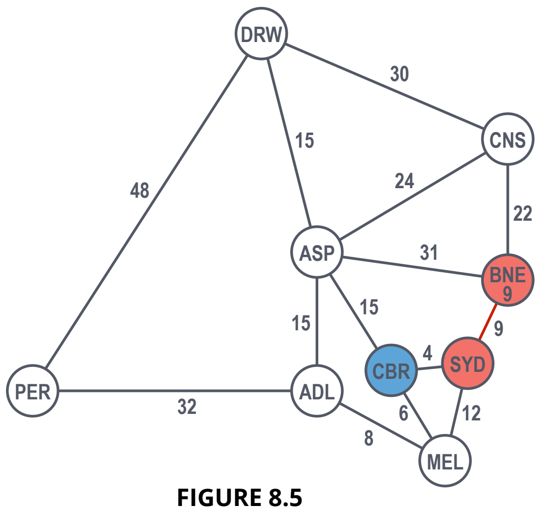


FIGURE 8.5

We move on, choosing between Melbourne, Cairns and Alice Springs. Melbourne is the shortest path at 10 hours from Sydney (via Canberra), so it becomes the next solved node (Figure 8.6).

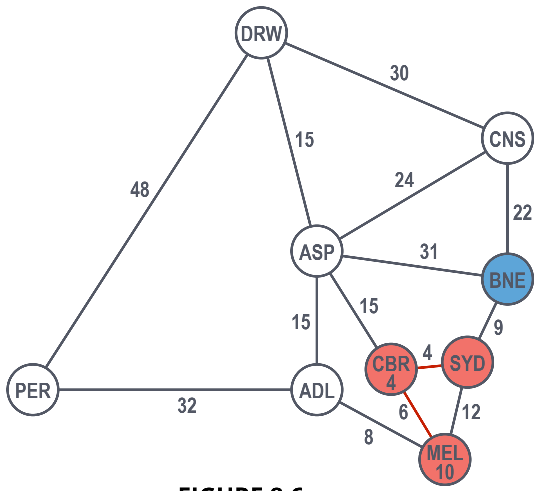


FIGURE 8.6

Our next few options of neighboring nodes are Adelaide, Cairns and Alice Springs. At 18 hours from Sydney (via Canberra and Melbourne), Adelaide is the next shortest path.

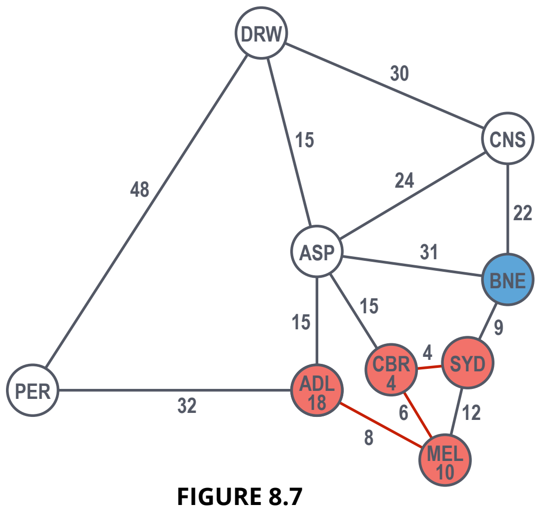


FIGURE 8.7

The next options are Perth (our final destination), Alice Springs and Cairns. While in reality it would be most efficient to head directly to Perth, according to Dijkstra's algorithm, Alice Springs is chosen because it has the current shortest path (19 total hours vs. 50 total hours).

Note that because this is a breadth-first search, Dijkstra's algorithm *must* first search all still-possible paths, not just the first solution that it happens across. **This principle is why Perth isn't immediately ruled out as the shortest path**.

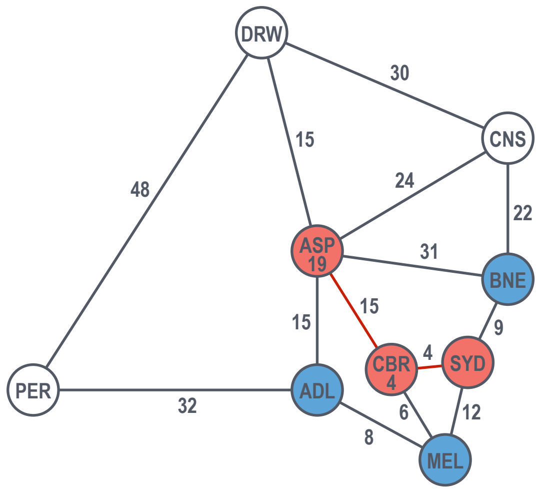


FIGURE 8.8: Dijkstra's algorithm continues to search all possible paths, even when it comes across one of many possible solutions.

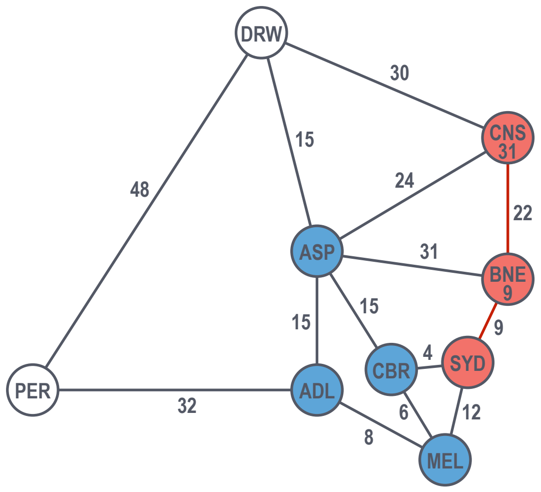


FIGURE 8.9: From Alice Springs, our two options are Darwin and Cairns. The latter is 31 hours to the former's 34 hours, so Cairns is the next solved node.

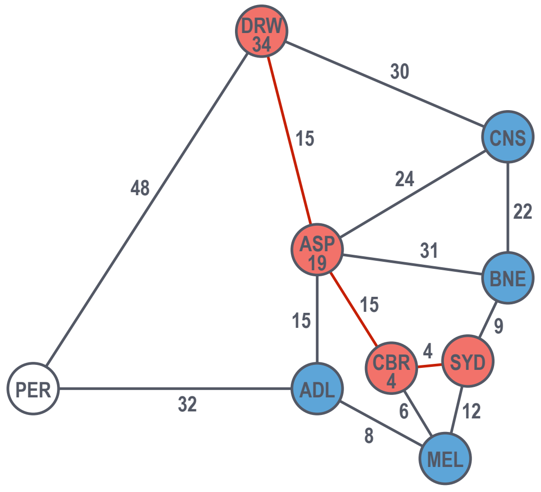


FIGURE 8.10: The path from Sydney to Darwin is a total of 34 hours.

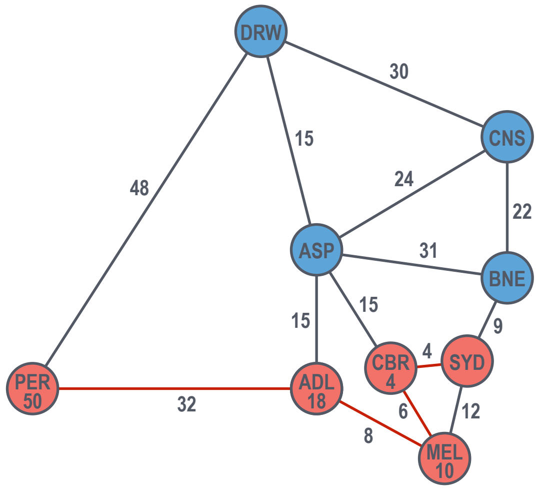


FIGURE 8.11: From Darwin, we examine the only remaining node on the graph: Perth. However, this given explored path to Perth via Darwin comes at a cost of 82 hours.

Now that Dijkstra's algorithm has solved for all possible paths, it can rightly compare the two routes to Perth via Adelaide at a cost of 50 hours, or via Darwin at a cost of 82 hours.

Accordingly, Dijkstra's algorithm would choose the route via Adelaide and consider Perth from Sydney solved at a shortest path of 50 hours.

In the end, we can see that the algorithm uses an undirected exploration to find its results. Occasionally, this causes us to explore more of the graph than is intuitively necessary, as Dijkstra's algorithm looks at each node in relative isolation and may end up following paths that do not contribute to the overall shortest path (like we saw above).

The example above could have been improved if the search had been guided by some heuristic like in a best-first search. To apply that in our own example, we could have chosen to prefer heading west over east and heading south over north, which would have helped avoid the unnecessary side trips taken.

Graph search algorithms are helpful in traversing a set of graph data and providing relevant information.

The A* Algorithm

Consequently, the A* algorithm improves upon Dijkstra's algorithm by combining some of its elements with elements of a best-first search. Pronounced "A-star", the A* algorithm is based on the observation that some searches are informed, which helps us make better choices over which paths to take through the graph. Like Dijkstra's algorithm, A* can search a large area of a graph, but like a best-first search, A* uses a heuristic to guide its search.

Additionally, while Dijkstra's algorithm prefers to search nodes close to the current starting point, a best-first search prefers nodes closer to the destination. A* balances the two approaches to ensure that at each level, it chooses the node with the lowest overall cost of the traversal. As demonstrated by the example in the previous section, it is possible for Dijkstra's algorithm to overlook a better route while trying to complete its search.

For more information on how the A* algorithm is used in practice, consult Chapter 7 of Graph Databases from O'Reilly Media.

Conclusion

As has been illustrated above, graph search algorithms are helpful in traversing a set of graph data and providing relevant information. However, they also have their limitations.

We have seen that there are many varieties of search algorithms, ranging from the more basic breadth-first and depth-first to uninformed and informed searches to Dijkstra's and the A* algorithms. Each has its own strengths and weaknesses; no one type is necessarily better than another.

Chapter 9
Why We Need NoSQL Databases

As the problems with SQL-based relational databases have become all too clear, there's been a meteoric rise in the popularity of a new family of data storage technologies known as NoSQL.

NoSQL is a cheeky acronym for Not Only SQL – or more confrontationally – No to SQL. But the term "NoSQL" only defines what these data stores are not, rather than what they are.

In this chapter, we'll discuss the many and motley world of NoSQL databases and why they've become so popular.

NoSQL is a cheeky acronym for Not Only SQL – or more confrontationally – No to SQL. But the term "NoSQL" only defines what these data stores are not, rather than what they are. In this chapter, we'll discuss the many and motley world of NoSQL databases and why they've become so popular.

The Diverse World of NoSQL Databases

NoSQL databases are a spectrum of data storage technologies that are more varied than they are similar so it's difficult to make sweeping generalizations about their characteristics.

In the following chapters, we'll explore a few types of NoSQL databases. Our tour will encompass the group collectively known as aggregate stores (highlighted in blue below), including key-value stores, column family stores and document stores, as well as the various types of graph databases (in green), which include property graphs, hypergraphs and RDF triple stores.

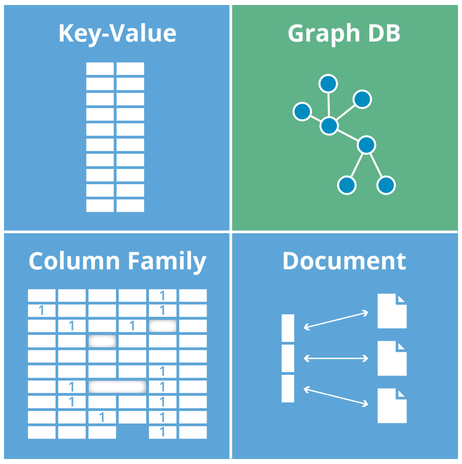


FIGURE 9.1: Aggregate data stores (blue) and graph databases (green).

Historically, most enterprise-level web applications ran on top of a relational database (RDBMS). But in the past decade alone, the data landscape has changed significantly and in a way that traditional RDBMS deployments simply can't manage.

The NoSQL database movement has emerged particularly in response to three of these data challenges: data volume, data velocity, and data variety.

We'll explore each of these challenges in further detail on the following page.

Data Volume

It's no surprise that as data storage has increased dramatically, data volume (i.e., the size of stored data) has become the principal driver behind the enterprise adoption of NoSQL databases.

Large datasets simply become too unwieldy when stored in relational databases. In particular, query execution times increase as the size of tables and the number of JOINs grow (so-called JOIN pain).

This isn't always the fault of the relational databases themselves though. Rather, it has to do with the underlying data model.

In order to avoid JOINs and JOIN pain, the NoSQL world has several alternatives to the relational model. While these NoSQL data models are better at handling today's larger datasets, most of them are simply not as expressive as the relational model. The only exception is the graph model, which is actually more expressive. (More on that in later chapters.)

Data Velocity

Volume isn't the only problem modern web-facing systems have to deal with. Besides being big, today's data often changes very rapidly.

Thus, ***data velocity*** (i.e., the rate at which data changes over time) is the next major challenge that NoSQL databases are designed to overcome.

Velocity is rarely a static metric. A lot of velocity measurements depend on the context of both internal and external changes to an application, some of which have considerable system-wide impact.

Coupled with high volume, variations in data velocity require a database to not only handle high levels of edits (tech lingo: write loads), but also deal with surging peaks of database activity. Relational databases simply aren't prepared to handle a sustained level of write loads and can crash during peak activity if not properly tuned.

But there's also another aspect of data velocity NoSQL technology helps us overcome: the rate at which the data ***structure*** changes. In other words, it's not just about the rapid change of specific data points but also the rapid change of the data model.

Data structures commonly shift for two major reasons. First is the fast-moving nature of business. As your enterprise changes, so do your data needs.

Second is that data acquisition is often experimental. Sometimes your application captures certain data points just in case you might need them later on. The data that proves valuable to your business usually sticks around, but if it isn't worthwhile, then those data points often fall by the wayside. Consequently, these experimental additions and eliminations affect your data model on a regular basis.

Both forms of data velocity are problematic for relational databases to handle. Frequent, high write loads come with expensive processing costs, and regular data structure changes come with high operational costs.

NoSQL databases address both data velocity challenges by optimizing for high write loads and by having flexible data models.

Data Variety

The final challenge in today's data landscape is data variety – that is, it can be dense or sparse, connected or disconnected, regularly or irregularly structured.

Today's data is far more varied than what relational databases were originally designed for. In fact, that's why many of today's RDBMS deployments have a number of nulls in their tables and null checks in their code – it's all to adjust to today's data variety.

On the other hand, NoSQL databases are designed from the bottom up to adjust for a wide diversity of data and flexibly address future data needs.

Conclusion

Relational databases can no longer handle today's data volume, velocity and variety. Yet, understanding how NoSQL databases overcome these challenges is only the prelude to finding the right database for your enterprise.

In the next few chapters, we'll explore the strengths and weaknesses of various NoSQL technologies so you can make the most informed decision possible.

Chapter 10
ACID vs. BASE Explained

When it comes to NoSQL databases, data consistency models can sometimes be strikingly different from those used by relational databases (as well as quite different from other NoSQL stores).

The two most common consistency models are known by the acronyms ACID and BASE. While they're often pitted against each other in a battle for ultimate victory, the fact remains that both consistency models come with advantages – and disadvantages – and neither is always a perfect fit.

The two most common consistency models are known by the acronyms ACID and BASE. While they're often pitted against each other in a battle for ultimate victory, the fact remains that both consistency models come with advantages – and disadvantages – and neither is always a perfect fit.

The following chapter discusses the key differences between ACID and BASE data consistency models and what their various trade-offs and advantages mean for your database transactions.

The ACID Consistency Model

Many developers are familiar with ACID transactions from working with relational databases. As such, the ACID consistency model has been the norm for some time.

The key ACID guarantee is that it provides a safe environment in which to operate on your data. The ACID acronym stands for:

Atomic
All operations in a transaction succeed or every operation is rolled back.

Consistent
On the completion of a transaction, the database is structurally sound.

Isolated
Transactions do not contend with one another. Contentious access to data is moderated by the database so that transactions appear to run sequentially.

Durable
The results of applying a transaction are permanent, even in the presence of failures.

ACID properties mean that once a transaction is complete, its data is consistent (tech lingo: write consistency) and stable on disk, which may involve multiple distinct memory locations.

Write consistency can be a wonderful thing for application developers, but it also requires sophisticated locking that is typically a heavyweight pattern for most use cases.

When it comes to NoSQL technologies, most graph databases (including Neo4j) use an ACID consistency model to ensure data is safe and consistently stored.

The BASE Consistency Model

For many domains and use cases, ACID transactions are far more pessimistic (i.e., they're more worried about data safety) than the domain actually requires.

In the NoSQL world, ACID transactions are less fashionable, as some databases have loosened the requirements for immediate consistency, data freshness and accuracy in order to gain other benefits, like scale and resilience.

(Notably, the .NET-based RavenDB has bucked the trend among aggregate stores in supporting ACID transactions.)

Given BASE's loose consistency, developers need to be more knowledgeable and rigorous about consistent data if they choose a BASE store for their application. It's essential to be familiar with the BASE behavior of your chosen aggregate store and work within those constraints.

Here's how the BASE acronym breaks down:

Basic Availability
The database appears to work most of the time.

Soft State
Stores don't have to be write-consistent, nor do different replicas have to be mutually consistent all the time.

Eventual Consistency
Stores exhibit consistency at some later point (e.g., lazily at read time).

BASE properties are much looser than ACID guarantees, but there isn't a direct one-for-one mapping between the two consistency models.

A BASE datastore values availability (since that's important for scale), but it doesn't offer guaranteed consistency of replicated data at write time. Overall, the BASE consistency model provides a less strict assurance than ACID: data will be consistent in the future, either at read time (e.g., Riak) or it will always be consistent, but only for certain processed past snapshots (e.g., Datomic).

The BASE consistency model is primarily used by aggregate stores, including column family, key-value and document stores.

Navigating ACID vs. BASE Trade-offs

There's no right answer to whether your application needs an ACID versus BASE consistency model. Developers and data architects should select their data consistency trade-offs on a case-by-case basis – not based just on what's trending or what model was used previously.

Given BASE's loose consistency, developers need to be more knowledgeable and rigorous about consistent data if they choose a BASE store for their application. It's essential to be familiar with the BASE behavior of your chosen aggregate store and work within those constraints.

On the other hand, planning around BASE limitations can sometimes be a major disadvantage when compared to the simplicity of ACID transactions. A fully ACID database is the perfect fit for use cases where data reliability and consistency are essential.

In the next two chapters we'll dive into more ACID/BASE specifics when it comes to aggregate stores and other graph technologies.

Chapter 11
A Tour of Aggregate Stores

Aggregate Stores and the World of NoSQL Databases

The group of NoSQL databases collectively known as aggregate stores (term coined by Martin Fowler) includes key-value stores, column family stores and document stores, which are all highlighted in blue below. (In Chapter 12, we'll examine the various types of graph technologies, which are another facet of NoSQL.)

It's worth noting that aggregate stores eschew connections between aggregates – only graph databases fully capitalize on data relationships.

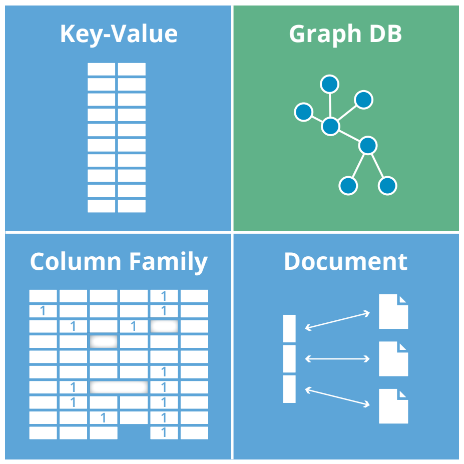


FIGURE 11.1: Aggregate data stores (blue) and graph databases (green).

In the following sections, we'll explore each of these three blue quadrants, highlighting the characteristics of each data model, operational aspects and the main drivers for adoption.

Key Value Stores

Key-value stores are large, distributed hashmap data structures that store and retrieve values organized by identifiers known as keys.

Here's a diagram of an example key-value store.

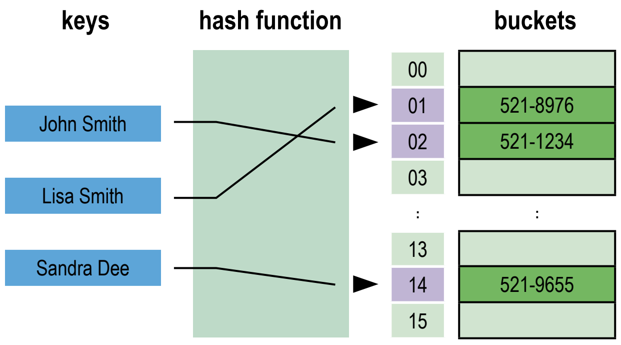


FIGURE 11.2: A basic key-value store. Source: Jorge Stolfi.

The group of NoSQL databases collectively known as aggregate stores includes key-value stores, column family stores and document stores.

In theory, key-value stores simply concern themselves with efficient storage and retrieval of data, unencumbered by its nature or usage. But this approach has its downsides: When extracting data from a stored value, applications often have to retrieve the entire value (which can be quite large) and then filter out any unwanted elements, which can be inefficient.

As you can see, a bucket contains a specific number of values, and for fault-tolerance reasons, each bucket is replicated onto several machines. However, machines should never be exact copies of one another – not only for data replication purposes but also for better load balancing.

An application wishing to store or retrieve data in a key-value store only needs to know (or compute) the corresponding key, which can be as natural as a username, an email address, Cartesian coordinates, a Social Security number or a ZIP code. With a sensibly designed system, the chance of losing data due to a missing key is low.

In theory, key-value stores simply concern themselves with efficient storage and retrieval of data, unencumbered by its nature or usage. But this approach has its downsides: When extracting data from a stored value, applications often have to retrieve the entire value (which can be quite large) and then filter out any unwanted elements, which can be inefficient.

Although simple, the key-value model doesn't offer much insight into data relationships. In order to retrieve sets of information across several records, you typically need to conduct external processing with an algorithm like MapReduce, often producing highly latent results.

However, key-value stores do have certain advantages. Since they're descended from Amazon's DynamoDB, they are optimized for high availability and scale. Or, as the Amazon team puts it, they should work even "if disks are failing, network routes are flapping or data centers are being destroyed by tornadoes."

Column Family Stores

Column family stores (also known as wide-column stores) are based on a sparsely populated table whose rows can contain arbitrary columns and where keys provide for natural indexing.

(Note: In the explanation below, we'll use terminology from Apache Cassandra since it is one of the most popular column family stores.)

In the diagram below, you can see the four building blocks of a column family database.

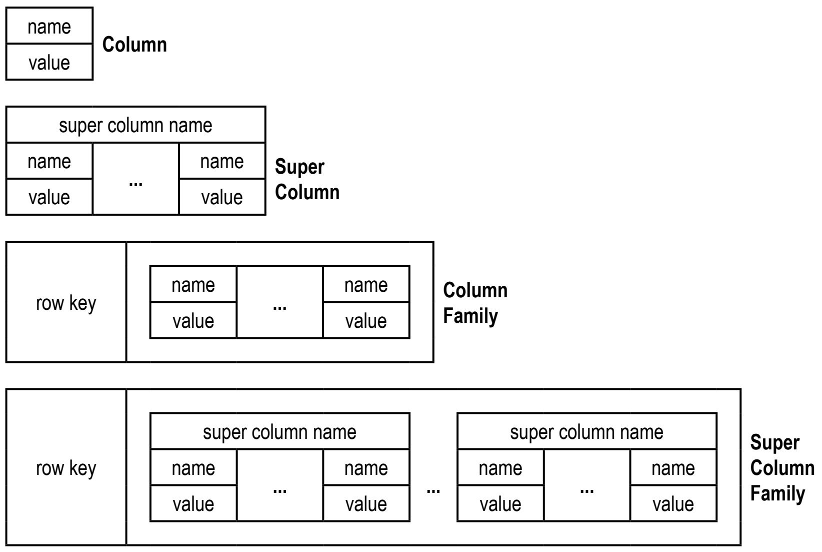


FIGURE 11.3: The four building blocks of a column family database.

The simplest unit of storage is the column itself consisting of a name-value pair. Any number of columns can then be combined into a super column, which gives a name to a particular set of columns. Columns are stored in rows, and when a row contains columns only, it is known as a column family, but when a row contains super columns, it is known as a super column family.

At first it might seem odd to include rows when the data is mostly organized via columns, but in fact, rows are vital since they provide a nested hashmap for columnar data. Consider the diagram below of a super column family mapping out a recording artist and his albums.

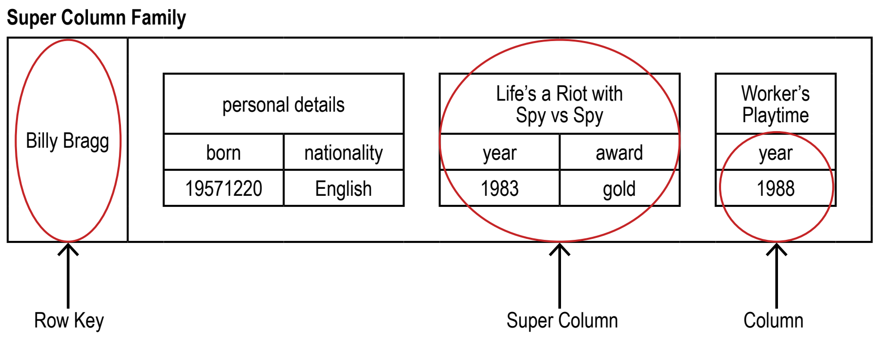


FIGURE 11.4: A super column family mapping out a recording artist and his albums.

In a column family database, each row in the table represents a particular overarching entity (e.g., everything about an artist). These column families are containers for related pieces of data, such as the artist's name and discography. Within the column families, we find actual key-value data, such as album release dates and the artist's date of birth.

Here's the kicker: This row-oriented view can also be turned 90 degrees to arrive at a column-oriented view. Where each row gives a complete view of one entity, the column view naturally indexes particular aspects across the whole dataset.

For example, let's look at the figure below:

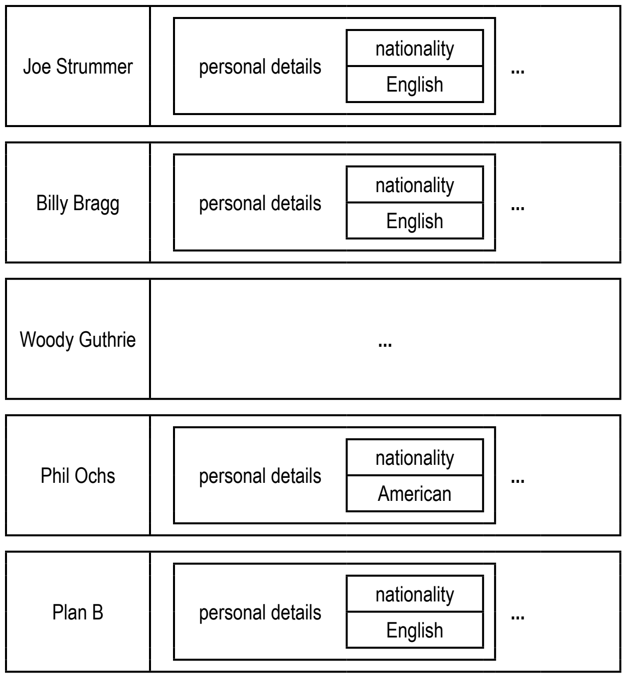


FIGURE 11.5: Keys form a natural index through rows in a column family database.

In a column family database, each row in the table represents a particular overarching entity (e.g., everything about an artist). These column families are containers for related pieces of data, such as the artist's name and discography.

It's no surprise that as data storage has increased dramatically, data volume (i.e., the size of stored data) has become the principal driver behind the enterprise adoption of NoSQL databases.

As you can see, by "lining up" keys we can find all the rows where the artist is English. From there it's easy to extract complete artist data from each row. It's not the same as the connected data we'd find in a graph, but it does provide some insight into related entities.

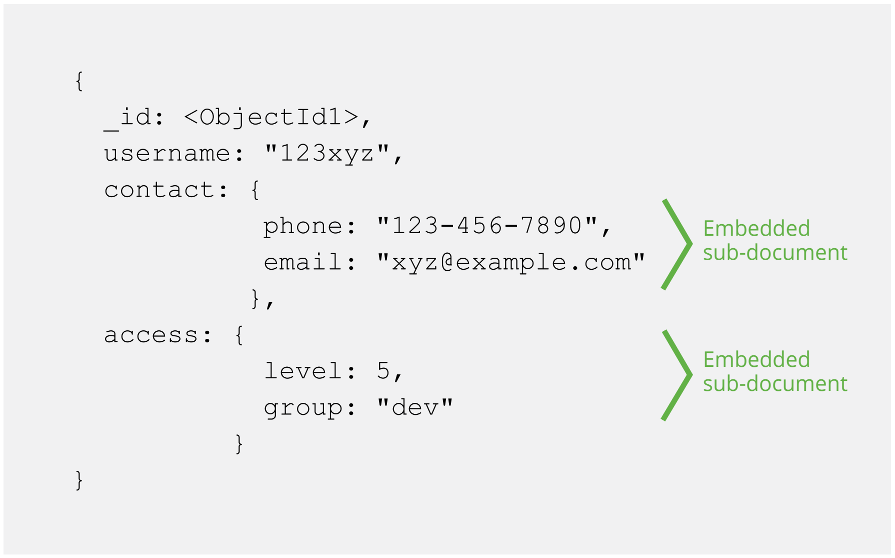
Column family databases are distinguished from document and key-value stores not only by their more expressive data model, but also by their architecture built for distribution, scale and failover. And yet they're still aggregate stores and, as such, lack JOINs or first-class data relationships.

Document Stores

Put simply, document databases store and retrieve documents just like an electronic filing cabinet. Documents can include maps and lists, allowing for natural hierarchies. In fact, document stores are most familiar to developers who are used to working with hierarchically structured documents.

At the most basic level, documents are stored and retrieved by ID. If an application remembers the IDs it's most interested in (such as usernames), then a document store acts much like a key-value store.

The document model usually involves having a hierarchical JSON document as the primary data structure, and any field inside of the hierarchy can then be indexed. For example, in the diagram to the right, the embedded sub-documents are part of the larger user document.



```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
              phone: "123-456-7890",
              email: "xyz@example.com"
             },
  access: {
              level: 5,
              group: "dev"
            }
}
```

FIGURE 11.6: Embedded data in a document store.
Source: MongoDB Data Model Design documentation

Because document stores have a data model around disconnected entities, their major advantage is horizontal scaling. However, most document databases require developers to explicitly plan for sharding of data across instances to support this horizontal scale while key-value stores and column family stores don't require this extra step.

To see an example of how MongoDB – one of the most popular document stores – integrates with Neo4j, check out the Wanderu case study.

Query Versus Processing in NoSQL Aggregate Stores

On balance, the similarities between NoSQL aggregate stores are greater than the differences. While each has a different storage strategy, they all share similar characteristics when it comes to querying data.

For simple queries, aggregate stores use indexing, basic document linking or a query language.

However, for more complex queries, aggregate stores cannot generate deeper insights simply by examining individual data points. To compensate, an application typically has to identify and extract a subset of data and run it through an external processing infrastructure such as the MapReduce framework (often in the form of Apache Hadoop).

MapReduce is a parallel programming model that splits data and operates on it in parallel before gathering it back together and aggregating it to provide focused information.

Graph databases, on the other hand, embrace relationships in order to solve problems that involve context and connectedness. Consequently, they have very different design principles and a different architectural foundation.

For example, if we wanted to use MapReduce to count the number of Americans there are in a recording artists database, we'd need to extract all artist data and discard the non-American ones in the map phase. Then, we'd count the remaining records in the reduce phase.

But even with a lot of machines and a fast network infrastructure, MapReduce can be quite latent. So latent, in fact, that often a development team needs to introduce new indexes or ad hoc queries in order to focus (and trim) the dataset for better MapReduce speeds.

Conclusion

Aggregate stores are good at storing big sets of discrete data, but they do that by sacrificing a data model, language and functionality for handling data relationships.

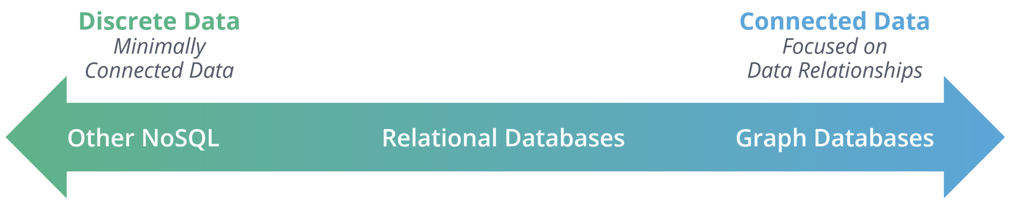


FIGURE 11.7: The spectrum of databases for discrete versus connected data.

If you try to use aggregate stores for interrelated data, it results in a disjointed development experience since you have to add a lot of code to fill in where the underlying aggregate store leaves off. And as the number of hops (or "degrees" of the query) increases, aggregate stores slow down significantly.

Graph databases, on the other hand, embrace relationships in order to solve problems that involve context and connectedness. Consequently, they have very different design principles and a different architectural foundation.

Do aggregate stores have their perfect use cases? Certainly. But they aren't for dealing with problems that require an understanding of how things are connected.

Chapter 12
Other Graph Database Technologies

Whether you're new to the world of graph databases or an old pro, it's easy to assume there's only a few types of graph database technologies.

In reality, it's one of the most diverse sectors of the NoSQL ecosystem. In this chapter, we'll discuss the spectrum of graph database technologies and where they belong in the world of NoSQL.

Review: The NoSQL Matrix

The macrocosm of NoSQL databases is a diverse one of which graph databases are only a part. In Chapter 11, we toured the three blue quadrants of the matrix, which are collectively known as aggregate stores, including key-value, column family and document stores.

Now we'll be double-clicking on the equally diverse world of graph database technologies, which occupy the green quadrant in the matrix to the right.

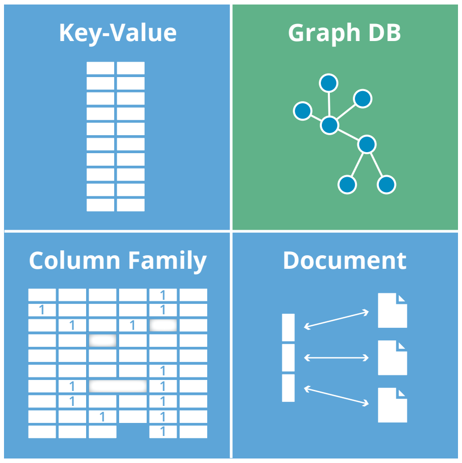


FIGURE 12.1: Aggregate data stores (blue) and graph databases (green).

The Spectrum of Graph Database Technologies

We already walked through a formal definition of a graph database in Chapter 1, but let's do a quick review.

A graph database is an online, operational database management system capable of Create, Read, Update, and Delete (tech lingo: CRUD) processes that operate on a graph data model.

There are two important properties of graph database technologies:

Graph Storage

Some graph databases use "native" graph storage that is specifically designed to store and manage graphs, while others use relational or object-oriented databases, which are often slower.

Graph Processing Engine

Native graph processing (tech lingo: index-free adjacency) is the most efficient means of processing data in a graph because connected nodes physically "point" to each other in the database. Non-native graph processing engines use other means to process CRUD operations. For more information on native vs. non-native graph technology, see Chapter 13.

Besides specifics around storage and processing, graph databases also adopt distinct data models. The most common graph data models include property graphs, hypergraphs and triples. Let's dive into each of these below.

Property Graphs

Property graphs are the type of graph database we've already talked about most. In fact, our original definition of a graph database was more precisely about a property graph.

Here's a quick recap of what makes a graph database a property graph (it's worth noting that Neo4j is a property graph database):

- Property graphs contain nodes (data entities) and relationships (data connections).
- Nodes can contain properties (tech lingo: key-value pairs).
- Nodes can be labeled with one or more labels.
- Relationships have both names and directions.
- Relationships always have a start node and an end node.

- Like nodes, relationships can also contain properties.

Hypergraphs

A hypergraph is a graph model in which a relationship (called a hyperedge) can connect any number of given nodes. While a property graph permits a relationship to have only one start node and one end node, the hypergraph model allows any number of nodes at either end of a relationship.

Hypergraphs can be useful when your data includes a large number of many-to-many relationships. Let's look at Figure 12.2.

In this simple (directed) hypergraph, we see that Alice and Bob are the owners of three vehicles, but we can express this relationship using a single hyperedge. In a property graph, we would have to use six relationships to express the concept.

In theory, hypergraphs should produce accurate, information-rich data models. However, in practice, it's very easy for us to miss some detail while modeling. For example, let's look at the figure below, which is the property graph equivalent of the hypergraph shown in Figure 12.3.

This property graph model requires several OWNS relationships to express what the hypergraph captured with just one hyperedge. Yet, by using six relationships instead of one, we have two distinct advantages:

1. We're using a more familiar and explicit data modeling technique (resulting in less confusion for a development team)

2. We can also fine-tune the model with properties such as "primary driver" (for insurance purposes), which is something we can't do with a single hyperedge.

Because hyperedges are multidimensional, hypergraph models are more generalized than property graphs. Yet, the two are isomorphic, so you can always represent a hypergraph as a property graph (albeit with more relationships and nodes).

While property graphs are widely considered to have the best balance of pragmatism and modeling efficiency, hypergraphs show their particular strength in capturing meta-intent. For example, if you need to qualify one relationship with another (e.g., I like the fact that you liked that car), then hypergraphs typically require fewer primitives than property graphs.

Whether a hypergraph or a property graph is best for you depends on your modeling mindset and the types of applications you're building.

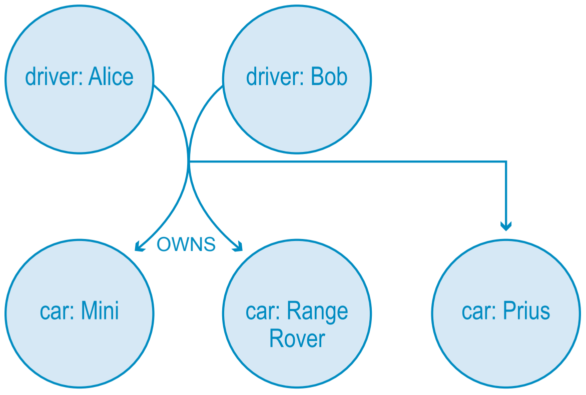


FIGURE 12.2: A directed hypergraph of the cars owned by Alice and Bob.

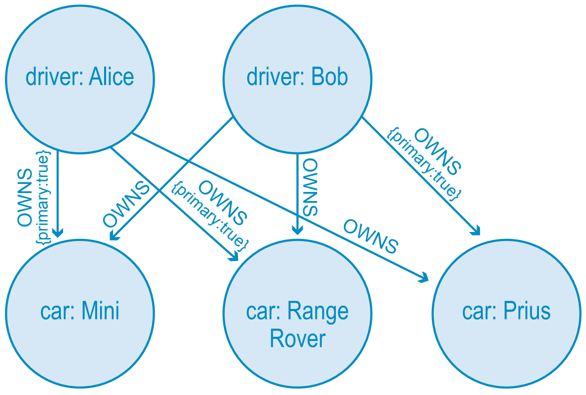


FIGURE 12.3: A property graph model of the cars owned by Alice and Bob.

Triple Stores

Triple stores come from the Semantic Web movement and store data in a format known as a triple. Triples consist of a subject-predicate-object data structure.

Using triples, we can capture facts such as "Ginger dances with Fred" and "Fred likes ice cream." Individually, single triples aren't very useful semantically, but en masse, they provide a rich dataset from which to harvest knowledge and infer connections.

Triple stores are modeled around the Resource Description Framework (RDF) specifications laid out by the W3C, using SPARQL as their query language.

Data processed by triple stores tends to be logically linked, thus triple stores are included in the category of graph databases. However, triple stores are not "native" graph databases because they don't support index-free adjacency, nor are their storage engines optimized for storing property graphs.

Triple stores store triples as independent elements, which allows them to scale horizontally but prevents them from rapidly traversing relationships. In order to perform graph queries, triple stores must create connections from individual, independent facts – adding latency to every query.

Because of these trade-offs in scale and latency, the most common use case for triple stores is offline analytics rather than for online transactions.

Conclusion

Just like for other NoSQL databases, every type of graph database is best suited for a different function. Hypergraphs are a good fit for capturing meta-intent and RDF triple stores are proficient at offline analytics. But for online transactional processing, nothing beats a property graph for a rapid traversal of data connections.

Just like for other NoSQL databases, every type of graph database is best suited for a different function. Hypergraphs are a good fit for capturing meta-intent and RDF triple stores are proficient at offline analytics. But for online transactional processing, nothing beats a property graph for a rapid traversal of data connections.

There are two main elements that distinguish native graph technology: storage and processing. Graph storage commonly refers to the underlying structure of the database that contains graph data. Native graph processing is another key element of graph technology, referring to how a graph database processes database operations, including both storage and queries.

Chapter 13
Native vs. Non-Native Graph Processing

No technology is equally good at everything, and databases are no exception. It's possible for databases to satisfy different kinds of functions: batch and transactional workloads, memory access and disk access, SQL and XML access, and graph and document data models.

When building a database management system (DBMS), development teams must decide early on what cases to optimize for, which will dictate how well the DBMS will handle the tasks it is dealt (what the DBMS will be amazing at, what it will be okay at, and what it may not do so well). As a result, the graph database world is populated with both technology designed to be "graph first," known as native, and technology where graphs are an afterthought, classified as non-native.

There's a considerable difference when it comes to the native architecture of both graph storage and processing. Unsurprisingly, native technologies tend to perform queries faster, scale bigger (retaining their hallmark query speed as the dataset grows in size) and run more efficiently, calling for much less hardware. As a result, it's critical to understand the differences.

Now that we have a relatively comprehensive grasp of the basics from earlier chapters in this ebook, it is time to go over the differing internal properties of a graph database. In this chapter, we will discuss some of the characteristics that distinguish native graph databases and why these characteristics are of interest to graph database users.

What "Graph First" Means for Native Graph Technology

There are two main elements that distinguish native graph technology: storage and processing.

Graph storage commonly refers to the underlying structure of the database that contains graph data. When built specifically for storing graph-like data, it is known as native graph storage. Graph databases with native graph storage are optimized for graphs in every aspect, ensuring that data is stored efficiently by writing nodes and relationships close to each other.

Graph storage is classified as non-native when the storage comes from an outside source, such as a relational, columnar or other NoSQL database. These databases use other algorithms to store data about nodes and relationships, which may end up being placed far apart. This non-native approach can lead to latent results as their storage layer is not optimized for graphs.

Native graph processing is another key element of graph technology, referring to how a graph database processes database operations, including both storage and queries. Index-free adjacency is the key differentiator of native graph processing.

At write time, index-free adjacency speeds up storage processing by ensuring that each node is stored directly to its adjacent nodes and relationships. Then, during query processing (i.e., read time), index-free adjacency ensures lightning-fast retrieval without the need for indexes. Graph databases that rely on global indexes (rather than index-free adjacency) to gather results are classified as having non-native processing.

Another important consideration is ACID writes. Related data brings an uncommonly strict need for data integrity beyond that of other NoSQL models. In order to store a connection between two things, we must not only write a relationship record but update the node at each end of the relationship as well. If any one of these three write operations fails, it will result in a corrupted graph.

The only way to ensure that graphs aren't corrupted over time is to carry out writes as fully ACID transactions. Systems with native graph processing include the proper internal guard rails to ensure that data quality remains impervious to network blips, server failures, competing transactions and the like.

Native Graph Storage

To dive into further detail, the element that makes a graph storage native is the structure of the graph database from the ground up. Graph databases with native graph storage have underlying storage designed specifically for the storage and management of graphs. They are designed to maximize the speed of traversals during arbitrary graph algorithms.

For example, let's take a look at the way Neo4j – a native graph database – is structured for native graph storage. Every layer of this architecture – from the Cypher query language to the files on disk – is optimized for storing graph data, and not a single part is substituted with other non-graph technologies.

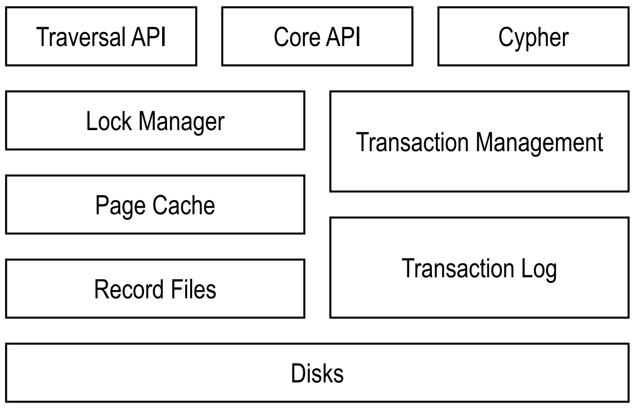


FIGURE 13.1: The Neo4j Structure for native graph storage.

Graph data is kept in store files, each of which contain data for a specific part of the graph, such as nodes, relationships, labels and properties. Dividing the storage in this way facilitates highly performant graph traversals.

In a native graph database, a node record's main purpose is to simply point to lists of relationships, labels and properties, making it quite lightweight.

So, what makes non-native graph storage different from a native graph database?

Non-native graph storage uses a relational database, a columnar database or some other general-purpose data store rather than being specifically engineered for the uniqueness of graph data. While the typical operations team might be more familiar with a non-graph backend (like MySQL or Cassandra), the disconnect between graph data and non-graph storage results in a number of performance and scalability concerns.

Non-native graph databases are not optimized for storing graphs, so the algorithms utilized for writing data may store nodes and relationships all over the place. This causes performance problems at the time of retrieval because all these nodes and relationships then have to be reassembled for every single query.

On the other hand, native graph storage is built to handle highly interconnected datasets from the ground up and is therefore the most efficient when it comes to the storage and retrieval of graph data.

Since graph databases store relationship data as first-class entities, relationships are easier to traverse in any direction with native graph processing. With processing that is specifically built for graph datasets, relationships – rather than over-reliance on indexes – are used to maximize the efficiency of traversals.

Native Graph Processing

A graph database has native processing capabilities if it uses index-free adjacency. This means that each node directly references its adjacent nodes, acting as a micro-index for all nearby nodes. Index-free adjacency is more efficient and cheaper than using global indexes, as query times are proportional to the amount of the graph searched, rather than increasing with the overall size of the data.

Since graph databases store relationship data as first-class entities, relationships are easier to traverse in any direction with native graph processing. With processing that is specifically built for graph datasets, relationships – rather than over-reliance on indexes – are used to maximize the efficiency of traversals.

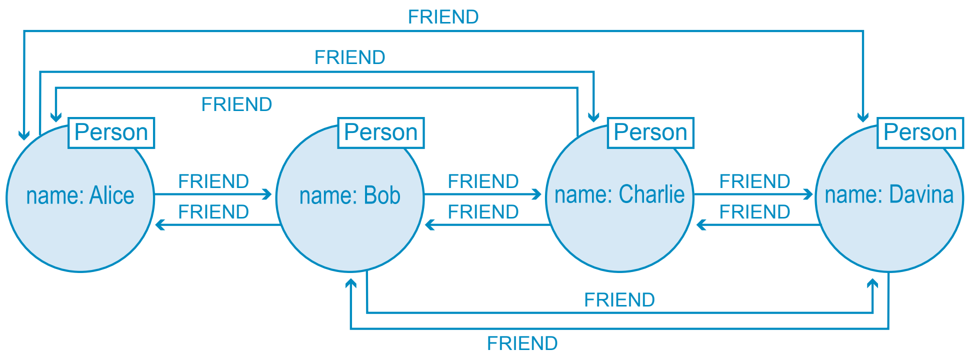
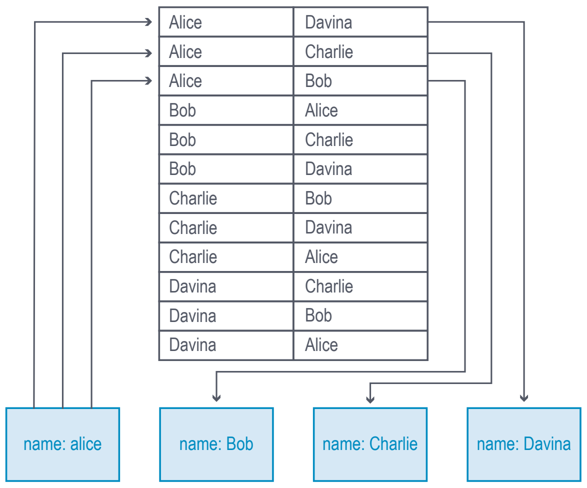


FIGURE 13.2: Because graph databases store relationship data as first-class entities, relationships are easier to traverse in any direction with native graph processing.

On the other hand, non-native graph databases use global indexes to link nodes together. This method is more costly, as the indexes add another layer to each traversal, which slows processing considerably.

First of all, using a global index lookup is already far more expensive. Queries with more than one layer of connection further reduce traversal performance with non-native graph processing.



Alice	Davina
Alice	Charlie
Alice	Bob
Bob	Alice
Bob	Charlie
Bob	Davina
Charlie	Bob
Charlie	Davina
Charlie	Alice
Davina	Charlie
Davina	Bob
Davina	Alice

FIGURE 13.3: An example global index look-up.

In addition, reversing the direction of a traversal is extremely difficult with non-native graph processing. To reverse a traversal's direction, you must either create a costly reverse-lookup index for each traversal, or perform a brute-force search through the original index.

The Bottom Line: Why Native vs. Non-Native Matters

When deciding between a native and non-native graph databases, it is important to understand the tradeoffs of working with each.

Non-native graph technology most likely has a persistence layer that your development team is already familiar with (such as Cassandra, MySQL or another relational database), and when your dataset is small or less connected, choosing non-native graph technology isn't likely to significantly affect the performance of your application.

However, it's important to note that datasets tend to grow over time, and today's datasets are more unstructured, interconnected and interrelated than ever before. Even if your dataset is small to begin with, it's essential to plan for the future if your data is likely to grow alongside your business. In this case, a native graph database will serve you better over the long term since the performance of non-native graph processing cripples under larger datasets.

One of the biggest drivers behind moving to a native graph architecture is that it scales. As you add more data to the database, many queries that would slow with size in a non-native graph database remain lithe and speedy in a native context.

Native graph scaling takes advantage of a large number of optimizations in storage and processing to yield a highly efficient approach, whereas non-native uses brute force to solve the problem, requiring more hardware (usually two to four times the amount of hardware or more) and resulting in lower latencies, especially for larger graphs.

Not all applications require low latency or processing efficiency, and in those use cases, a non-native graph database might just do the job. But if your application requires storing, querying and traversing large interconnected datasets in real time for an always-on, mission-critical application, then you need a database architecture specifically designed for handling graph data at scale.

The bottom line: The importance of native vs. non-native graph technology depends on the particular needs of your application, but for enterprises hoping to leverage the connections in their data like never before, the performance, efficiency and scaling advantages of a native graph database are crucial for success.

The importance of native vs. non-native graph technology depends on the particular needs of your application, but for enterprises hoping to leverage the connections in their data like never before, the performance, efficiency and scaling advantages of a native graph database are crucial for success.

Whether you need a solution that provides real-time recommendations, graph-based search or supply-chain management, be sure to review all the different ways in which graph technology can work for your company.

Conclusion
Making the Switch to Graph Technology

Now that you've made it through this high-level introduction to graph databases, you've learned what makes graphs stand apart from the typical NoSQL aggregate stores and RDBMS technology. And while such databases serve important functions in the wide world of data, there are times when only graph technoloy holds the answers to the questions you, your company and your customers need answered.

Why? Because of its central focus on connections combined with the ability to handle increasingly-growing volumes and varieties of data at mind-boggling speeds. Together, this means that no matter how much your business or data model changes, your database will continue working for you.

Whether you need a solution that provides real-time recommendations, graph-based search or supply-chain management, be sure to review all the different ways in which graph technology can work for your company.

And while our customers span several continents and professional fields, they all agree that using the Neo4j graph database is a critical component of their business success and competitiveness.

Are you a developer eager to learn more about making the switch? With so many ways to quickly get started, mastering graph database development is one of the best time investments you can make.

Other Resources

Videos:

- Intro to Neo4j and Graph Databases
- Intro to Graph Databases Episode #1 - Evolution of DBs

Books:

- O'Reilly book: Graph Databases
- Learning Neo4j

Trainings:

- Online Training: Getting Started with Neo4j
- Classroom Trainings

Questions about Neo4j?

Contact us:
1-855-636-4532
info@neo4j.com