

GRAPH DATABASES

NOSQL DATABASE SOLUTION FOR MANAGING LINKED DATA

eMag Issue 34 - October 2015



InfoQ
new

ARTICLE

Full Stack Web
Development Using
Neo4j

INTERVIEW

Data Modeling in
Graph Databases

ARTICLE

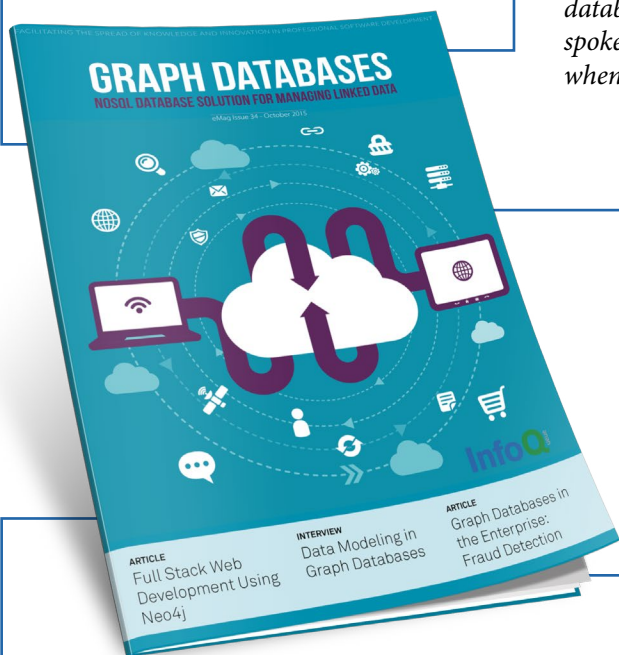
Graph Databases in
the Enterprise:
Fraud Detection

Let Me Graph That For You

In this article on Graph Databases, author Ian Robinson discusses the problems Graph DBs aim to solve. He also talks about the data, storage, and query models for managing graph data.

Data Modeling in Graph Databases: Interview with Jim Webber and Ian Robinson

Data modeling with Graph databases requires a different paradigm than modeling in Relational or other NoSQL databases like Document databases, Key Value data stores, or Column Family databases. InfoQ spoke with Jim Webber and Ian Robinson about data modeling efforts when using Graph databases.



High Tech, High Sec.: Security Concerns in Graph Databases

Graph NoSQL databases support data models with connected data and relationships. In this article, the author discusses the security implications of graph database technology. He talks about the privacy and security concerns in use cases like graph discovery, knowledge management, and prediction.

Graph Databases in the Enterprise: Fraud Detection

Financial services organizations lose billions of dollars every year to fraud transactions. Graph NoSQL databases can be used to uncover fraud rings and other complex scams using contextual link analysis and potentially stop advanced fraud scenarios in real time. Checkout this article for a detailed discussion of one of the most impactful and interesting use cases of graph database technologies, fraud detection.

Full Stack Web Development Using Neo4j

When building a web application there are a lot of choices for the database. In this article, author discusses why Neo4j Graph database is a good choice as a data store for your web application if your data model contains lot of connected data and relationships.

FOLLOW US



facebook.com
/InfoQ



@InfoQ



google.com
/+InfoQ



linkedin.com
company/infoq

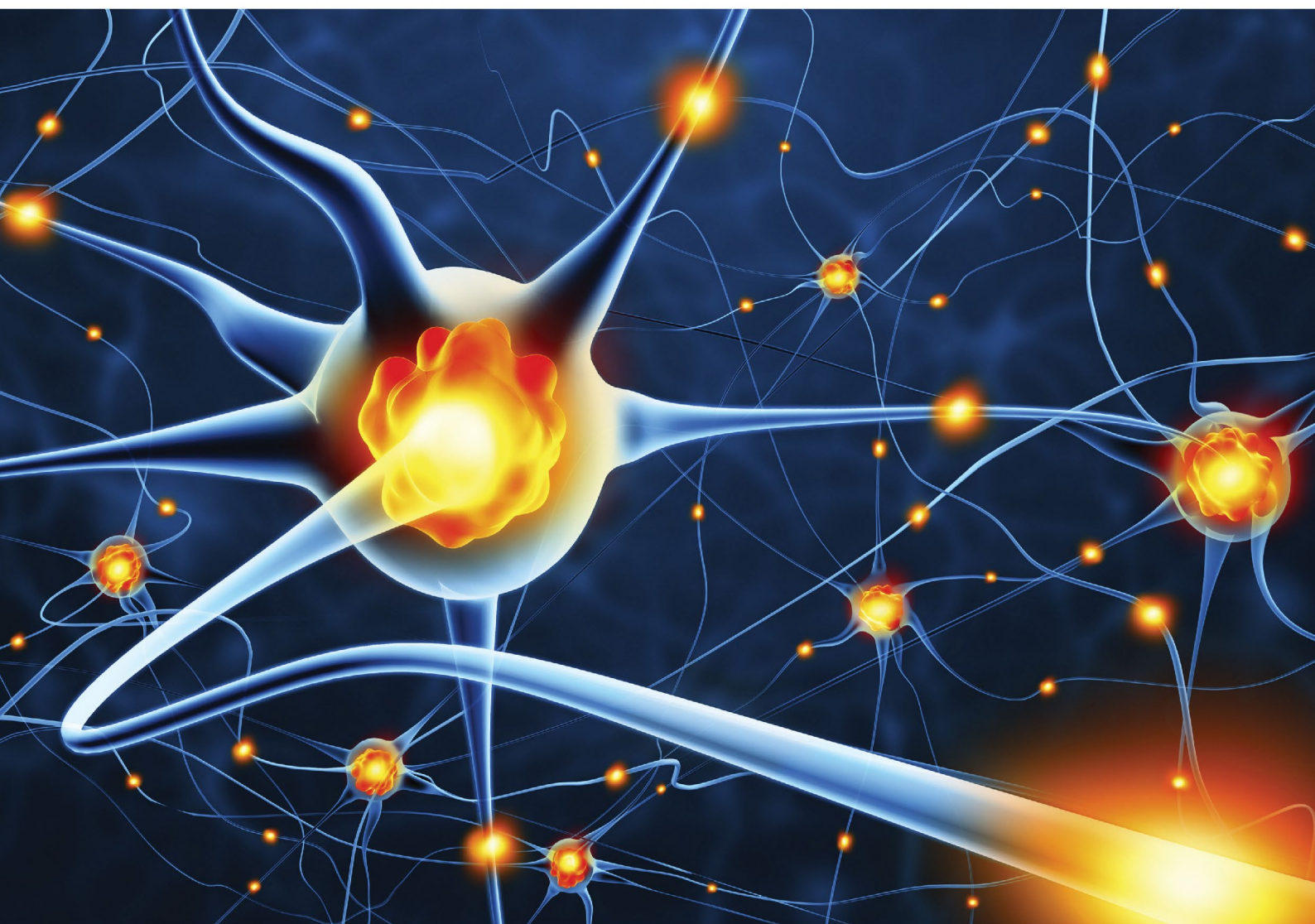
CONTACT US

GENERAL FEEDBACK feedback@infoq.com

ADVERTISING sales@infoq.com

EDITORIAL editors@infoq.com

Your Brain Makes Connections Naturally.



Shouldn't Your Database?

As the world's leading graph database, Neo4j is the natural choice of enterprises worldwide for its flexibility, performance and scalability. Neo4j's native graph processing and storage delivers real-time query performance with zero latency, so it's perfect for building applications in today's ever-connected world.

Best of all, Neo4j connects relationships in your data automatically, making your applications smarter than ever.

What can you build with the world's most powerful graph database? Imagine the possibilities.



[Download Neo4j Today](#)



**SRINI
PENCHIKALA** currently works as Senior Software Architect at a financial services organization in Austin, Texas. He is also the Lead Editor for Big Data and NoSQL Database communities at InfoQ. Srinipen has over 20 years of experience in software architecture, design and development. He is currently authoring a book on Big Data Processing with Apache Spark. He is also the co-author of "Spring Roo in Action" book from Manning Publications. Srinipen has presented at conferences like Big Data Conference, Enterprise Data World, JavaOne, SEI Architecture Technology Conference (SATURN), IT Architect Conference (ITARC), No Fluff Just Stuff, NoSQL Now and Project World Conference. He also published several articles on software architecture, security and risk management, and NoSQL databases on websites like InfoQ, The ServerSide, O'Reilly Network (ONJava), DevX Java, java.net and JavaWorld.

A LETTER FROM THE EDITOR

Graph databases are getting lot of attention lately. They are used to manage connected data and are better solutions for several real-world use cases which require the mapping of relationships between data entities for data driven business decision making.

Real-world use cases of graph databases include fraud detection, Anti-Money Laundering (AML) and Master Data Management (MDM), Trading Analytics, and Online Gaming.

The graph data management space includes three different areas:

- 1 Specialized Graph databases to store the connected data. These databases include Neo4J, TitanDB, InfiniteGraph, and AllegroGraph.
- 2 Graph data processing and real-time graph analytics frameworks like Spark GraphX and GraphLab.
- 3 Graph data visualization tools which give non-technical users insights into the connected data, to use for data driven business decision making.

Another emerging trend in Graph Database space is the Multi-Model Databases. NoSQL databases like OrientDB support storing Document and Graph data sets.

This eMag focuses on the graph database landscape and the real world use cases of graph databases. It includes articles and interviews covering topics like data modeling in graph databases and how companies use graph databases in their application. It also includes an article on full stack web development using a graph database so the readers can see the power of Graph databases to manage the connected data.

Let Me Graph That for You



Ian Robinson works on research and development for future versions of the Neo4j graph database. Harboring a long-held interest in connected data, he was for many years one of the foremost proponents of REST architectures before turning his focus from the Web's global graph to the realm of graph databases. Follow him on Twitter: @iansrobinson

Neo4j is designed to address challenges in the contemporary data landscape.

Why we would consider using a graph database to tackle complexity, generate insight, and bring value to end users. More specifically, to wrest insight from the kind of complexity that arises wherever three contemporary forces meet: an increase in the amount of generated and stored data, a need to accommodate a high degree of structural variation, and a need to understand the multiple facets of connectedness inherent in the domain to which the data belongs.

Increased data size — big data — is perhaps the best understood of these three forces. The volume of new data is growing exponentially each year, a trend that looks set to continue for the foreseeable future. But as the volume of data increases and we learn more about the instances in our domain, each instance begins to look subtly unique. In other words, as data volumes grow, we trade insight for

uniformity. The more data we gather about a group of entities, the more likely that data is to be variably structured.

Variably structured data is the kind of messy, real-world data that doesn't fit comfortably into a uniform, one-size-fits-all, rigid relational schema; the kind that gives rise to lots of sparse tables and null-checking logic. It's the increasing prevalence of variably structured data in today's applications that has led many organisations to adopt schema-free alternatives to the relational model and document stores.

But the challenges that face us today aren't just the management of increasingly large volumes of data nor do they simply extend to us having to accommodate ever-increasing degrees of structural variation in that data. The real challenge to generating significant insight is understanding connect-

edness: to answer the most important questions we want to ask of our domains, we must first know which things are connected and then, having identified these connected entities, understand in what ways and with what strength, weight, or quality they are connected.

Consider these questions:

- Which friends and colleagues do we have in common?
- Which applications and services in my network will be affected if a particular network element — a router or switch, for example — fails? Do we have redundancy throughout the network for our most important customers?
- What's the quickest route between two stations on the underground?
- What do you recommend this customer should buy, view, or listen to next?
- Which products, services, and subscriptions does a user have permission to access and modify?
- What's the cheapest or fastest means of delivering this parcel from A to B?
- Which parties are likely working together to defraud their bank or insurer?
- Which institutions are most at risk of poisoning the financial markets?

Have you had to answer questions such as these? If so, you've already encountered the need to manage and make sense of large volumes of variably structured, densely connected data. These are the kinds of problems for which graph databases are ideally suited. Understanding what depends on what, and how things flow; identifying and assessing risk, and analysing the impact of events on deep dependency chains: these are all connected data problems. Today, Neo4j is being used in business-critical applications in domains as diverse as social networking, recommendations, datacentre management, logistics, entitlements and authorization, route finding, telecommunications network monitoring, fraud analysis, and many others. Its widespread adoption challenges the notion that the relational databases are the best tools for working with connected data. At the same time, it proposes an alternative to the simplified, aggregate-oriented data models adopted by NoSQL.

The rise of NoSQL was largely driven by a need to remedy the perceived performance and operational limitations of relational technology. But in addressing performance and scalability, NoSQL has tended to surrender the expressive and flexible modelling capabilities of its relational predecessor, particularly with regard to connected data. Graph databases, in contrast, revitalise the world of connected data, shunning the simplifications of the NoSQL models

yet outperforming relational databases by several orders of magnitude.

To understand how graphs and graph databases help tackle complexity, we need first to understand Neo4j's graph data model.

The Labelled Property Graph Model

Neo4j uses a particular graph data model, called the labelled property graph model, to represent network structures. A labelled property graph consists of nodes, relationships, properties, and labels. Here's an example of a property graph:

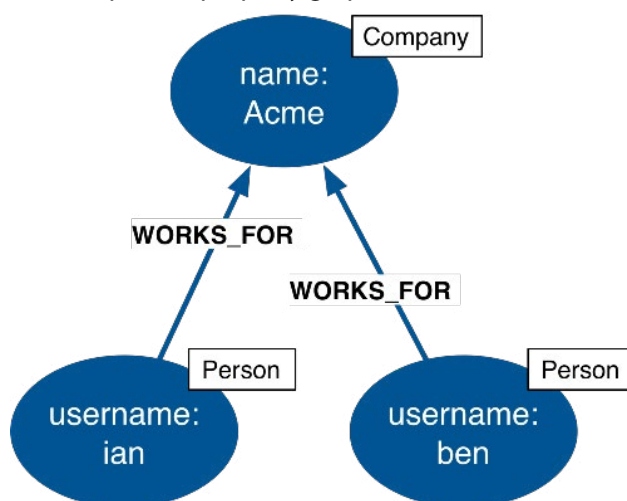


Diagram 1

Nodes

Nodes represent entity instances. To capture an entity's attributes, we attach key-value pairs (properties) to a node, thereby creating a record-like structure for each individual thing in our domain. Because Neo4j is a schema-free database, no two nodes need share the same set of properties: no two nodes representing persons, for example, need have the exact same attributes.

Relationships

Relationships represent the connections between entities. By connecting pairs of nodes with relationships, we introduce structure into the model. Every relationship must have a start node and an end node. Just as importantly, every relationship must have a name and a direction. A relationship's name and direction lend semantic clarity and context to the nodes attached to the relationship. This allows us — in, for example, a Twitter-like graph — to say that "Bill" (a node) "follows" (a named and directed relationship) "Sally" (another node). Like nodes, relationships can also contain properties. We typically use relationship properties to represent some distinguishing feature of each connection. This is particularly important when, in answering the questions we want to ask of our domain, we must not only trace

the connections between things, but also take account of the strength, weight, or quality of each of those connections.

Node labels

Nodes, relationships, and properties provide for tremendous flexibility. In effect, no two parts of the graph need have anything in common. Labels, in contrast, allow us to introduce an element of commonality with which to group nodes together and indicate the roles they play within our domain. We do this by attaching one or more labels to each of the nodes we want to group: we can, for example, label a node to make it represent both a user and, more specifically, an administrator. (Labels are optional, and therefore a node can have zero labels.) Node labels are similar to relationship names insofar as they lend additional semantic context to elements in the graph but while a relationship instance must perform exactly one role because it connects precisely two nodes, a node, by virtue of the fact it can be connected to any number of other nodes (or to none), may fulfil several different roles.

On top of this simple grouping capability, labels also allow us to associate indexes and constraints with nodes bearing specific labels. We can, for example, require that all nodes labelled “Book” are indexed by their ISBN property, and then further require that each ISBN property value is unique within the context of the graph.

Representing complexity

This graph model is probably the best abstraction we have for modelling both variable structure and connectedness. Variable structure grows by virtue of connections being specified at the instance level rather than the class level. Relationships join individual nodes, not classes of nodes: in consequence, no two nodes need connect in exactly the same way to their neighbours and no two subgraphs need be structured exactly alike. Each relationship in the graph represents a specific connection between two particular things. It’s this instance-level focus on things and the connections between things that make graphs ideal for representing and navigating a variably structured domain. Relationships not only specify that two things are connected, they also describe the nature and quality of that connection. To the extent that complexity is a function of the ways in which the semantic, structural, and qualitative aspects of the connections in a domain can vary, our data models require a means to express and exploit this connectedness. Neo4j’s labelled property graph model, wherein every relationship can not only be specified independently of every other but also may be annotated with properties that describe how,

in what degree, and with what weight, strength, or quality those entities connect, provides one of the most powerful means for managing complexity today.

And doing it fast

Join-intensive queries in a relational database are notoriously expensive, in large part because joins must be resolved at query time by way of an indirect index lookup. As an application’s dataset size grows, these join-inspired lookups slow down, causing performance to deteriorate. In Neo4j, in contrast, every relationship acts as a pre-computed join and every node acts as an index of its associated nodes. By having each element maintain direct references to its adjacent entities in this way, a graph database avoids the performance penalty imposed by index lookups — a feature sometimes known as index-free adjacency. As a result, Neo4j can be many thousands of times faster than a join-intensive operation in a relational database for complexly connected queries.

Index-free adjacency provides for queries whose performance characteristics are a function of the amount of the graph they choose to explore, rather than the overall size of the dataset. In other words, query performance tends to remain reasonably constant even as the dataset grows. Consider, for example, a social network in which every person has, on average, fifty friends. Given this invariant, friend-of-a-friend queries will remain reasonably constant, irrespective of whether the network has a thousand, a million, or a billion nodes.

Graph data modelling

This section looks at how to design and implement an application’s graph data model and associated queries.

From user story to domain questions

Imagine we’re building a cross-organizational skills finder: an application that allows us to find people with particular skills in a network of professional relationships.

To see how we might design a data model and associated queries for this application, we’ll follow the progress of one of our agile user stories, from analysis through to implementation in the database. The story follows.

As an employee, I want to know which of my colleagues have similar skills to mine so that I can exchange knowledge with them or ask them for help.

Given this end-user goal, our first task is to identify the questions we would have to ask of our domain in order to satisfy it. Here’s the story rephrased as a question. ►

Which people, who work for the same company as me, have similar skills to mine?

Whereas the user story describes what it is we're trying to achieve, the questions we pose to our domain hint as to how we might satisfy our users' goals. A good application graph data model makes it easy to ask and answer such questions. Fortunately, the questions themselves contain the germ of the structure we're looking for.

Language itself is a structuring of logical relationships. At its simplest, a sentence describes a person or thing, some action performed by this person or thing, and the target or recipient of that action, together with circumstantial detail such as when, where, or how this action takes place. By attending closely to the language we use to describe our domain and the questions we want to ask of our domain, we can readily identify a graph structure that represents this logical structuring in terms of nodes, relationships, properties, and labels.

From domain questions to Cypher path expressions

The particular question we outlined earlier names some of the significant entities in our domain: people, companies, and skills. Moreover, the question tells us something about how these entities are connected to one another:

- A person works for a company.
- A person has several skills.

These simple natural-language representations of our domain can now be transformed into Neo4j's query language, Cypher. Cypher is a declarative, SQL-like graph-pattern-matching language built around the concept of path expressions: declarative structures that allow us to describe to the database the kinds of graph patterns we wish either to find or to create inside our graph.

When translating our ordinary-language descriptions of the domain into Cypher path expressions, the nouns become candidate node labels, the verbs relationship names:

```
001 (:Person)-[:WORKS_FOR]->(:Company),  
    (:Person)-[:HAS_SKILL]->(:Skill)
```

Cypher uses parentheses to represent nodes, and dashes and less-than and greater-than signs (<- and ->) to represent relationships and their directions. Node labels and relationship names are prefixed with a colon; relationship names are placed inside square brackets in the middle of the relationship.

In creating our Cypher expressions, we've tweaked some of the language. The labels we've chosen refer to entities in the singular. More importantly, we've used HAS_SKILL rather than HAS to denote the relationship that connects a person to a

skill because HAS is too general a term. Rightsizing a graph's relationship names is key to developing a good application graph model. If the same relationship name is used with different semantics in several different contexts, queries that traverse those relationships will tend to explore far more of the graph than is strictly necessary — something we are mindful to avoid.

The expressions we've derived from the questions we want to ask of our domain form a prototypical path for our data model. In fact, we can refactor the expressions to form a single path expression:

```
001 (:Company)<-[:WORKS_FOR]-(:Person)-  
    [:HAS_SKILL]->(:Skill)
```

While there are likely many other requirements for our application and many other data elements to discover as a result of analysing those requirements, for the story at hand, this path structure captures all that is needed to meet our end users' immediate goals. There is still some work to do to design an application that can create instances of this path structure at run time as users add and amend their details, but insofar as this article is focussed on the design and implementation of the data model and associated queries, our next task is to implement the queries that target this structure.

A sample graph

To illustrate the query examples, we'll use Cypher's CREATE statement to build a small sample graph comprising two companies, their employees, and the skills and levels of proficiency possessed by each employee. (Code 1)

This statement uses Cypher path expressions to declare or describe the kind of graph structure we wish to introduce into the graph. In the first half, we create all the nodes we're interested in — in this instance, nodes representing companies, people and skills — and in the second half, we connect these nodes using appropriately named and directed relationships. The entire statement, however, executes as a single transaction.

Let's take a look at the first node definition.

```
001 (p1:Person{username:'ben'})
```

This expression describes a node labelled Person. The node has a username property whose value is "ben". The node definition is contained within parentheses. Inside the parentheses, we specify a colon-prefixed list of the labels attached to the node (there's just one here, Person), together with the node's properties. Cypher uses a JSON-like syntax to define the properties that belong to a node.

Having created the node, we then assign it to an identifier, p1. This identifier allows us to refer to the ▶


```

001 // Create skills-finder network
002
003 CREATE (p1:Person{username:'ben'}),
004 (p2:Person{username:'charlie'}),
005 (p3:Person{username:'lucy'}),
006 (p4:Person{username:'ian'}),
007 (p5:Person{username:'sarah'}),
008 (p6:Person{username:'emily'}),
009 (p7:Person{username:'gordon'}),
010 (p8:Person{username:'kate'}),
011 (c1:Company{name:'Acme'}),
012 (c2:Company{name:'Startup'}),
013 (s1:Skill{name:'Neo4j'}),
014 (s2:Skill{name:'REST'}),
015 (s3:Skill{name:'DotNet'}),
016 (s4:Skill{name:'Ruby'}),
017 (s5:Skill{name:'SQL'}),
018 (s6:Skill{name:'Architecture'}),
019 (s7:Skill{name:'Java'}),
020 (s8:Skill{name:'Python'}),
021 (s9:Skill{name:'Javascript'}),
022 (s10:Skill{name:'Clojure'}),
023 (p1)-[:WORKS_FOR]->(c1),
024 (p2)-[:WORKS_FOR]->(c1),
025 (p3)-[:WORKS_FOR]->(c1),
026 (p4)-[:WORKS_FOR]->(c1),
027 (p5)-[:WORKS_FOR]->(c2),
028 (p6)-[:WORKS_FOR]->(c2),
029 (p7)-[:WORKS_FOR]->(c2),
030 (p8)-[:WORKS_FOR]->(c2),
031 (p1)-[:HAS_SKILL{level:1}]->(s1),
032 (p1)-[:HAS_SKILL{level:3}]->(s2),
033 (p2)-[:HAS_SKILL{level:2}]->(s1),
034 (p2)-[:HAS_SKILL{level:1}]->(s9),
035 (p2)-[:HAS_SKILL{level:2}]->(s5),
036 (p3)-[:HAS_SKILL{level:3}]->(s3),
037 (p3)-[:HAS_SKILL{level:2}]->(s6),
038 (p3)-[:HAS_SKILL{level:1}]->(s8),
039 (p4)-[:HAS_SKILL{level:2}]->(s7),
040 (p4)-[:HAS_SKILL{level:3}]->(s1),
041 (p4)-[:HAS_SKILL{level:2}]->(s2),
042 (p5)-[:HAS_SKILL{level:1}]->(s1),
043 (p5)-[:HAS_SKILL{level:3}]->(s7),
044 (p5)-[:HAS_SKILL{level:2}]->(s2),
045 (p5)-[:HAS_SKILL{level:1}]->(s10),
046 (p6)-[:HAS_SKILL{level:2}]->(s1),
047 (p6)-[:HAS_SKILL{level:1}]->(s3),
048 (p6)-[:HAS_SKILL{level:2}]->(s8),
049 (p7)-[:HAS_SKILL{level:3}]->(s3),
050 (p7)-[:HAS_SKILL{level:1}]->(s4),
051 (p8)-[:HAS_SKILL{level:2}]->(s6),
052 (p8)-[:HAS_SKILL{level:3}]->(s8)

```

Code 1

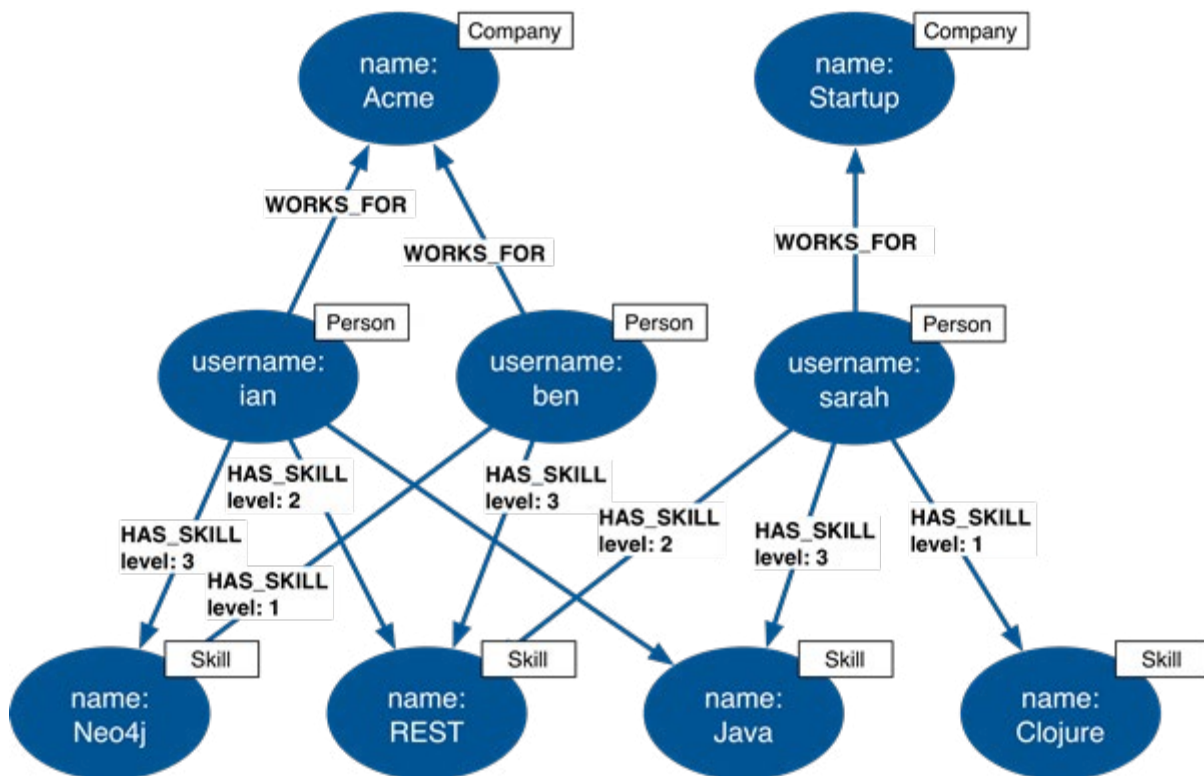


Diagram 2

newly created node elsewhere in the query. Identifiers are arbitrarily named, ephemeral, in-memory phenomena; they exist only within the scope of the query (or subquery) in which they are declared. They are not considered part of the graph and therefore are discarded when the data is persisted to disk.

Having created all the nodes representing people, companies, and skills, we then connect them as per our prototypical path expression: each person `WORKS_FOR` a company; each person `HAS_SKILL` one or more skills. Here's the first of the `HAS_SKILL` relationships.

```
001 (p1)-[:HAS_SKILL{level:1}]->(s1)
```

This relationship connects the node identified by `p1` to the node identified by `s1`. Besides specifying the relationship name, we've also attached a level property to this relationship using the same JSON-like syntax we used for node properties.

(We've used a single `CREATE` statement here to create an entire sample graph. This is not how we would populate a graph in a running application, where individual end-user activities trigger the creation or modification of data. For such applications, we'd use a mixture of `CREATE`, `SET`, `MERGE`, and `DELETE` to create and modify portions of the graph. You can read more about these operations in the online Cypher documentation.)

Diagram 2 shows a portion of the sample data. Within this structure you can clearly see multiple instances of our prototypical path.

Find colleagues with similar skills

Now that we've a sample dataset that exemplifies the path expressions we derived from our user story, we can return to the question we want to ask of our domain and express it more formally as a Cypher query. Here's the question again.

Which people, who work for the same company as me, have similar skills to me?

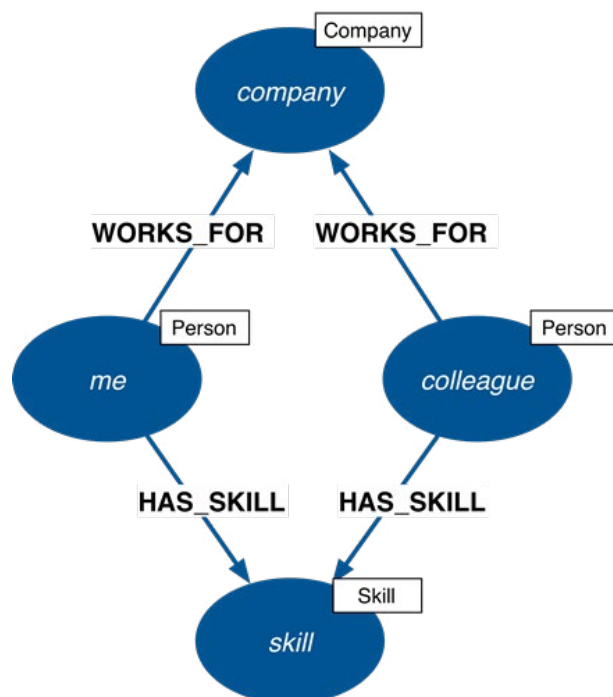


Diagram 3

To answer this question, we're going to have to find a particular graph pattern in our sample data. Let's assume that somewhere in the existing data is a node labelled Person that represents me (I have the username "ian"). That node will be connected to a node labelled Company by way of an outgoing WORKS_FOR relationship. It will also be connected to one or more nodes labelled Skill by way of several outgoing HAS_SKILL relationships. To find colleagues who share my skill set, we're going to have to find all the other nodes labelled Person that are connected to the same company node as me and that are also connected to at least one of the skill nodes to which I'm connected. In diagrammatic form, the following is the pattern we're looking for. (Diagram 3)

Our query will look for multiple instances of this pattern inside the existing graph data. For each colleague who shares one skill with me, we'll match the pattern once. If a person has two skills in common with me, we'll match the pattern twice, and so on. Each match will be anchored on the node that represents me. Using Cypher path expressions, we can describe this pattern to Neo4j. Here's the full query.

```
001 // Find colleagues with similar
    skills
002 MATCH (me:Person{username:'ian'})
003       -[:WORKS_FOR]->(company:Company),
004       (me)-[:HAS_SKILL]->(skill:Skill),
005       (colleague:Person)-[:WORKS_FOR]-
006         >(company),
007       (colleague)-[:HAS_SKILL]-
008         >(skill)
009 RETURN colleague.username AS
    username,
010       count(skill) AS score,
011       collect(skill.name) AS skills
012 ORDER BY score DESC
```

This query comprises two clauses: a MATCH clause and a RETURN clause. The MATCH clause describes the graph pattern we want to find in the existing data; the RETURN clause generates a projection of the results on behalf of the client.

The first line of the MATCH clause, (me:Person{username:'ian'}), locates the node in the existing data that represents me — a node labelled Person with a username property whose value is "ian" — and assigns it to the identifier me. If there are multiple nodes matching these criteria (unlikely because username ought to be unique), me will be bound to a list of nodes.

The rest of the MATCH clause then describes the diamond-shaped pattern we want to find in the graph. In describing this pattern, we specify the labels that must be attached to a node for it to match (Company for companies, Skill for skills, Person for

colleagues), and the names and the directions of the relationships that must be present between nodes for them to match (a Person must be connected to a Company with an outgoing WORKS_FOR relationship, and to a Skill with an outgoing HAS_SKILL relationship). Where we want to refer to a matched node later in the query, we assign it to an identifier (we've chosen colleague, company, and skill). By being as explicit as we can about the pattern, we help ensure that Cypher explores no more of the graph than is strictly necessary to answer the query.

The RETURN clause generates a tabular projection of the results. As I mentioned earlier, we're matching multiple instances of the pattern. Colleagues with more than one skill in common with me will match multiple times. In the results, however, we only want to see one line per colleague. Using the count and collect functions, we aggregate the results on a per colleague basis. The count function counts the number of skills we've matched per colleague, and aliases this as their score. The collect function creates a comma-separated list of the skills that each colleague has in common with me, and aliases this as skills. Finally, we order the results, highest score first.

Executing this query against the sample dataset generates the following results:

username	score	skills
ben	2	['Neo4j', 'REST']
charlie	1	['Neo4j']

The important point about this query and the process that led to its formulation is that the paths we use to search the data are very similar to the paths we use to create the data in the first place. The diamond-shaped pattern at the heart of our query has two legs, each comprising a path that joins a person to a company and a skill:

```
001 (:Company)<-[:WORKS_FOR]-(:Person)
    [:HAS_SKILL]->(:Skill)
```

This is the very same path structure we came up with for our data model. The similarity shouldn't surprise us; after all, both the underlying model and the query we execute against that model are derived from the question we wanted to ask of our domain.

Filter by skill level

In our sample graph, we qualified each HAS_SKILL relationship with a level property that indicates an individual's proficiency with regard to the skill to which the relationship points: 1 for beginner, 2 for intermediate, 3 for expert. We can use this property in our query to restrict matches to only those people who are, for example, level 2 or above in the skills they share with us: ►

```

001 // Find colleagues with shared
    skills,
002 level 2 or above
003 MATCH (me:Person{username:'ian'})
004       -[:WORKS_FOR]->(company),
005       (me)-[:HAS_SKILL]->(skill),
006       (colleague)-[:WORKS_FOR]-
    >(company),
007       (colleague)-[r:HAS_SKILL]-
    >(skill)
008 WHERE r.level >= 2
009 RETURN colleague.username AS
    username,
010       count(skill) AS score,
011       collect(skill.name) AS skills
012 ORDER BY score DESC

```

I've highlighted the changes to the original query. In the MATCH clause, we now assign a colleague's HAS_SKILL relationships to an identifier *r* (meaning that *r* will be bound to a list of such relationships). We then introduce a WHERE clause that limits the match to cases where the value of the level property on the relationships bound to *r* is 2 or greater.

Running this query against the sample data returns the following results:

Username	score	skills
Charlie	1	['Neo4j']
Ben	1	['Neo4j']

Search across companies

As a final illustration of the flexibility of our simple data model, we'll tweak the query again so that we no longer limit it to the company where I work, but instead search across all companies for people with skills in common with me:

```

001 // Find people with shared skills,
    level 2
002 or above
003 MATCH (me:Person{username:'ian'})
004       -[:HAS_SKILL]->(skill),
005       (other)-[:WORKS_FOR]->(company),
006       (other)-[r:HAS_SKILL]->(skill)
007 WHERE r.level >= 2
008 RETURN other.username AS username,
009
010       company.name AS company,
011       count(skill) AS score,
012       collect(skill.name) AS skills
013 ORDER BY score DESC

```

To facilitate this search, we've removed the requirement that the other person must be connected to the same company node as me. We do, however, still identify the company for whom this other person works. This then allows us to add the company name to the results. The pattern described by the MATCH clause now looks as follows. (Diagram 4)

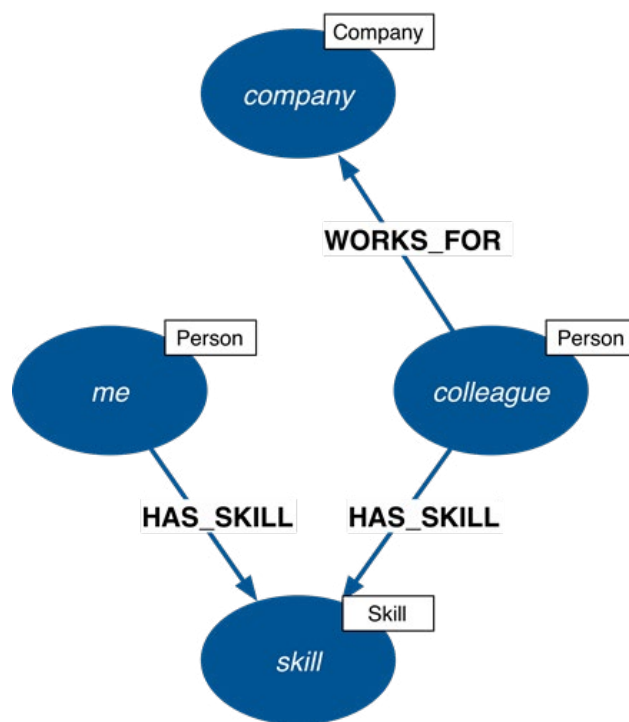


Diagram 4

Running this query against the sample data returns the following results:

username	company	score	skills
sarah	Startup, Ltd	2	['Java', 'REST']
ben	Acme, Inc	1	['REST']
emily	Startup, Ltd	1	['Neo4j']
charlie	Acme, Inc	1	['Neo4j']

Modelling

We've looked at how we derive an application's graph data model and associated queries from end-user requirements. In summary, we:

- describe the client or end-user goals that motivate our model;
- rewrite those goals as questions we would have to ask of our domain;
- identify the entities and the relationships between them that appear in these questions;
- translate these entities and relationships into Cypher path expressions; and
- express the questions we want to ask of our domain as graph patterns using path expressions similar to the ones we used to model the domain.

Use Cypher to describe your model

Use Cypher path expressions rather than an intermediate modelling language such as UML to describe your domain and its model. As we've seen, many of the noun and verb phrases in the questions we want

to ask of our domain can be straightforwardly transformed into Cypher path expressions, which then become the basis of both the model itself and the queries we want to execute against that model. In such circumstances, the use of an intermediate modelling language adds very little. This is not to say that Cypher path expressions comprehensively address all of our modelling needs. Besides capturing the structure of the graph, we also need to describe how both the graph structure and the values of individual node and relationship properties ought to be constrained. Cypher does provide for some constraints today, and the number of constraints it supports will rise with each release, but there are occasions where domain invariants must be expressed as annotations to the expressions we use to capture the core of the model.

Name relationships based on use cases

Derive your relationship names from your use cases. Doing so creates paths in your model that easily align with the patterns you want to find in your data. This ensures that queries that take advantage of these paths will ignore all other nodes and relationships.

Relationships both compose and partition the graph. In connecting nodes, they structure the whole, creating a complex composite from what would otherwise be simple islands of data. Because they can be differentiated based on their names, directions, and property values, relationships also serve to partition the graph, allowing us to identity specific subgraphs within a larger, more generally connected structure. By focussing our queries on certain relationship names and directions and the paths they form, we exclude other relationships and other paths, effectively materializing a particular view of the graph dedicated to addressing a particular need.

You might think this smacks somewhat of an overly specialized approach, and in many ways it is one. But it's rarely an issue. Graphs don't exhibit the same degree of specialization tax as relational models. The relational world has an uneasy relationship with specialization, both abhorring it yet requiring it, and then suffering when it's used.

Consider that we apply the normal forms in order to derive a logical structure capable of supporting ad hoc queries —

that is, queries we haven't yet thought of. All is well and good — until we go into production. At that point, for the sake of performance, we denormalize the data, effectively specializing it on behalf of an application's specific access patterns. This denormalization helps in the near term but poses a risk for the future, for in specializing for one access pattern, we effectively close the door on many others. Relational modellers frequently face these kinds of either/or dilemmas: either stick with the normal forms and degrade performance or denormalize and limit the scope for evolving the application further down the line.

Not so with graph modelling. Because the graph allows us to introduce new relationships at the level of individual node instances, we can specialize it over and over again, use case by use case, in an additive fashion — that is, by adding new routes to an existing structure. We don't need to destroy the old to accommodate the new; rather, we simply introduce the new configuration by connecting old nodes with new relationships. These new relationships effectively materialize previously unthought-of graph structures to new queries. Their introduction into the graph, however, need not upset the view enjoyed by existing queries.

Pay attention to language

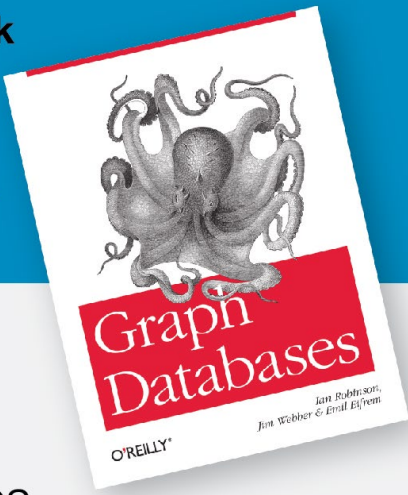
In our modelling example, we derived a couple of path expressions from the noun and verb phrases we used to describe common relationships. There are a few rules of thumb to analysing a natural-language representation of a domain. Common nouns become candidates for labels: "person", "company" and "skill" become Person, Company, and Skill. Verbs that ►

FREE O'Reilly Book

Graph Databases

The definitive resource on graph databases

Download Now



take an object — “owns”, “wrote”, and “bought”, for example — are candidates for relationship names. Proper nouns — a person or company name, for example — refer to an instance of a thing, which we then typically model as a node.

Things aren’t always so straightforward. Subject-verb-object constructs easily transform into graph structures, but a lot of the sentences we use to describe our domain are not so simple. Adverbial phrases, for example — those additional parts of a sentence that describe how, when, or where an action takes place — result in what entity-relational modelling calls “n-ary” relationships: complex, multidimensional relationships that bind several things and concepts.

The representation of n-ary relationships would appear to require something more sophisticated than a property graph, like a model that allows relationships to connect more than two nodes or that permits one relationship to connect to and thereby qualify another. Such data-model constructs are, however, almost always unnecessary. To express a complex interrelation of several things, we need only introduce an intermediate node: a hub-like node that connects all the parties to an n-ary relationship.

Intermediate nodes are a common occurrence in many application graph data models. Does their widespread use imply that there is a deficiency in the property graph model? I think not. More often than not, an intermediate node makes visible one more element of the domain — a hidden or implicit concept with informational content and a meaningful domain semantic all its own.

Intermediate nodes are usually self-evident wherever an adverbial phrase qualifies a clause. “Bill worked at Acme from 2005-2007 as a software engineer” leads us to introduce an intermediate node that connects Bill, Acme, and the role of software engineer. It quickly becomes apparent that this node represents a job or an instance of employment, to which we can attach the date properties “from” and “to”.

It’s not always so straightforward. Some intermediate nodes lie hidden in far more obscure locales. Verbing — the language habit whereby a noun is used as a verb — can often occlude the presence of an intermediate node. Technical and business jargon is particularly rife with such neologisms: we “email” one another, rather than send an email and “google” for results, rather than search Google.

The verb “email” provides a ready example of the kinds of difficulties we can encounter if we ignore the noun origins of some verbs. The following path shows the result of us treating “email” as a relationship name.

```
001 (:Person{name:'Alice'}) -[:EMAILED]->(:Person{name:'Lucy'})
```

This looks straightforward enough. In fact, it’s a little too straightforward, for with this construct it becomes extremely difficult to indicate that Alice also carbon-copied Alex. But if we unpack the noun origins of “email”, we discover both an important domain concept — the electronic communication itself — and an intermediate node that connects senders and receivers:

```
001 (:Person{name:'Alice'})
002 -[:SENT]->(e:Email{subject:'Annual
003 report'})
004 -[:TO]->(:Person{name:'Lucy'}),
005 (e)-[:CC]->(:Person{name:'Alex'})
```

If you’re struggling to come up with a graph structure that captures the complex interdependencies among several things in your domain, look for the nouns, and hence the domain concepts, hidden on the far side of some of the verb phrases you use to describe the structure of your domain.

Conclusion

Once a niche academic topic, graphs are now a commodity technology. Neo4j makes it easy to model, store, and query large amounts of variably structured, densely connected data, and we can design and implement an application graph data model by transforming user stories into graph structures and declarative graph-pattern-matching queries. If you’re beginning to think in graphs, head over to <http://neo4j.com> and grab a copy of Neo4j. ■

Data Modelling in Graph Databases: Interview with Jim Webber and Ian Robinson



by Srimi Penchikala



Jim Webber is chief scientist at Neo Technology, a distributed-systems specialist working on very-large-scale graph data technology.



Ian Robinson works on research and development for future versions of the Neo4j graph database. Harboring a long-held interest in connected data, he was for many years one of the foremost proponents of REST architectures before turning his focus from the Web's global graph to the realm of graph databases. Follow him on Twitter: @iansrobinson

Graph databases are NoSQL database systems that use graph data model for storage and processing of data.

Matt Aslett from the 451 Group notes that graphs are now emerging from the general NoSQL umbrella as a category in their own right. In 2014-2015, there has been growth in the category of all things graph.

Data modeling with a graph database is different from how

you model the data stored in relational or other NoSQL databases like document databases, key-value data stores, or column-family databases. You can use graph data models to create rich and highly connected data to represent real-world use cases and applications.

InfoQ spoke with Ian Robinson and Jim Webber of Neo Technologies, who co-authored O'Reilly's [Graph Databases](#), about data modelling and best practices when using graph databases for data management and analytics. ►

InfoQ: What type of data is not suitable for storing in a relational database but is a good candidate for a graph database?

Ian Robinson and Jim Webber: That's pretty straightforward to answer: anything that's interconnected either immediately, because the coding and schema design is complicated, or eventually, because of the [join-bomb problem](#) inherent in any practical application of the relational model.

Relational databases are fine things, even for large data sets, up to the point where you have to join. And in every relational-database use case that we've seen, there's always a join — and in extreme cases, when an ORM has written and hidden particularly poor SQL, many indiscriminate joins.

The problem with a join is that you never know what intermediate set will be produced, meaning you never quite know the memory use or latency of a query with a join. Multiplying that out with several joins means you have enormous potential for queries to run slowly while consuming lots of (scarce) resources.

What are the advantages of a graph database over a relational database?

As an industry, we've become rather creative at forcing all kinds of data into a relational database (and we've become philosophical about the consequences!). Relational databases are truly the golden hammer of computer-systems development, to the point where many of us are reluctant to drop RDBMS from our tool chain in favour of something

better, simply because of familiarity.

However, there are often two drivers underlying a move from a relational database to Neo4j. The first is the observation that your domain is a connected data structure (e.g. social network, healthcare, rights management, real-time logistics, recommendations...) and then understanding that such domains are easy and pleasant to store and query in Neo4j, but difficult and unpleasant in a relational database. Typically these cases are driven by technologists who understand, at least to some degree, that they are dealing with a graph problem and are prepared to use Neo4j to solve that graph problem elegantly and quickly — they don't want to be stuck in a miasma of sparse tables and the über-join table.

The second driver is performance. Join pain in relational databases is debilitating for systems that use them. Perhaps your first join performs well; if you're lucky, maybe even your second does, too. But as the size of a dataset grows, confidence in those joins diminishes as query times get longer and longer. Join-intensive models usually come about because they're trying to solve some kind of connection or path problem, but the maths underlying relational databases simply aren't well suited to emulating path operations. Neo4j has no such penalties for path operations: query times scale linearly with the amount of data you choose to explore as part of your query, not with the overall size of the dataset (which can be enormous). Having join pain is another indicator that a Neo4j graph will be a superior solution to a complex data model in a relational database.

Why use a graph database over other NoSQL databases?

Fowler and Sadalage answer this, we think, very clearly in their book *NoSQL Distilled*. They point out that of the four types of NoSQL store — graph, key-value, column, and document — the latter three are what they term "aggregate stores". Aggregate stores work well when the pattern of storage and retrieval is symmetric. Store a shopping basket by key, retrieve it by key; store a customer document and retrieve it, and so on.

But when you want to analyse the data across aggregates, things get trickier. How do you find the popular products for different customer demographics? How do you do this in real time, as you're serving a customer on your system right now?

These activities, though basic in domain terms, are tricky to solve with aggregate stores. As a result, developers using these stores are forced to compute rather than query to get an answer. This is why aggregate stores are so keen on map-reduce-style interactions.

Neo4j's data model is far more expressive than aggregate stores or relational databases. Importantly, the model stresses connectivity as a first-class concept. It is connectivity in the graph between customers, products, demographics, and trends that yields the answers to these kinds of real-time analytics problems. Neo4j provides answers by traversing (querying) the graph rather than resorting to latent map-reduce computations.

In a graph, you bring together arbitrary dimensions (different relationship types) at query time to answer sophisticated questions with ease and excellent performance. In non-native graph databases (which includes

the other kinds of NoSQL stores), traversals are faked: they happen at the application level in code you have to write and maintain. They're also over the network and are orders of magnitude slower than the graph-native, highly optimised graph query engine that underpins Neo4j.

How do you typically model data with a graph database?

Neo4j uses a graph model called the "labelled property graph". This is a pragmatic model that eschews some of the more esoteric mathematical bits of graph theory in favour of ease of understanding and design.

The labelled property graph consists of nodes (which we typically use to represent entities) connected by relationships. Nodes can be tagged with one or more labels to indicate the role each node plays within our dataset. Every relationship has a name and a direction, which together provide semantic context for the nodes connected by that relationship. Both nodes and relationships can contain one or more properties. We typically use node properties to represent the attributes of an entity, and relationship properties to specify the weight, strength, or quality of that particular relationship.

These graph primitives provide us with a simple, compact, and easy-to-reason-about modelling kit. For example, using Neo4j's Cypher query language, we can easily describe that Alice loves cookies:

```
(:Person {name: 'Alice'}) -[:LOVES]->(:Food {type: 'Cookie'})
```

This path expression says that there's a node representing a Person named Alice who loves a particular food type, cookie. It's

easy to imagine how repeating this many times gives a large and interesting graph of people and their food preferences (or allergies, etc.).

Data modelling consists of using the property-graph primitives — nodes, relationships, properties, and labels — to build an application-specific graph data model that allows us to easily express the questions we want to ask of that application's domain.

When building an application with Neo4j, we typically employ a multistep process, which starts with a description of the problem we're trying to solve and ends with the queries we want to execute against an application-specific graph data model. This process can be applied in an iterative and incremental manner to build a data model that evolves in step with the iterative and incremental development of the rest of the application.

Step 1: Describe the client or end-user goals that motivate our model.

What's the problem we're trying to solve? We've found that agile user stories are great for providing concise, natural-language descriptions of the problems we intend our model to address, but pretty much any description of a requirement can serve as the basis of our modelling efforts.

Step 2: Rewrite those goals as questions we would have to ask of our domain.

An agile user story describes what it is we're trying to achieve with our application. By rewriting each application goal in terms of the questions the domain would have to answer to achieve that goal, we take a step towards identifying how we might go about implementing that application. We're still dealing with an informal description

of the solution at this stage, but the domain-specific questions we describe now provide rich input for the next step of the process.

Step 3: Identify the entities and the relationships between them that appear in these questions.

Language itself is a structuring of logical relationships. By attending closely to the language we use to describe our domain and the questions we want to ask of it, we can readily identify a graph structure that represents this logical structuring in terms of nodes, relationships, properties, and labels. Common nouns — "person" or "company", for example — tend to refer to groups or classes of things, or the roles that such things perform: these common nouns become candidate label names. Verbs that take an object indicate how things are connected: these then become candidate relationship names. Proper nouns — a person's or company's name, for example — tend to refer to an instance of a thing, which we model as a node and its properties.

Step 4: Translate these entities and relationships into Cypher path expressions.

In this step, we formalize the description of our candidate nodes, relationships, properties, and labels using Cypher path expressions. These path expressions form the basis of our application graph data model in that they describe the paths, and hence the structure, that we would expect to find in a graph dedicated to addressing our application's needs.

Step 5: Express the questions we want to ask of our domain as graph patterns using path expressions similar to the ones we used to model the domain. ►

Having derived the basic graph structure from the questions we would want to ask of our domain, we're now in a position to express the questions themselves as graph queries that target this same structure. At the heart of most Cypher queries is a MATCH clause containing one or more path expressions that describe the kind of graph structure we want either to find or to create inside our dataset. If we've been diligent in allowing our natural-language descriptions of the domain to guide the basic graph structure, we will find that many of the queries we execute against our data will use similar path expressions to the ones we used to structure the graph. The key point here is that the resulting structure is an expression of the questions we want to ask of the domain: the model is isomorphic to the queries we want to execute against the model.

Should the modelling happen in the database or application layer?

As will be evident from our description of the modelling process, much of the modelling takes place in the database. The labelled-property-graph primitives allow us to create extremely expressive, semantically rich graph models, with little or no accidental complexity — there are no foreign keys or join tables, for example, to obscure our modelling intent.

That said, there are several characteristics of the domain that are best implemented in the application. Neo4j doesn't store behaviour. Nor does it have any strict concept of what domain-driven design calls an "aggregate": a composite structure bounded by a root entity that

controls access to and manages the lifecycle of the whole.

That the graph has no notion of an aggregate boundary (beyond the boundary imposed by a node's record-like structure) is not necessarily a drawback. The graph model, after all, emphasizes interconnectedness. Some of the most valuable insights we can generate from our data require us to take account of connections between things that in any other context would be considered discrete entities. Many techniques of predictive analysis and forensic analysis depend on our being able to infer or identify new composites, new boundary-breaking connected structures that don't appear in our initial conception of the domain. Cypher's rich path-expression syntax, with its support for variable-length paths and optional subgraph structures, effectively allows us to identify and materialize these new composite structures at query time.

What are hyperedges and how should they be modelled?

Hyperedges come from a different graph model known as a hypergraph. A hyperedge is a special kind of relationship that connects more than two nodes. You can imagine a somewhat contrived example where you like something on Facebook and your friend likes that like. While they're beloved of theoreticians and some other graph databases, hyperedges are not a first-class citizen in Neo4j. Our experience is that they're only useful in a relatively small number of use cases. Their narrow utility is offset by their additional complexity and intellectual cost.

You can, of course, model hyperedges in Neo4j just by

adding another node (an intermediate node) to the graph. For example, we could cast the original like, which was (alice)-[:LIKES]-(post), as (alice)-[:CREATED]->(-like)-[:FOR]->(post) — and now that we have the (like) node, it's easy to like it as (bob)-[:LIKES]->(-like), giving hyperedge equivalent functionality when you need it and avoiding those complexities when you don't (which is most of the time).

What are the best practices for modelling the graph data?

Derive your relationship names from your use cases. Doing so creates paths in your model that align easily with the patterns you want to find in your data. This ensures that the queries you derive from your use cases only see these paths, thereby eliminating irrelevant parts of the surrounding graph from consideration. As new use cases emerge, you can reuse existing relationships or introduce new ones, as needs dictate.

Use intermediate nodes to connect multiple dimensions. Intermediate nodes represent domain-meaningful hubs that connect an arbitrary number of entities. A job node, for example, can connect a person, a role, and a company to represent a time-bounded instance of employment. As we learn more about the context of that job (where the individual worked, for example), we can enrich our initial structure with additional nodes.

Connect nodes in linked lists to represent logical or temporal ordering. Using different relationships, we can even interleave linked lists. The episodes

of a TV programme, for example, can be connected in one linked list to represent the order in which they were broadcast (using, say, NEXT_BROADCAST relationships), while at the same time being connected in another linked list to represent the order in which they were made (using NEXT_IN_PRODUCTION relationships). Same nodes, same entities, but two different structures.

Can you discuss the design considerations for graph traversal?

Unlike RDBMS, query latency in a graph is proportional to how much of the graph you choose to search. That means you as a query designer should try to minimize the amount of graph you search. You should search just enough to answer your query.

In Neo4j, that means adding as many constraints into your query as you can. Constraints will prevent the database from searching paths that you already know will be fruitless for your result. For example, if you know that you're only interested in other people in a large social graph who are single and share compatible sexual orientation/hobbies/interests, you should constrain your search accordingly. That way, you'll avoid visiting parts of that social network with incompatible sexual orientation/boring hobbies/bizarre interests, and you'll get an answer much more quickly.

In Cypher, you can express a poor match very loosely, like this:

```
(me)-[*]->(other)
```

This query matches all relationship types, at any depth (that's the asterisk), to any kind of thing. As a result, it would likely visit the whole graph many times

over, thereby impacting performance. In contrast, because we know some domain invariants, we can instead cast this query as:

```
(me)-[:FRIEND]->()-[:FRIEND]->(other)
```

Here we've constrained both the relationship type and the depth to find only friends of friends (because dating friends is yucky). This is better, but we can go even further, adding additional constraints such as gender, sexual preference, and interests:

```
(me)-[:FRIEND]->()-[:FRIEND]->(other:Heterosexual:Female)-[:LIKES]->(:TV_SHOW {title: 'Doctor Who'})
```

Now we'll only match against heterosexual females (the two colon-separated labels on the other node) who also like the TV show titled *Doctor Who*. Neo4j can now aggressively prune any parts of the graph that don't match, significantly reducing the amount of work that needs to be done, and thereby keeping latency very low (typically small milliseconds even for large data sets).

Are there any anti-patterns when working with graph data?

There certainly are anti-patterns that can catch out the unwary or people who are new to graph modelling. For example, in Graph Databases, we discuss a case of e-mail forensics (in Chapter 3). In that example, we're looking for potentially illegal activities in an organization by individuals swapping information for purposes like insider trading (think: Enron).

When we design graph models, we sanity-check the graph by reading it. For exam-

ple, if we had the structure (Alice)-[:EMAILED]->(Bob) then we might think we've built a sound model since it reads well left to right (Alice e-mailed Bob) and makes sense the reverse way (Bob was e-mailed by Alice).

Initially, we went with this model but it soon became apparent that it was lossy. When it came time to query an e-mail that could violate communications policy, we found the e-mail didn't actually exist — quite a problem! Furthermore, where we expected to see several e-mails confirming the corrupt activity, all we saw was that Alice and Bob had e-mailed each other several times. Because of our imprecise use of English, we'd accidentally encoded a core domain entity — the e-mail itself — into a relationship when it should have been a node.

Fortunately, once we'd understood that we'd created a lossy model, it was straightforward to correct it using an intermediate node: (Alice)-[:SENT]->(email)-[:TO]->(Bob). Now we have that intermediate node representing the e-mail, we know who sent it (Alice) and to whom it was addressed (Bob). It's easy to extend the model so that we can capture who was CC'd or BCC'd like so: (Charlie)-[:CC]-(email)-[:BCC]->(Daisy). From here, it's easy to see how we'd construct a large-scale graph of all e-mail exchanged and map out patterns that would catch anyone violating the rules, but we'd have missed them if we hadn't thought carefully about nodes and relationships.

If there was just one piece of advice for people coming fresh to graph modelling, it would be don't (accidentally) encode entities as relationships. ►

Can you talk about any gotchas or limitations of graph databases?

The biggest gotcha we both found when moving from RDBMS to graphs some years back (when we were users of early Neo4j versions, way before we were developers on the product) was the years of entrenched relational modelling. We found it difficult to leave behind the design process of creating a normalized model and then denormalizing it for performance — we didn't feel like we'd worked hard enough with the graph to be successful.

It was agonizing in a way. You'd finish up some piece of work quickly and the model would work efficiently with high fidelity, then you'd spend ages worrying that you'd missed something.

Today, there are materials to help get people off the ground far more quickly. There are good books, blog posts, Google groups, and a healthy Meetup community all focussed on graphs and Neo4j. That sense of not working hard enough is quickly dispelled once you've gotten involved with graphs and you get on the real work of modelling and querying your domain — the data model gets out of the way, which is as it should be.

What is the current status of standards in graph data management with respect to data queries, analytics, etc.?

As with most NoSQL stores, it's a little early to think about standards (though the folks at Oracle would seemingly disagree). At last year's NoSQL Now! conference, the general opinion was that operational interfaces should be standardized so that users can plug their databases into a stan-

dard monitoring and alerting infrastructure. Indeed, that's now happening.

In terms of standards specific to the graph space, it's still premature. For example, Neo4j recently introduced labels for nodes, something that didn't exist in the previous 10-year history of the database. Now that graph databases are taking off, this kind of rapid innovation in the model and in the implementation of graph databases means it's too early to try to standardize: our industry is still too immature and the lack of standards is definitely not inhibiting take-up of graph databases, which are now the fastest growing segment of the whole data market (RDBMS and NoSQL).

In the meantime, since Neo4j makes up in excess of 90% of the entire graph database market, it acts as a de facto standard itself, with a plethora of third-party connectors and frameworks allowing you to plug it into your application or monitoring stack.

What is the future of graph databases in general and Neo4j in particular?

Graphs are a very-general-purpose data model. Much as we have seen RDBMS become a technology that has been widely deployed in many domains in the past, we expect graph databases to be even more widely deployed across many domains in the future. That is, we expect graph databases to be the default choice and the first model that you think of when you hear the word "database."

To do that, there are a few things that we need to solve. Firstly, graph databases need to be even easier to use — the out-of-box experience has to be painless or even pleasurable. Although the

model is always going to take a little more learning than a document store (it is, after all, a richer model), there are certain mechanical things we can do just to make that learning curve easier.

You've seen this with the recent release of Neo4j 2.0, where we introduced labels to the graph model, provided declarative indexing based on those labels, and built a fabulous new UI with excellent visualization and a new version of Cypher with a productive REPL. But our efforts don't end there: soon we'll be releasing a point version of Neo4j that takes all the pain out of bulk imports to the database (something all users do at least once) and then we'll keep refining our ease of use.

The other research and development thread that we're following is performance and scale. For example, we've got some great ideas to vertically scale ACID transactional writes in Neo4j by orders of magnitude using a write window to batch IO. From the user's perspective, Neo4j remains ACID compliant, but under the covers, we amortize the cost of IO across many writes. This is only possible in native graph databases like Neo4j because we own the code all the way down to the disk and can optimize the whole stack accordingly. Non-native graph databases simply don't have this option.

For performance in the large, we're excited by the work by Bailis et al (2013) on highly available transactions (HATs), which provide non-blocking transactional agreement across a cluster, and by RAMP transactions which maintain ACID constraints in a database cluster while allowing non-contending transactions to execute in parallel. These kind of ideas for the mechanical bedrock for our work on highly distributed graph databases, on which you'll hear more from us in the coming months. ■

High Tech, High Security: Security Concerns in Graph Databases



George Hurlburt is chief scientist at STEMCorp, a nonprofit that works to further economic development via the adoption of network science and to advance autonomous technologies as useful tools for human use. Contact him at ghurlburt@change-index.com.

Cybersecurity measures are best accommodated in system design because retrofits can be costly. New technologies and applications, however, bring new security and privacy challenges, and the consequences of new technology are often difficult to anticipate. Such is the case with graph databases, a relatively new database technology that's gaining popularity.

The emergence of NoSQL

The relational-database management system (RDBMS), initially designed to maximize highly expensive storage, has indeed proven to be highly effective in transaction-rich and process-stable environments. For example, the RDBMS excels in large-scale credit-card transaction processing and cyclic billing operations. It offers superior performance

in the realm of indexed spatial data, but it fares poorly in highly dynamic environments, such as in a management information system that depends on volatile data or a systems architecture with a high churn of many-to-many relationships. In such environments, RDBMS design imposes far too much mathematical and managerial overhead.

The rise of the NoSQL database represents an alternative

to the decades-long reign of the RDBMS. Various forms of NoSQL database opened doors to a vastly improved dynamic data portrayal with far less overhead. For example, schemas need not be as rigorous in the NoSQL world. NoSQL database designs include wide-column stores, document stores, key-value (tuple) stores, multimodal databases, object databases, grid/cloud databases, and graph databases. The graph

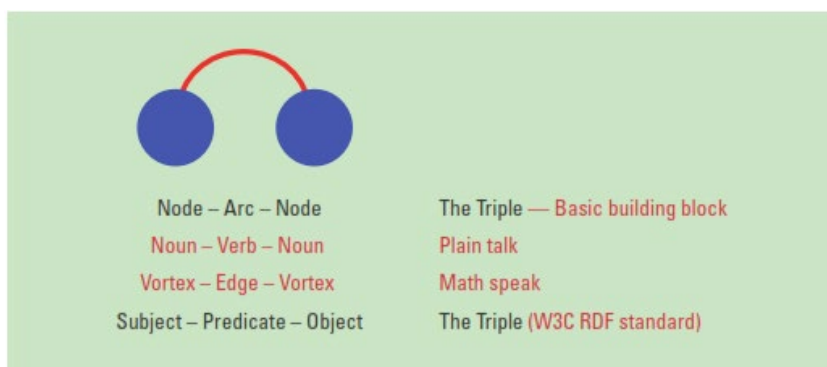


Figure 1. This simple node-arc-node triad, often called a triple, is the fundamental building block for describing all manner of complex networks in great detail.

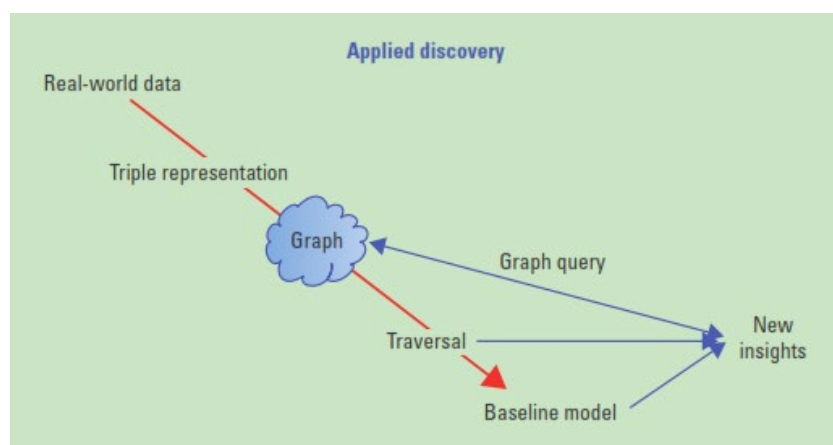


Figure 2. A graph database harnessing discovery. Such discovery could support a detailed build-out of the complex relationships between ocean and atmosphere that compose climatic conditions, or could hasten the discovery of how Ebola might spread in Western Africa.

data base, crossing many lines in the NoSQL world, stands poised to become a successful technology.

The graph database

The graph database relies on the familiar node-arc-node or, perhaps more simplistically, noun-verb-noun relationship of a network (see Figure 1). A node can be any object. An arc represents the relationship between nodes. Both nodes and arcs can contain properties. This simple node-arc-node triad, often called a triple, is the fundamental building block for describing all manner of complex networks in great detail.

Networks such as an electrical grid, a corporate supply chain, or an entire ecosystem are often composed of numerous nodes that share huge numbers of multi-

ple relationships across arcs. Networks of all kinds lend themselves well to graph representation. The graph database harnesses this powerful capability to represent network composition and connectivity. Graph databases have matured to support discovery, knowledge management, and even prediction.

In an Internet-connected world, where networks of all types become increasingly preeminent, such a network capability is becoming essential to modern sense making. However, like the RDBMS, the graph database is just another tool in the box, and it can be harnessed for good or ill. Thus, it's not premature to consider the large-scale security implications of this new and rather exciting technology, at least at the highest levels.

Graph discovery

Because they deal with properties and connections, graph databases represent rich pools of information, often hidden until discovered. Discovery is a means by which a large collection of related data is mined for new insights, without a strong precognition of what these insights might be.

The graph database wasn't initially considered a useful tool for discovery. It took a specially designed family of supercomputers to realize the full power of graph discovery. Although it's straightforward to represent graphs, as the volume of triples increases into the billions, the ability to rapidly traverse multiple paths becomes compute-bound in all but the most powerful machines.

This is particularly true in the case of dense graphs, such as tightly woven protein-protein networks. Here, detailed graph queries can overwhelm less capable computational systems. The graph supercomputer, built from the ground up to traverse graphs, overcomes time and capacity limitations. Such devices, some complete with Hadoop analysis tools, recently became available in the high-end graph-database marketplace via Cray.

The high-end graph supercomputer, built for discovery, brings great promise. For example, it can support a detailed build-out of the complex relationships between the ocean and atmosphere that compose climatic conditions. In a time of great climate change, further discovery of indirect, nonlinear causes and effects becomes increasingly crucial. Likewise, a graph supercomputer could hasten a discovery concerning the spread of Ebola in Western Africa, which could serve to stem the spread of the disease. Figure 2 illustrates the notion of discovery using a graph database.

Discovery: Privacy and security

Graph discovery, which has great promise for resolving complex interrelated problems, presents privacy and security concerns, however. For example, one's identity can be further laid bare if the graph supercomputer becomes the device of choice to further mine our social and financial transactions for purposes of surveillance, targeted advertising, and other overt exploits that tend to rob individuals of their privacy.

While perhaps it's an alien thought in a thriving free enterprise system, placing an ethical bar on the acceptable extent of intrusion into one's personal life might well prove necessary for financial, if not constitutional, reasons. It's quite acceptable to expect law enforcement to use all necessary means to remove real threats from our midst, but at what expense to the rest of society? Likewise, those anxious to move their products will take advantage of every opportunity to do so by whatever means possible, but at what personal price for those targeted? Such high-end exploitation amounts to nothing more than a projection of currently established trends.

In the design of such socioeconomic studies, especially involving a wide range of social and business transaction relationships, the security bar must be set exceedingly high. Any intentionally perpetrated breach could be far more devastating than recent massive hacks against corporations such as credit-card issuers or motion-picture companies. This is further exacerbated by the notion that the Internet of Anything (IoA) consists of myriads of sensors, actuators, and mobile devices, all of which seem to be optimized for privacy leakage.

Graph knowledge management

The concept of the node-arc-node triple strongly resembles the subject-predicate-object relationships expressed in the Resource Description Framework (RDF) descriptive language. RDF creates a level of formal expression that lets us describe and reason about the data housed in a graph database. Moreover, RDF nicely feeds a formal ontology, thus permitting a rigorous semantic definition of terms. The "how much is enough" question, however, might take years to resolve with regard to a tolerable degree of practical formalization.

Together, RDF and a formal ontology speak to the World Wide Web Consortium (W3C) view of linked data: an endeavor to make reusable structured knowledge generally available in a common referential format via the Web. There's a downside though. Whereas it's relatively straight-

The Top 5 Use Cases for Graph Databases



Discover the five most impactful and profitable use cases of graph technologies and how to leverage them for greatest competitive advantage.

[Read the White Paper](#)

forward to convert highly structured data, such as well-organized spreadsheets and databases, into RDF, only high-end tools can reliably convert unstructured data into RDF, and that carries some restrictive caveats.

Not all graph databases, however, require RDF-style triple representation. A number of thriving commercial graph databases employ triples in their own unique ways without engaging RDF. Many offer a number of attractive features, such as graph visualization, backup, and recovery. Emil Eifrem, founder and CEO of Neo Technology, expects that these tools will attract the corporate nod and the consumer base will continue to grow, pushing the graph-database industry from 2 percent to [an estimated 25 percent](#) of the database market by 2017. Of course, many companies employ their own languages and techniques for data management. A real need exists for standards that, at a minimum, support data transportability.

Knowledge management: Privacy and security

Security, particularly for proprietary architectural designs, must be taken into consideration. If Web sharing is envisioned as a reasonable means to generate a lot of system-representative triples from resident experts, a secure portal to the RDF data store becomes exceedingly important. User authentication and verification also become important.

Although knowledge management is perhaps less extensive than discovery, related databases still might possess specific identity attributes that must be well protected. Front-end provisions must assure the existence of both security against intrusion and the privacy of any personal data contained in the graph database. Failure to offer adequate protection could disqualify otherwise prom-

ising candidate graph databases for their vulnerability to attack.

Graph prediction

In dynamic circumstances involving an unfolding process such as weather or economic trends, the ability to predict future behavior becomes highly desirable.

Graph representations facilitate predictions because they let us both qualify and quantify a system represented as a network. The ability to assign properties to nodes and arcs — such as location, time, weights, or quantities — lets us qualitatively evaluate the graph on the basis of similar properties. More importantly, quantitative techniques let us evaluate metrics inherent in almost all graphs.

The ability to apply proven metrics to graphs means that their characteristics might be quantified to allow an objective evaluation of the graph. In cases where graph data is dynamic, such as in an ongoing process, a powerful predictive capability becomes possible, assuming the data stream is accessible. This approach presumes combinations of graph theory and combinatorial mathematics can be applied against a real-time data stream. Moreover, various graph configurations could be classified based on their metrics. Such classification templates, each with a graph signature based on its metrics, could then permit identification of and a predictive baseline for similar graphs as they arise.

Prediction: Security and privacy

Current cybersecurity best practices suggest the importance of taking a snapshot of a system under study to determine its security and privacy vulnerabilities, leading to the accreditation of systems proven to be “secure.” The fallacy of such practice is that most systems are influenced by ever-changing environments, which serve to

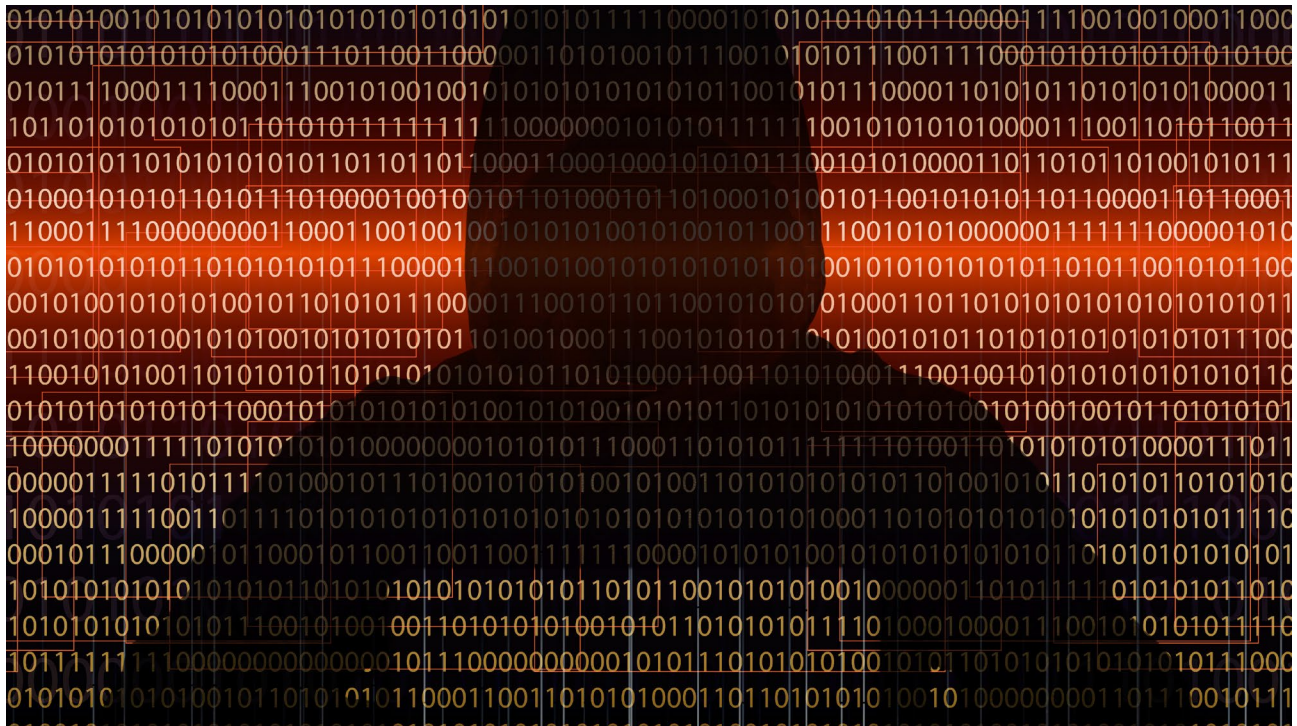
change systemic behaviors over time. Thus, the accreditation is good only for the moment of time at which the snapshot was taken.

Given their growing sophistication, graph databases offer the potential to let us monitor dynamic change in near real time. By monitoring data streams with quantitative methods, looking for anomalous node or changing relationship patterns, we could detect and investigate intrusions and other security breaches early on, quickly prosecuting any identified perpetrators.

From the predictive perspective, data integrity must take a front seat. Data provenance becomes a crucial issue because the stakes of prediction are high. The results of a prediction are as accurate as the data underlying the predictive tools. False data could gravely affect outcomes and literally endanger security. Consider the consequence of a faulty predictive model for disaster relief, which calls for distributing resources in an unaffected region as opposed to the affected region. In this regard, good security practice results in the highest ethical standards of applied science.

Although graph databases hold great promise in a world wrapped in networks of all kinds, they also contain some inherent security risks that have yet to be fully understood, much less appreciated. Rather than piling on the bandwagon, the prudent IT professional must carefully evaluate potential risks in the context of the intended operating environment and perform the necessary tradeoffs to achieve acceptable levels of security and data protection. If security and privacy issues surrounding relatively new technologies, such as increasingly popular graph databases, aren’t considered up-front, they become far more costly to implement downstream. ■

Graph Databases in the Enterprise: Fraud Detection



By Jim Webber and Ian Robinson

Banks and insurance companies lose billions of dollars every year to fraud.

Traditional methods of fraud detection fail to minimize these losses since they perform discrete analyses that are susceptible to false positives (and false negatives). Knowing this, increasingly sophisticated fraudsters have developed a variety of ways to exploit the weaknesses of discrete analysis.

Graph databases, on the other hand, offer [new methods of uncovering fraud rings](#) and other complex scams with a high level of accuracy through advanced contextual link analysis, and they are capable of stopping advanced fraud scenarios in real time.

The key challenges in fraud detection

Between the enormous amounts of data available for analysis and today's experienced fraud rings (and solo fraudsters), fraud-detection professionals are beset

with challenges. Here are some of their biggest:

- Complex link analysis to discover fraud patterns — Uncovering fraud rings requires you to traverse data relationships with high computational complexity, a problem that's exacerbated as a fraud ring grows.
- Detect and prevent fraud as it happens — To prevent a fraud ring, you need real-time link analysis on an interconnected dataset, from the time a false account is created to

when a fraudulent transaction occurs.

- Evolving and dynamic fraud rings — Fraud rings are continuously growing in shape and size, and your application needs to detect these fraud patterns in this highly dynamic and emerging environment.

Overcoming fraud detection challenges with graph databases

While no fraud-prevention measures are perfect, significant improvements occur when you

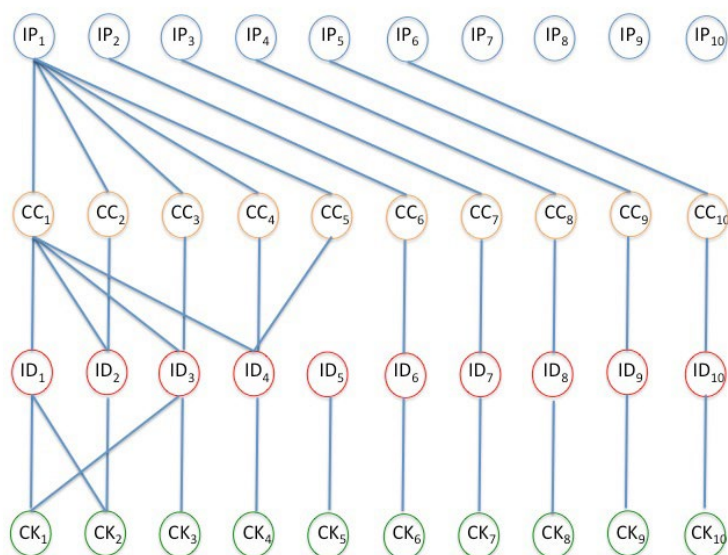


Figure 1. A graph of a series of transactions from different IP addresses with a likely fraud event occurring from IP1, which has carried out multiple transactions with five different credit cards.

look beyond individual data points to the connections that link them.

Understanding the connections between data, and deriving meaning from these links, doesn't necessarily mean gathering new data. You can draw significant insights from your existing data simply by reframing the problem in a new way: as a [graph](#).

Unlike most other ways of looking at data, graphs are designed to express relatedness. Graph databases uncover patterns that are difficult to detect with traditional representations such as tables. An increasing number of companies use graph databases to solve a variety of connected-data problems, including fraud detection.

Example: E-commerce fraud

As our lives become increasingly digital, a growing number of financial transactions are conducted online. Fraudsters have adapted quickly to this trend and have devised clever ways to defraud online payment systems.

While this type of activity can and does involve criminal rings, even a single well-informed

fraudster can create a large number of synthetic identities to carry out sizeable schemes.

Consider an online transaction with the following identifiers: user ID, IP address, location, tracking cookie, and credit-card number. Typically, the relationships between these identifiers should be (almost) one-to-one. Some variations naturally account for shared machines, families sharing a single credit-card number, individuals using multiple computers, and the like.

However, as soon as the relationships between these variables exceed a reasonable number, fraud should be considered as a strong possibility. The more interconnections exist amongst identifiers, the greater the cause for concern. Large and tightly knit graphs are very strong indicators that fraud is taking place.

See Figure 1 for an example.

By putting checks into place and associating them with the appropriate event triggers, such schemes can be discovered before they are able to inflict significant damage.

Triggers can include events such as logging in, placing an order, or registering a credit card —

any of which can cause the transaction to be evaluated against the fraud graph. Fan-out might be skipped, but complex graphs can be flagged as a possible instance of fraud.

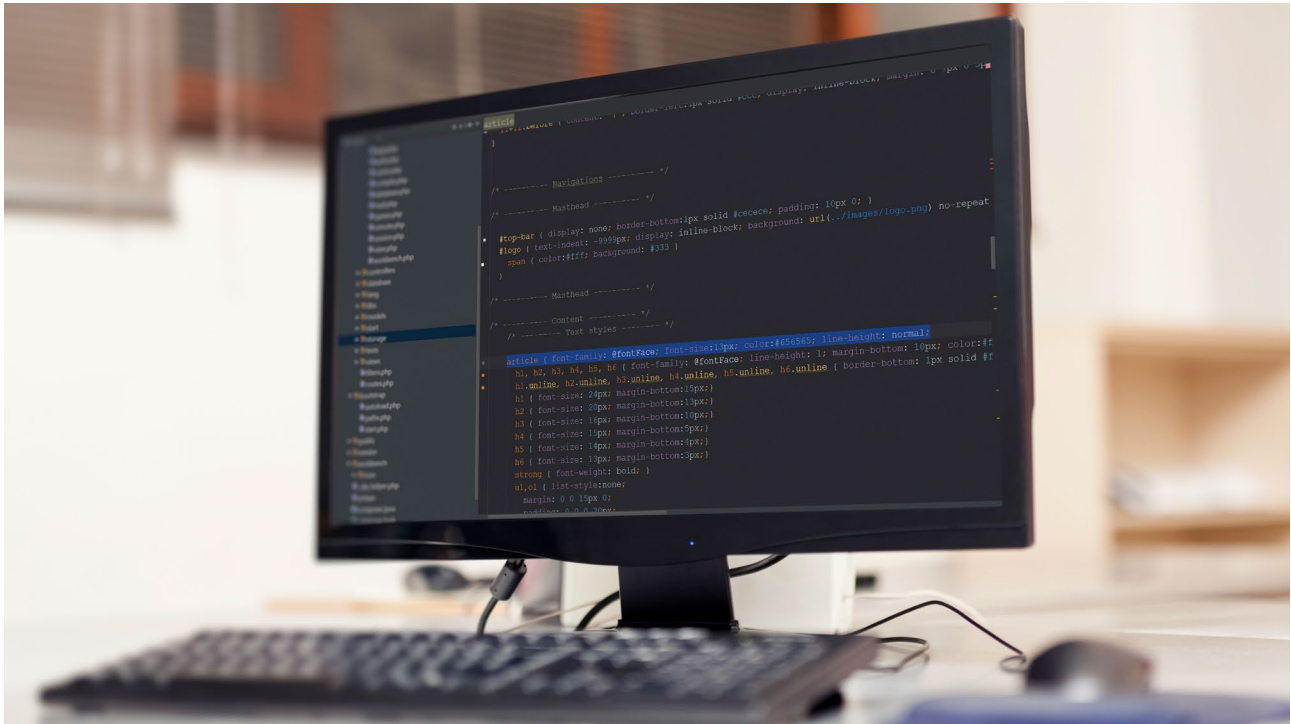
Conclusion

When it comes to **graph-based fraud detection**, you need to augment your fraud-detection capability with link analysis. That being said, two points are clear:

- As business processes become faster and more automated, the time margins for detecting fraud are narrowing, increasing the need for a real-time solution.
- Traditional technologies are not designed to detect elaborate fraud rings. Graph databases add value through analysis of connected data points.

Graph databases are the ideal enabler for efficient and manageable fraud-detection solutions. Graph databases uncover a variety of important fraud patterns from fraud rings and collusive groups to educated criminals operating on their own — all in real time. ■

Full-Stack Web Development Using Neo4j



Brian Underwood is a software engineer and lover of all things data. As a developer advocate for Neo4j and co-maintainer of the Neo4j Ruby gem, Brian regularly lectures and writes on his blog about the power and simplicity of graph databases. Brian is currently traveling the world with his wife and son. Follow Brian on Twitter or join him on LinkedIn.

When building a full-stack Web application, you have many choices for the database that you will put on the bottom of the stack. You want a database that is dependable, certainly, but which also allows you to model your data well. Neo4j is a good choice as the foundation of your Web-application stack if your data model contains a lot of connected data and relationships.

What is Neo4j?

Neo4j is a graph database, which simply means that instead of storing data in tables or collections, it stores data as nodes and relationships between nodes. In Neo4j, both nodes and relationships can contain properties with values. In addition:

- Nodes can have zero or more labels (like “Author” or “Book”).
- Relationships have exactly one type (like “WROTE” or “FRIEND_OF”).
- Relationships are always directed from one node to another (but can be queried regardless of direction).

Why Neo4j?

To choose a database for a Web application, you should consider what it is that you want from it. Top criteria include:

- Is it easy to use?
- Will it let you easily respond to changes in requirements?
- Is it capable of high-performance queries?
- Does it allow for easy data modeling?
- Is it transactional?
- Does it scale?
- Is it fun (sadly, an often overlooked quality in a database)? ▶

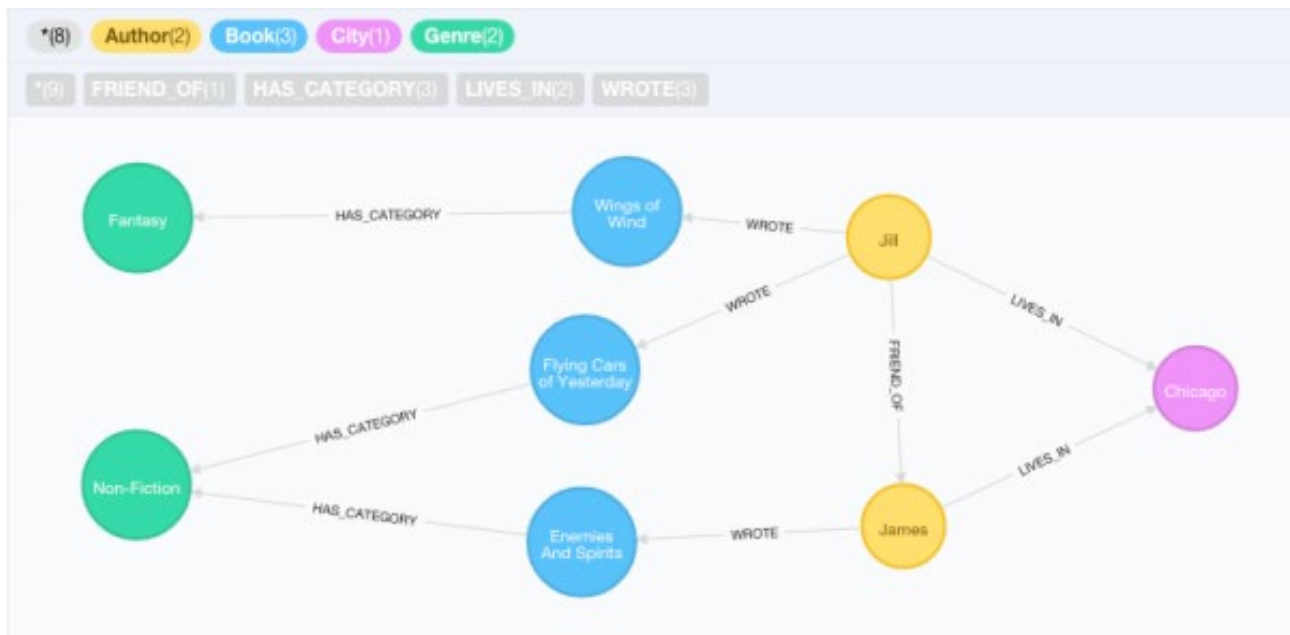


Figure 1. The Neo4j Web console.

In this respect, Neo4j fits the bill nicely:

- It has its own, easy-to-learn query language (Cypher).
- It's schema-less, which allows it to be whatever you want it to be.
- It can perform queries on highly related data (graph data) much faster than traditional databases.
- It has an entity and relationship structure that naturally fits human intuition.
- It supports ACID-compliant transactions.
- It has a high-availability mode for query throughput, scaling, backups, data locality, and redundancy.
- It's hard to grow tired of its visual query console.

When to not use Neo4j?

While Neo4j, as a graph NoSQL database, has a lot to offer, no solution can be perfect. Some use cases where Neo4j isn't as good of a fit are for:

- recording large amounts of event-based data such as log entries or sensor data,
- large-scale distributed data processing like with Hadoop,
- binary data storage, and
- structured data that's a good candidate for storage in a relational database.

In the sample graph at the beginning of this section, you can see a graph of Author, City, Book, and Category and the relationships that tie these together. To use Cypher to find all authors in Chicago and show that result in the Neo4j console, you could execute the following search.

```
001 MATCH
002   (city:City)<-[:LIVES_IN]-
003   (:Author)-[:WROTE]->
004   (book:Book)-[:HAS_CATEGORY]-
005   >(category:Category)
006 WHERE city.name = "Chicago"
007 RETURN *
```

Note the ASCII-art syntax that has parentheses surrounding nodes and arrows representing the relationships that point from one node to another. This is Cypher's way to match a given subgraph pattern.

Of course, Neo4j isn't just about pretty graphs. If you wanted to count the categories of books by the location (City) of the author, you can use the same MATCH pattern to return a different set of columns, like so:

```
001 MATCH
002   (city:City)<-[:LIVES_IN]-
003   (:Author)-[:WROTE]->
004   (book:Book)-[:HAS_CATEGORY]-
005   >(category:Category)
006 RETURN city.name, category.name,
007        COUNT(book)
```

That would return the following:

city.name	category.name	COUNT(category)
Chicago	Fantasy	1
Chicago	Non-Fiction	2

While Neo4j can handle big data, it isn't Hadoop, HBase, or Cassandra and you won't typically be crunching massive (petabyte) analytics directly in your Neo4j database. But when you're interested in serving up information about an entity and its data neighborhood (like you would when generating a webpage or an API result), it is a great choice for

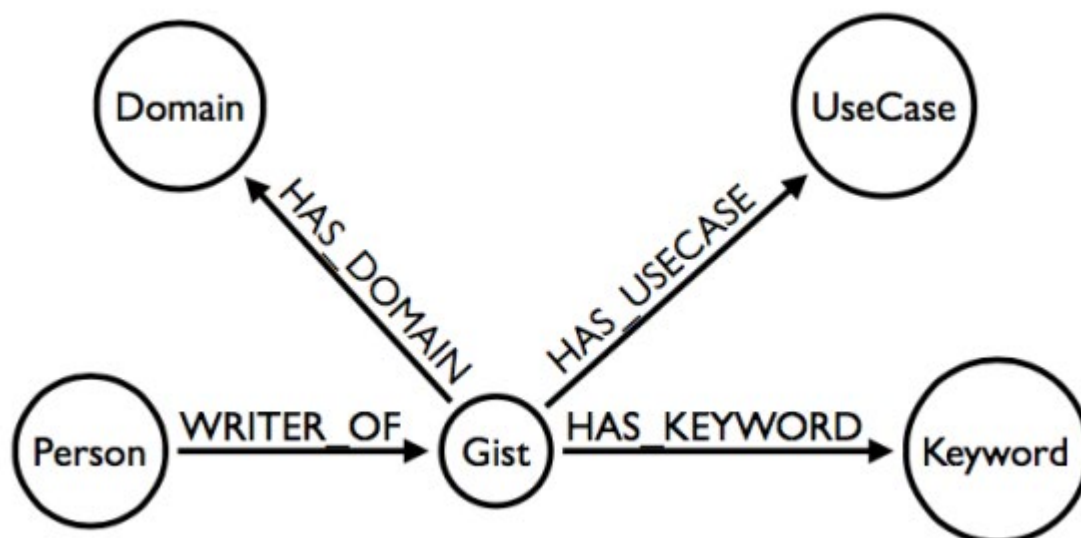


Figure 2

anything from simple CRUD access to a complicated, deeply nested view of a resource.

Which stack should you use with Neo4j?

All major programming languages have support for Neo4j via the HTTP API, either via a basic HTTP library or via a number of native libraries that offer higher-level abstractions. Since Neo4j is written in Java, all languages that have a JVM interface can take advantage of the high-performance APIs in Neo4j.

Neo4j also has its own “stack” to allow you to choose different access methods ranging from easy access to raw performance. It offers:

- a HTTP API for making Cypher queries and retrieving results in JSON,
- an “unmanaged extension” facility in which you can write your own endpoints for your Neo4j database,
- A Java API for specifying traversals of nodes and relationships at a higher level,
- A low-level batch-loading API for massive initial data ingestion, and
- A core Java API for direct access to nodes and relationships for maximum performance.

An application example

I recently took on a project to expand a Neo4j-based application. The application (which

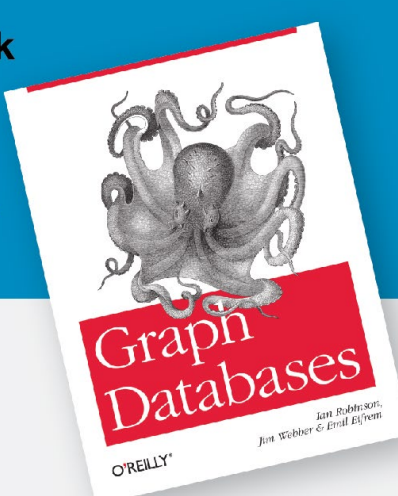
you can see at graphgist.neo4j.com) is a portal for GraphGists. A GraphGist is a simple AsciiDoc text file (with images if need be) that describes the data model, setup, and use-case queries to be executed. A reader’s browser interactively renders the file and visualizes it live. A GraphGist is much like an [IPython notebook](#) or an interactive white paper. It also allows readers to write their own queries to explore the dataset from the browser.

Neo Technology, the creators of Neo4j, wanted to provide a showcase for GraphGists created by the community, and this was my project. Neo4j was used as the back end, of course, but for the rest of the stack I used [Node.js](#) with [Express.js](#) and the Neo4j package, [Angular.js](#), and [Swagger UI](#). ►

FREE O'Reilly Book

Graph Databases

The definitive resource on graph databases



Download Now

All of the code is open-sourced and available [on GitHub](#).

This GraphGist portal is conceptually a simple app, providing a list of GraphGists and allowing users to view details about each as well as the GraphGist itself. The data domain consists of Gist, gist categories of Keyword/Domain/UseCase, and Person (for the authors). (Figure 2)

Now that you're familiar with the model, I'd like to give you a quick intro to the Cypher query language before we dig deeper. For example, if we wanted to return all gists and their keywords, we could do the following.

```
001 MATCH (gist:Gist)-[:HAS_KEYWORD]->(keyword:Keyword)
002 RETURN gist.title, keyword.name
```

This would give a table with one row for every combination of Gist and Keyword, just like an SQL join. If we want to find all Domains for which a given Person has written Gists, we could perform the following query.

```
001 MATCH (person:Person)-[:WRITER_OF]->(gist:Gist)-[:HAS_DOMAIN]->(domain:Domain)
002 WHERE person.name = "John Doe"
003 RETURN domain.name, COUNT(gist)
```

This would return another table of results. Each row of the table would have the name of the Domain accompanied by the number of Gists that the Person has written for that Domain. There's no need for a GROUP BY clause because when we use an aggregate function like COUNT(), Neo4j automatically groups by the other columns in the RETURN clause.

Let's look at a real-world query from our app. When building the portal, it was useful to be able to provide a way to make just one request to the database and retrieve all the data that we need with almost exactly the format in which we want it.

Let's build the query that the portal's API uses (you can view it [on GitHub](#)). First, we need to match the Gist in question by its title property to any related Gist nodes.

```
001 // Match Gists based on title
002 MATCH (gist:Gist) WHERE gist.title =~ {search_query}
003 // Optionally match Gists with the same keyword
004 // and pass on these related Gists with the
005 // most common keywords first
006 OPTIONAL MATCH (gist)-[:HAS_KEYWORD]->(keyword)<-[:HAS_KEYWORD]-(related_gist)
```

There are a couple of things to note here. Firstly, the WHERE clause is matching the title using a regular expression (that's the =~ operator) and a parameter.

Parameters are features of Neo4j that separate the query from the data that the query uses. Parameters let Neo4j cache queries and query plans, and it also means that you don't need to worry about query-injection attacks. Secondly, we're using an OPTIONAL MATCH clause here, which simply means that we still want to return the Gist that we're originally matching with even if there are no related gists.

Now, let's take that part of the query and expand on it by replacing the RETURN clause with a WITH clause. (Code 1)

The COLLECT() in the RETURN transforms a result with pairs of Gist and related_gist nodes so that each row has the Gist only once, along with an array of related_gist nodes. Inside COLLECT(), we specify only the data we need from the related gists in order to reduce the size of our response.

Lastly, we'll take the query so far and use WITH one last time. (Code 2)

In this last part, we optionally match all associated Domain, UseCase, Keyword, and Person nodes and collect them together just like we did with related Gists. Rather than having a flat, denormalized result, we now return a list of Gist nodes with arrays of associated "has many" relationships with no duplication. Pretty cool!

If tables of data are too old school for you, Cypher can return objects as well.

```
001 RETURN
002 {
003   gist: gist,
004   domains: collect(DISTINCT domain.name) AS domains,
005   usecases: collect(DISTINCT usecase.name) AS usecases,
006   writers: collect(DISTINCT writer.name) AS writers,
007   keywords: collect(DISTINCT keyword.name) AS keywords,
008   related_gists: related
009 }
010 ORDER BY gist.title
```

Traditionally in a decently sized Web application, you need a number of database calls to populate the HTTP response. Even if you can execute queries in parallel, it is often necessary to get the results of one query before you can make a second to get related data. In SQL, you can execute complicated and expensive joins on tables to get results from many tables in one query, but anybody who has done more than a couple of SQL joins in the same query knows how quickly that can get complicated — not to mention that the database still needs to scan tables or indexes to get the associated data. In Neo4j, retrieving entities via relationships uses pointers directly to the

```

001 MATCH (gist:Gist) WHERE gist.title =~ {search_query}
002 OPTIONAL MATCH (gist)-[:HAS_KEYWORD]->(keyword)<-[:HAS_KEYWORD]-(related_gist)
003 WITH gist, related_gist, COUNT(DISTINCT keyword.name) AS keyword_count
004 ORDER BY keyword_count DESC
005
006 RETURN
007     gist,
008     COLLECT(DISTINCT {related: { id: related_gist.id, title: related_gist.title,
    poster_image: related_gist.poster_image, url: related_gist.url }, weight: keyword_
    count }) AS related

```

Code 1

```

001 MATCH (gist:Gist) WHERE gist.title =~ {search_query}
002 OPTIONAL MATCH (gist)-[:HAS_KEYWORD]->(keyword)<-[:HAS_KEYWORD]-(related_gist)
003 WITH gist, related_gist, COUNT(DISTINCT keyword.name) AS keyword_count
004 ORDER BY keyword_count DESC
005
006 WITH
007     gist,
008     COLLECT(DISTINCT {related: { id: related_gist.id, title: related_gist.title,
    poster_image: related_gist.poster_image, url: related_gist.url }, weight: keyword_
    count }) AS related
009
010 // Optionally match domains, use cases, writers, and keywords for each Gist
011 OPTIONAL MATCH (gist)-[:HAS_DOMAIN]->(domain:Domain)
012 OPTIONAL MATCH (gist)-[:HAS_USECASE]->(usecase:UseCase)
013 OPTIONAL MATCH (gist)<-[:WRITER_OF]-(writer:Person)
014 OPTIONAL MATCH (gist)-[:HAS_KEYWORD]->(keyword:Keyword)
015
016 // Return one Gist per row with arrays of domains, use cases, writers, and
    keywords
017 RETURN
018     gist,
019     related,
020     COLLECT(DISTINCT domain.name) AS domains,
021     COLLECT(DISTINCT usecase.name) AS usecases,
022     COLLECT(DISTINCT keyword.name) AS keywords
023     COLLECT(DISTINCT writer.name) AS writers,
024 ORDER BY gist.title

```

Code 2

related nodes so that the server can traverse right where it needs to go.

That said, there are a couple of downsides to this approach. While it's possible to retrieve all of the data required in one query, the query is quite long. I haven't yet found a way to modularize it for reuse. Along the same lines, we might want to use this same endpoint in another place but show more information about the related gists. We could modify the query to return that data but then it would be returning data unnecessary for the original use case.

We are fortunate to have many excellent database choices. While relational databases are still the best choice for storing structured data, NoSQL databases are a good choice for managing semi-structured, unstructured, and graph data. If you have a data model with lot of connected data and want a database that is intuitive, fun, and fast, you should get to know Neo4j.

This article was authored by Brian Underwood with contributions from Michael Hunger. ■

PREVIOUS ISSUES



Cloud Migration

In this eMag, you'll find practical advice from leading practitioners in cloud. Discover new ideas and considerations for planning out workload migrations.

Business, Design and Technology: Joining Forces for a Truly Competitive Advantage



This eMag offers readers tactical approaches to building software experiences that your users will love. Break down existing silos and create an environment for cross-collaborative teams: placing technology, business and user experience design at the core.

Architectures You Always Wondered About



In this eMag we take a look at the state of the art for the microservice architectural style in both theory and practice. Amongst others Martin Fowler talks about Microservice trade-offs, Eric Evans explores the interplay of Domain-Driven Design, microservices, event-sourcing, and CQRS, Randy Shoup talks about Lessons from Google and eBay, and Yoni Goldberg describes Gilt's experience.

Description, Discovery, and Profiles



This eMag focuses on three key areas of "meta-language" for Web APIs: API Description, API Discovery, and API Profiles. You'll see articles covering all three of these important trends as well as interviews with some of the key personalities in this fast-moving space.