Compliments of Neo Technology

Early Release

RAW & UNEDITED

Graph Databases

Ian Robinson,
Jim Webber & Emil Eifrem

O'REILLY®

Graph Databases

Ian Robinson, Jim Webber, and Emil Eifrem

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Graph Databases

by Ian Robinson, Jim Webber, and Emil Eifrem

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Proofreader: FIX ME!
Cover Designer: FIX ME!
Interior Designer: FIX ME!
Illustrator: FIX ME!

Revision History for the :

2013-02-25: Early release revision 1

See *http://oreilly.com/catalog/errata.csp?isbn=9781449356262* for release details.

ISBN: 978-1-449-35626-2

Table of Contents

Part I. GDB Book

PART I

GDB Book

CHAPTER 1
Introduction

Graph databases address one of the great macroscopic business trends of today: leveraging complex and dynamic relationships to generate insight and competitive advantage. Whether we want to understand relationships between customers, elements in a telephone or datacenter network, entertainment producers and consumers, or genes and proteins, the ability to understand and analyze vast graphs of highly-connected data will be a key factor in the determining which companies outperform their competitors over the coming decade.

For data of any significant size or value, graph databases are the best way to represent and query connected data. While large corporates realized this some time ago, creating their own proprietary graph processing technologies, we're now in an era where that technology has rapidly become democratized. Today, general-purpose graph databases are a reality, allowing mainstream users to experience the benefits of connected data without having to invest in building their own graph infrastructure.

What's remarkable about this renaissance of graph data and graph thinking is that graph theory itself is not new. Graph theory was pioneered by Euler in the 18th century, and has been actively researched and improved by mathematicians, sociologists, anthropologists, and others since then. However, it is only in the last few years that graph theory and graph thinking have been applied to information management. Graph databases have been proven to solve some of the more relevant data management challenges of today, including important problems in the areas of social networking, master data management, geospatial, recommendations engines, and more. This increased focus on graph is driven by the massive commercial successes of companies such as Facebook, Google, and Twitter, all of whom have centered their business models around their own proprietary graph technologies, together with the introduction of general purpose graph databases into the technology landscape (a very recent phenomenon in informatics).

Graphs and graph databases are tremendously relevant to anyone seeking to model the real world. The real world—unlike the forms-based model behind the relational data-

base—is rich and interrelated: uniform and rule-bound in parts, exceptional and irregular in others. Unlike relational databases, which require a level of erudition and specialized training to understand, graph databases store information in ways that much more closely resemble the ways humans think about data.

One of the unique things about graph databases that makes them especially adapted to modelling the real world is that they elevate relationships to be first-class citizens of the data model. We take for granted the fact that the items stored in a database—whether they are rows, documents, objects, or nodes—each merit their own rich set of descriptors. A Person, for example, is often attributed with rich metadata such as name, gender, and date of birth. What may not be so clear, however, is that metadata also exists in the relationships between people. Unlike a relational database, where a relationship is effectively just a runtime constraint, part of what makes graph databases special is that they put relationships on the same level as the data items themselves. A "friend" relationship can therefore include a "friends since" datestamp, together with properties describing the degree and quality of friendship, and so on. As information processing continues to evolve, the next frontier arguably lies in the ability to capture, analyze, and understand these rich relationships.

We believe, and many of the major analysts agree, that in a few years the "Not Only SQL" conversation that appears so pertinent today will cease to be about the "Not", and will instead focus on what *is*: a heterogeneous data landscape where post-relational technologies sit alongside relational. There is no question that graph databases, which are currently recognized as one of the four major types of NOSQL database, will be one of the technology categories from which future data architects will choose the best tool for the job at hand.

The purpose of this book is to introduce graphs and graph databases to technology workers, including developers, database professionals, and technology decision makers. Reading this book, you will come away with a practical understanding of graph databases. We show how data is "shaped" by the graph model, and how it is queried, reasoned about, understood and *acted upon* using a graph database. We discuss the kinds of problems that are well aligned with graph databases, with examples drawn from practical, real-world use cases. And we describe the surrounding ecosystem of complementary technologies, highlighting what differentiates graph databases from other database technologies, both relational and NOSQL.

While much of the book talks about graph data models, it is not a book about graph theory. The annals of graph theory include numerous fascinating and complex academic papers, yet very little graph theory is needed in order to take advantage of the practical benefits afforded by graph databases—provided we understand what a graph is, we're practically there. Before we discuss just how productive graph data and graph databases can be, let's refresh our memories about graphs in general.

What is a Graph?

Formally a graph is just a collection of vertexes and edges (as we learned in school)--or, in less intimidating language, a set of nodes and the relationships that connect them. We can use graphs to model all kinds of scenarios, from the construction of a space rocket, to a system of roads, and from the supply-chain or provenance of foodstuff, to medical history for populations, and beyond. Graphs are general-purpose and expressive, allowing us to model entities as nodes and their semantic contexts using relationships.

For example, Facebook's data is easily represented as a graph. In Figure 1-1 we see a small network of friends. The relationships are key here in establishing the semantic context: namely, that Billy considers Harry to be a friend, and that Harry, in turn, considers Billy to be a friend. Ruth and Harry have likewise expressed their mutual friendship, but sadly, while Ruth is friends with Billy, Billy hasn't (yet) reciprocated that friendship.

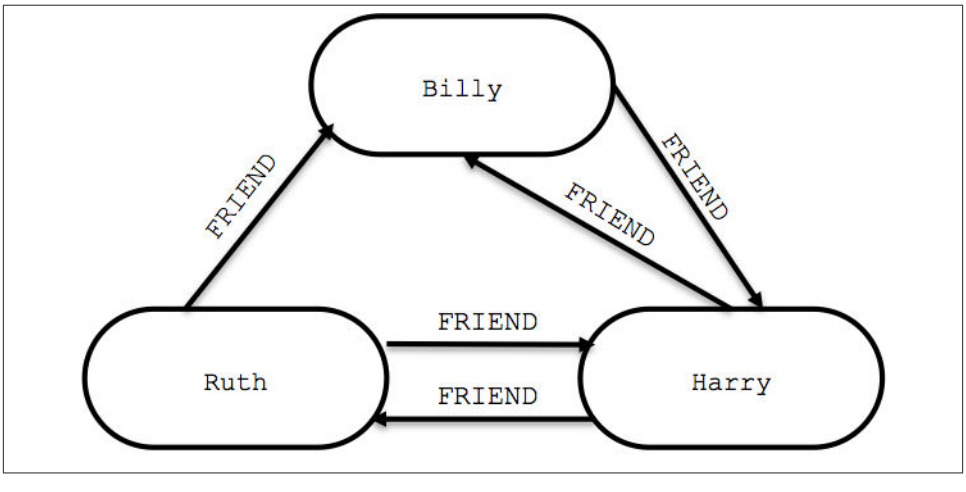


Figure 1-1. A small social graph

Of course, Facebook's real graph is hundreds of millions of times larger than the example in Figure 1-1, but it works on precisely the same principles, plus a few extra features: as well as being "friends" with one another, Facebook users can express consistency of opinion by adding "like" links to the things they have positive sentiments about. No problem: we can do the same. In Figure 1-2 we've expanded our sample network to include "like" relationships.

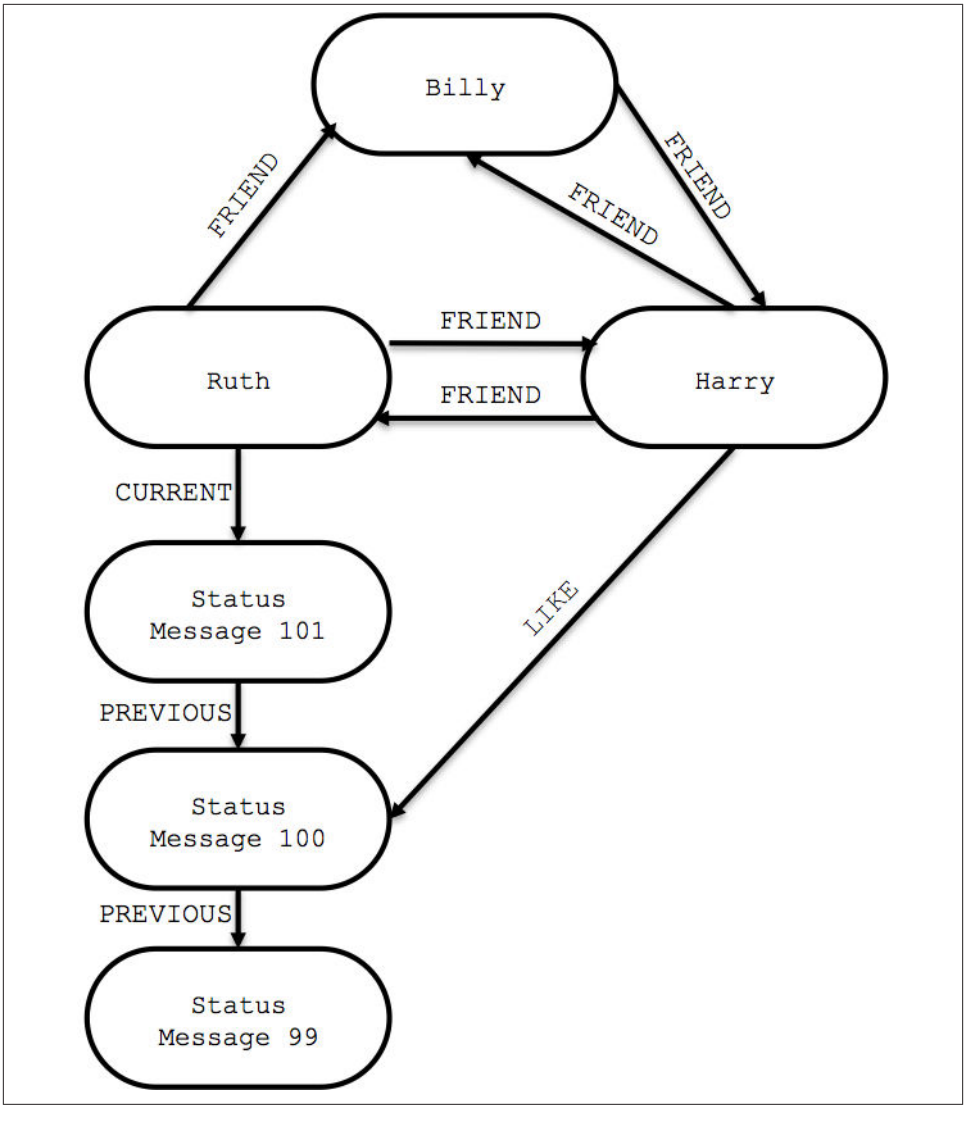


Figure 1-2. Liking posts on Facebook

Though simple, the graph in Figure 1-2 shows the expressive power of the model. It's easy to see that Ruth has made a list of status updates (starting from status message 101 all the way back to status message 99); that the most recent of these updates is linked via a relationship marked `CURRENT`; and that earlier updates are joined into a kind of list by relationships marked `PREVIOUS`, thus creating a timeline of posts. We can also see that Harry `LIKE`s Ruth's status message 100, but hasn't expressed the same sentiments about any of the others.

In discussing Figure 1-2 we've also informally introduced the most popular variant of graph model, the *Property Graph*. A Property Graph has the following characteristics:

- It contains nodes and relationships
- Nodes contain properties (key-value pairs)
- Relationships are named, directed and always have a start and end node
- Relationships can also contain properties

Most people find the Property Graph model intuitive and easy to understand. While simple, it can be used to describe the overwhelming majority of graph use cases in ways that yield useful insight into our data.[1] The Property Graph model is supported by most of the popular graph databases on the market today—including the market leader, Neo4j—and in consequence, it's the model we'll use throughout the remainder of this book.

Making the Connection to NOSQL

Now we've refreshed our memory on graphs, it's time for us to to move on to data and databases. Not only must graph databases support graph modelling, they must also support querying and analysing graph data. Fortunately, we have hundreds of years of graph theory to help with analytics, and decades of research into graph algorithms to help with queries. Today this knowledge has been packaged into a set of database technologies that together bring the power of the graph to the developer community.

It's an exciting time to be working with data, and the current trends around NOSQL and Big Data are fascinating (and fast-moving). To put the expressive power of graph databases into context, we need first to understand what is offered by the non-graph databases.

1. There are other graph models with different constraints. For example property graphs don't natively suit hyperedges without some additional effort, but that's really the only drawback.

CHAPTER 2

The NOSQL Phenomenon

Recent years have seen a meteoric rise in popularity of a family of data storage technologies known as *NOSQL* (a cheeky acronym for *Not Only SQL*, or more confrontationally, *No to SQL*). But NOSQL as a term defines what those data stores are not—they're not SQL-centric relational databases—rather than what they are, an interesting and useful set of storage technologies whose operational, functional, and architectural characteristics are many and varied.

In this chapter we're going to discuss some of the recent forces that have given rise to these NOSQL databases: the exponential growth in data volumes, the rise of connectedness, and the increase in degrees of semi-structure. We define data complexity in terms of these three forces: data size, connectedness and semi-structure. By examining how the different NOSQL technologies address these forces, we'll be better placed to understand the role of graph databases in the rapidly changing world of data technologies.

The Rise of NOSQL

Historically, most enterprise-level Web apps could be run atop a relational database. The NOSQL movement has arisen in response to the needs of an emerging class of applications that must *easily* store and process data which is bigger in volume, changes more rapidly, and is more structurally varied than can be dealt with by traditional RDBMS deployments. For these applications, data has begun to move out of the SQL sweet-spot.

It's no surprise that as storage has increased dramatically, *volume* has become the principal driver behind much of the penetration of NOSQL stores into organizations. Volume may be defined simply as:

- Volume: the size of the stored data

As is well known, large datasets become unwieldy when stored in relational databases; in particular, query execution times increase as the size of tables and the number of joins grow (so called *join pain*). This isn't the fault of the databases themselves; rather, it is an aspect of the underlying data model, which builds a set of all possible answers to a query before filtering to arrive at the correct solution.

In an effort to avoid joins and join pain, and thereby cope better with extremely large data sets, the NOSQL world has adopted several alternatives to the relational model. Though more adept at dealing very large data sets, these alternative models tend to be less expressive than the relational one (with the exception of the graph model, which is *more* expressive).

But volume isn't the only problem modern Web-facing systems have to deal with. Besides being big, today's data often changes very rapidly. This is the *velocity* metric:

- Velocity: the rate at which data changes over time

Velocity is rarely a static metric: internal and external changes to a system and the context in which it is employed can have considerable impact on velocity. Coupled with high volume, variable velocity requires data stores to not only handle sustained levels of high write-loads, but also deal with peaks.

There is another aspect to velocity, which is the rate at which the structure of the data changes. In other words, as well as the value of specific properties changing, the overall structure of the elements hosting those properties can change as well. This commonly occurs for two reasons. The first is fast-moving business dynamics: as the business changes, so do its data needs. The second is that data acquisition is often an experimental affair: some properties are captured "just in case", others are introduced at a later point based on changed needs; the ones that prove valuable to the business stay around, others fall by the wayside. Both these forms of velocity are problematic in the relational world, where high write loads translate into a high processing cost, and high schema volatility has a high operational cost.

While commentators have later added other useful requirements to the original quest for scale, the final key aspect is the realization that data is far more varied than we've been comfortably been able to cope with in the relational world — for existential proof think of all those nulls in our tables and the null checks in our code — that has driven out the final widely agreed upon facet, *variety* which we can define as:

- Variety: the degree to which data is regularly or irregularly structured, dense or sparse, and importantly *connected or disconnected*.

The notion of variety manifests itself in different ways. For some data stores it's simply that denormalized documents are easier for developers than shoe-horning data into a

relational schema, for others it's that simple keys and values don't need all the sophisticated features that relational databases provide [1].

Nonetheless the point is that we accept data is no longer necessarily tabular in nature, and retrospectively it's clear that in the modern era not much of that data ever was.

ACID Versus BASE

When we first encounter NOSQL we often consider it in the context of what many of us are already familiar with: relational databases. Although we know the data and query model will be different (after all, there's no SQL!) the consistency models used by NOSQL stores tend to be quite different to mature relational databases to support the volume, velocity and variety of data for which the general NOSQL term is famed.

That being said, it's worth taking a few moments to explore what consistency features are available to help keep data safe and what trade-offs are involved when using (most) NOSQL stores [2].

In the relational database world, we're all familiar with *ACID* transactions, which have been the norm for some time. The ACID guarantees provide us with a safe environment in which to operate on data:

- Atomic: All operations in a transaction succeed or every operation is rolled back.
- Consistent: On transaction completion, the database is structurally sound.
- Isolated: Transactions do not contend with one another, contentious access to state is moderated by the database so that transactions appear to run sequentially.
- Durable: The results of applying a transaction are permanent, even in the presence of failures.

These properties mean that once a transaction completes, its data is consistent (so-called *write consistency*) and stable on disk (or disks, or indeed in multiple distinct memory locations). However while this is a wonderful abstraction for the application developer, it requires sophisticated locking (causing logical unavailability) and is typically considered to be a heavyweight pattern for most use cases.

In the NOSQL world, ACID transactions have gone out of fashion as stores have loosened requirements for immediate consistency, data freshness, and accuracy in order to gain other benefits like scale and resilience (with the observation that for many domains,

1. For the graph stores that we consider elsewhere, it's that the relational model is semantically weak in comparison.
2. The .NET-based RavenDB has bucked the trend amongst aggregate stores in supporting ACID transactions. As we'll see in subsequent chapters, ACID properties are still upheld by competent graph databases.

ACID transactions are far more pessimistic than the domain actually requires). Instead of using ACID, the term *BASE* has arisen as a popular way of describing the properties of a more optimistic storage strategy.

- Basic Availability: The store appears to work most of the time.
- Soft-state: Stores don't have to be write-consistent, nor do different replicas have to be mutually consistent all the time.
- Eventual consistency: Stores exhibit consistency at some later point (e.g. lazily at read time).

The BASE properties are far looser than the ACID guarantee, and there is no direct mapping between them. A BASE store values availability (since that is a core building block for scale) and does not offer write-consistency (though read your own writes tends to mask this). BASE stores provide a less strict assurance that data will be consistent in the future, perhaps at read time (e.g. Riak), or that data will always be consistent but only for certain processed past snapshots (e.g. Datamic).

Given such loose support for consistency, we as developers need to be far more rigorous in our approach to developing against these stores and cannot any longer rely on the transaction manager to sort out all our data access woes. Instead we must be intimately familiar with the BASE behavior of our chosen stores and work within those constraints.

The NOSQL Quadrants

Having discussed the BASE model that underpins consistency in NOSQL stores, we're ready to start thinking about the numerous user-level data models. To disambiguate these models, we've devised a simple taxonomy in Figure 2-1. That taxonomy shows of the four fundamental types of data store in the contemporary NOSQL space. Within that taxonomy, each store type addresses a different kind of functional use case [3].

3. Though often non-functional requirements strongly influence our choice of database too.

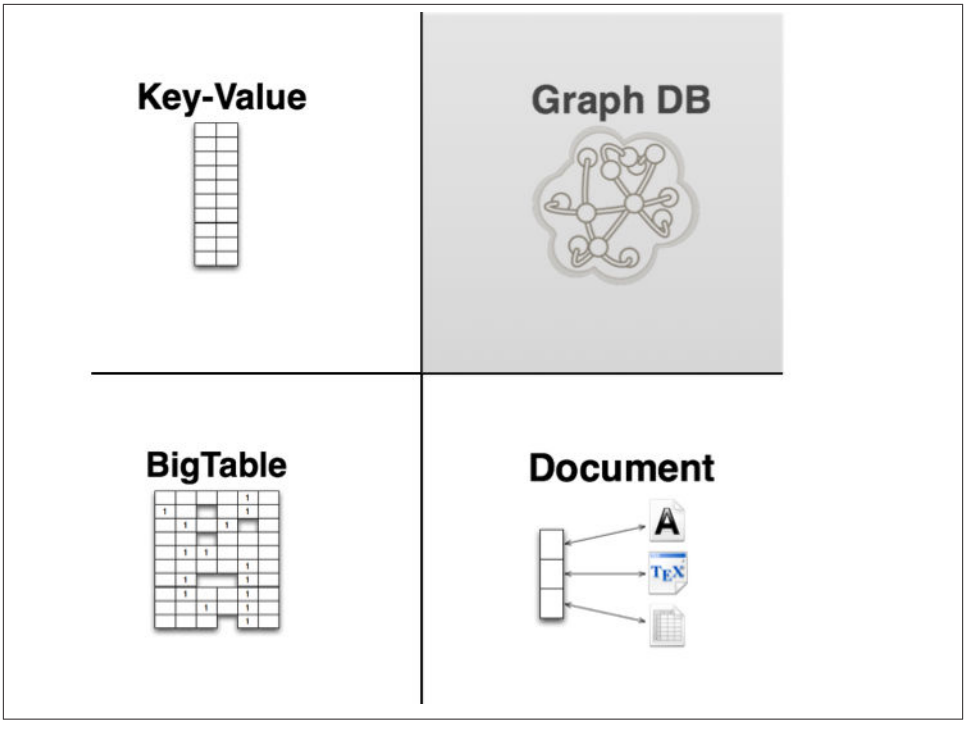


Figure 2-1. The NOSQL store quadrants

In the following sections we'll deal with each of these (with the exception of graphs which will receive a much fuller analysis in subsequent chapters), highlighting the characteristics of the data model, operational aspects, and drivers for adoption.

Document Stores

The document databases are perhaps offer the most immediately familiar paradigm for developers. At its most fundamental level the model is simply that we store and retrieve documents, just like an electronic filing cabinet. Documents tend to comprise the usual property key-value pairs, but where those values themselves can be lists, maps, or similar allowing for natural hierarchies in the document just as we're used to with formats like JSON and XML.

At the simplest level, documents can be stored and retrieved by ID. Providing an application remembers the IDs it's interested in (e.g. usernames) then a document store can act much like a key-value store (of which we'll see more later). But in the general case document stores rely on indexes to facilitate access to documents based on any of their attributes. For example, in an e-commerce scenario a it would be useful to have indexes that represent distinct product types so that they can be offered up to potential

sellers as we see in Figure 2-2. In general indexes are used to reify sets of related documents out of the store for some application use.

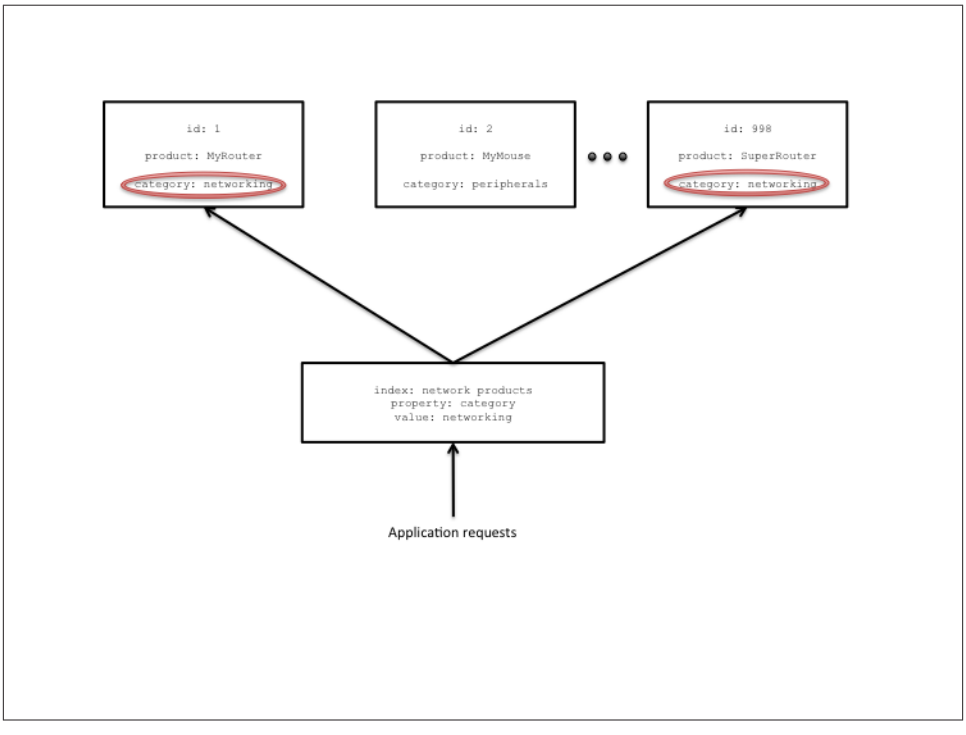


Figure 2-2. Indexing reifies sets of entities in a document store

Much like indexes in relational databases, indexes in a document store allow us to trade write performance (since we have to maintain indexes) for greater read performance (because we examine fewer records to find pertinent data). For write-heavy records, it's worth bearing in mind that indexes might actually degrade performance overall.

Where data hasn't been indexed, queries are typically much slower since a full search of the data set has to happen [4]. This is obviously an expensive task and is to be avoided wherever possible — and as we shall see rather than process these queries internally, it's normal for document database users to externalize this kind of processing in parallel compute frameworks.

Since the data model of a document store is one of disconnected entities, document stores tend to have interesting and useful operational characteristics. For example, since documents are mutually independent, document stores **should** have the ability to scale

4. This isn't the case for graph databases because the graph itself provides a natural adjacency index

horizontally very well since there is no contended state between records at write time, and no need to transact across replicas.

Sharding

Most document databases (e.g. MongoDB, RavenDB) make scaling-out an explicit aspect of development and operations by requiring that the user plan for *sharding* of data across logical instances to support horizontal scale [5]. It's often also puzzlingly cited as a positive reason for embracing such stores, most likely because it induces a (misplaced) excitement that scale is something to be embraced and lauded rather than something to be skilfully and diligently mastered.

At the write level, document databases overwhelmingly provide (limited) transactionality at the individual record level. That is a document database will ensure that writes to a single document are atomically persisted [6], but do not help across document updates. That is, there is no locking support for operating across sets of documents atomically, and such abstractions are left to application code to implement in a domain-specific manner.

However since stored documents are not connected (save through indexes) there are numerous optimistic concurrency control mechanisms that can be used to help reconcile concurrent contending writes for a single document without having to resort to strict locks. In fact some document stores (like CouchDB) have made this a key point of their value proposition: that documents can be held in a multi-master database which automatically replicates concurrently accessed, contended state across instances without undue interference from the user, making such stores very operationally convenient.

In other stores too the database management system may be able to distinguish and reconcile writes to different parts of a document, or even use logical timestamps to reconcile several contended writes into a single logically consistent outcome. This kind of feature is an example of a reasonable, optimistic trade off: it reduces the need for transactions (which we know tend to be latent and decrease availability [7]) by using alternative mechanisms which optimistically provide greater availability, lower latency and higher throughput.

5. Key-value and column stores tend not to require this planning, and subsume allocation of data to replicas as a normal part of their internal implementation

6. assuming the administrator has opted for safe levels of persistence when setting up the database

7. Though optimistic concurrency control mechanisms are useful, we also rather like transactions, and there are numerous example of high-throughput performance transaction processing systems in the literature.

Key-Value Stores

Key-value stores are cousins of the document store family but their lineage comes from the Amazon's Dynamo database [8]. They act like (large, distributed) hashmap data structures where (usually) opaque values are stored and retrieved by key.

As shown in Figure 2-3 the key space of the hashmap is spread across numerous buckets on the network. For fault-tolerance reasons each bucket is replicated onto several machines and such that every bucket is replicated the desired number of times [9] and so that no machine (where possible) is an exact replica of any other so that we can load-balance during recovery of a machine and its buckets (and avoid hotspots causing inadvertent self denial-of-service).

Clients by comparison have an easy task. They store a data element by hashing a domain-specific identifier (key). The hash function is crafted such that it provides a uniform distribution across the available buckets, so that no single machine becomes a hotspot. Given the hashed key, the client can use that address to store the value in a corresponding bucket. A similar process occurs for retrieval of stored values.

8. http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

9. The formula for number of replicas required is given by R = 2F +1 where F is the number of failures we should tolerate.

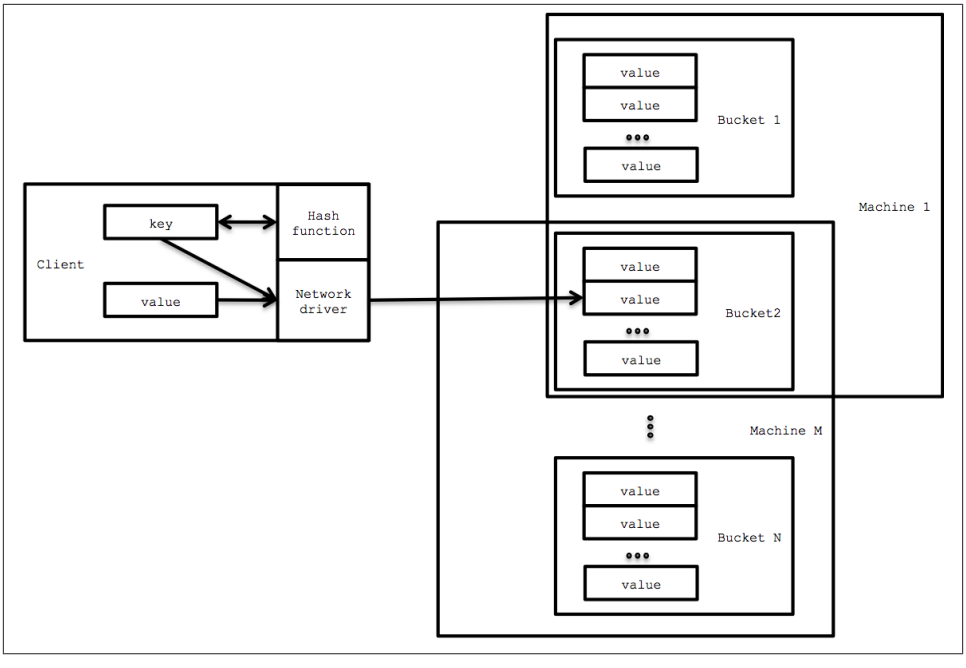


Figure 2-3. Key-Value stores act like distributed hashmap data structures

Consistent hashing

In any computer system failures will occur. In dependable systems those failures are masked via redundant replacements being switched in for faulty components. In a key-value store — as with any distributed database — individual machines will likely become unavailable during normal operation as networks go down, or internal hardware fails.

Such events are to be considered normal for any long running system, but the side effects of recovering from such failures should not itself cause problems. For example if a machine supporting a particular hash-range fails, it should not prevent new values in that range being stored or cause unavailability while internal reorganization occurs.

This is where the technique of *consistent hashing* [10] is often applied. With this technique writes to a failed hash-range are (cheaply) remapped to the next available machine without disturbing the entire stored data set (in fact typically only the fraction of the keys within the failed range need to be remapped, rather than the whole set). When the

10. *http://en.wikipedia.org/wiki/Consistent_hashing*

failed machine recovers (or is replaced), consistent hashing again ensures only a fraction of the total key space is remapped.

Given such a model, applications wishing to store data in, or retrieve data from a key-value store need only know (or compute) the corresponding key. While there are a very large number of possible keys in the key set, in practice keys tend to fall out quite naturally from the application domain. User names and email addresses, Cartesian co-ordinates for places of interest, social security numbers and zip codes are all natural keys for various domains, and so the likelihood of data being "lost" in the store due to a missing key is unlikely in sensibly designed systems.

Although we've dipped into a little bit of basic algorithms and data structures, if we think only about the key-value data model it's plainly quite similar in nature to document stores we discussed earlier. However if we were to draw out a key differentiator in data model between a document store and a key-value store in terms of data model only, it would be the level of *insight* each has into the stored data.

In theory, key-value stores are oblivious to the information contained in the structure and content of the stored values, considering them opaque. Pure key-value stores simply concern themselves with efficient storage and retrieval of arbitrary data on behalf of applications, unencumbered by its nature and application-level usage.

Opacity and access to subelements inside structured data

This opacity has a downside. When extracting an element of data from within a stored value, often the whole value must be returned to the client which then filters out the unwanted (parent or sibling) data elements. Compared to document stores where such operations happen on the server, this is typically somewhat less efficient.

In practice data model such distinctions aren't always so clear cut. Some of the popular key-value stores (like Riak for instance) also offer visibility into certain types of structured stored data like XML and JSON. Therefore at a product level there is some overlap between the document and key-value stores.

While the key-value model is unarguably simple, like the document model it can suffer from paucity of data insight from an application programmer's point of view. To reify sets of useful information out of individual records, typically external processing infrastructure like map-reduce frameworks tend to be used, implying high latency in comparison with queries in the data store.

Though it isn't rich in terms data model, the key-value model offers certain operational and scale advantages. Since contemporary key-value stores generally trace their lineage to Amazon's Dynamo database — a platform designed for a non-stop shopping cart

service — they are optimized for high availability and scale, or as the Amazon team puts it, they should work even "if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. [9]". This impressive model might sway a decision to use such a store even if the data model isn't a perfect fit simply because operationally these stores tend to be very robust.

Column Stores

The heritage of the column family stores comes from Google's BigTable paper [11] where the authors described a novel kind of data store whose data model is based on a sparsely populated table where each row can contain arbitrary columns. At first this seems an unusual data model, but a positive side-effect of this model means that column stores provide natural data indexing based on the keys stored. Let's dig a little deeper.

In our discussion we'll use terminology from Apache Cassandra. Cassandra isn't necessarily a faithful interpretation of BigTable, but it is widely deployed and its terminology is widely understood.

In Figure 2-4 we see the four common building blocks used in column stores. The simplest unit of storage is the *column* itself, consisting of a name-value pair. Any number of columns can be combined into a *super column* which gives a name to a sorted set of columns. Columns are stored in rows, and when a row contains columns only it is known as a *column family*. Similarly when a row contains super columns it is known as a *super column family*.

11. *http://research.google.com/archive/bigtable.html*

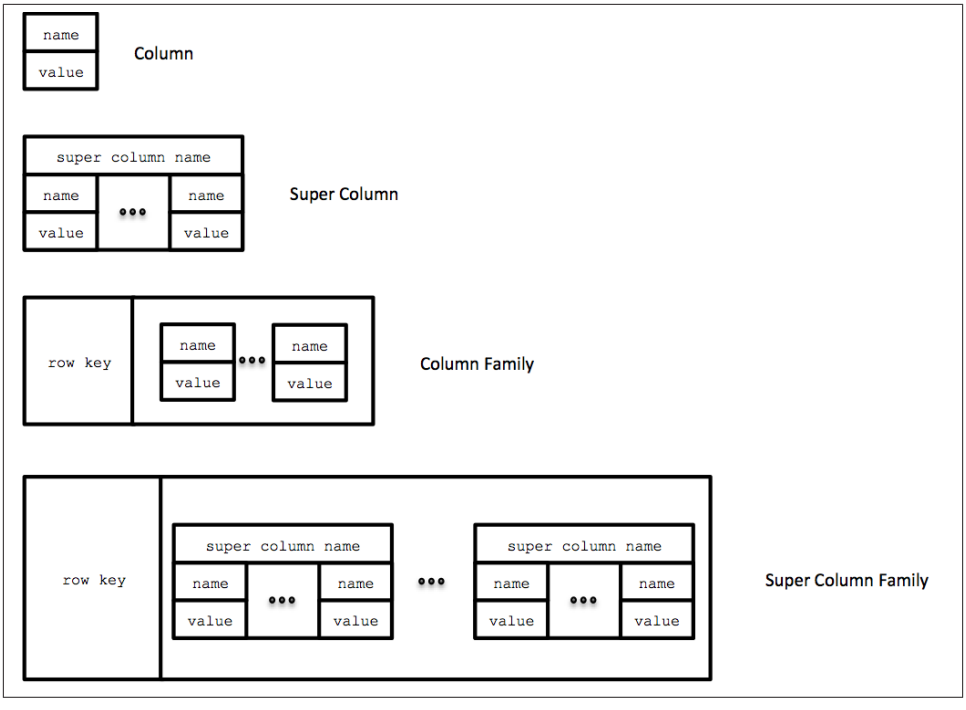


Figure 2-4. The four building blocks of column storage

It might seem odd to focus on rows to such an extent in a data model which is ostensibly columnar, but at in individual level rows really are important, providing the nested hashmap structure into which we decompose our data. In Figure 2-5 we've shown an example of how we might map a recording artist and their albums into a super column family structure — it really is logically nothing more than maps of maps which is a simple metaphor even if this column nomenclature seems unfamiliar.

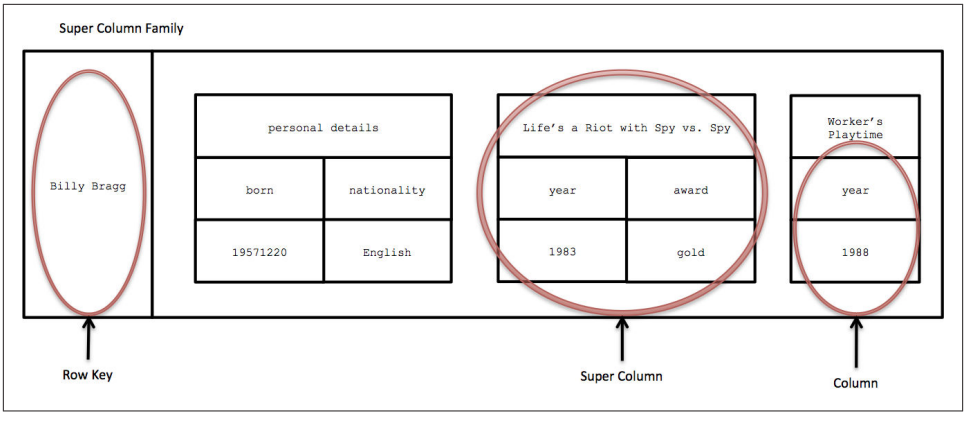


Figure 2-5. Storing line of business data in a super column family

In a column database, each row in the table represents a particular overarching entity (e.g. everything about an artist). These column families are containers for related pieces of data and provide demarcation (e.g. the artist's name and discography). It is within the column families we find actual key-value data—the state that the database exists to store—like the album's release dates and the artist's date of birth.

But helpfully, this row oriented view can be turned through 90 degrees to arrive at a column-oriented view. Where each row gives a complete view of one small part of the data set, the column view provides a natural index of a particular aspect across the whole data set. For example as we can see in Figure 2-6, by "lining up" keys we should be able to find all the rows where the artist is English, and from there it's easy to extract complete artist data from each row. It's not connected data as we'd find in a graph, but it does at least provide some insight.

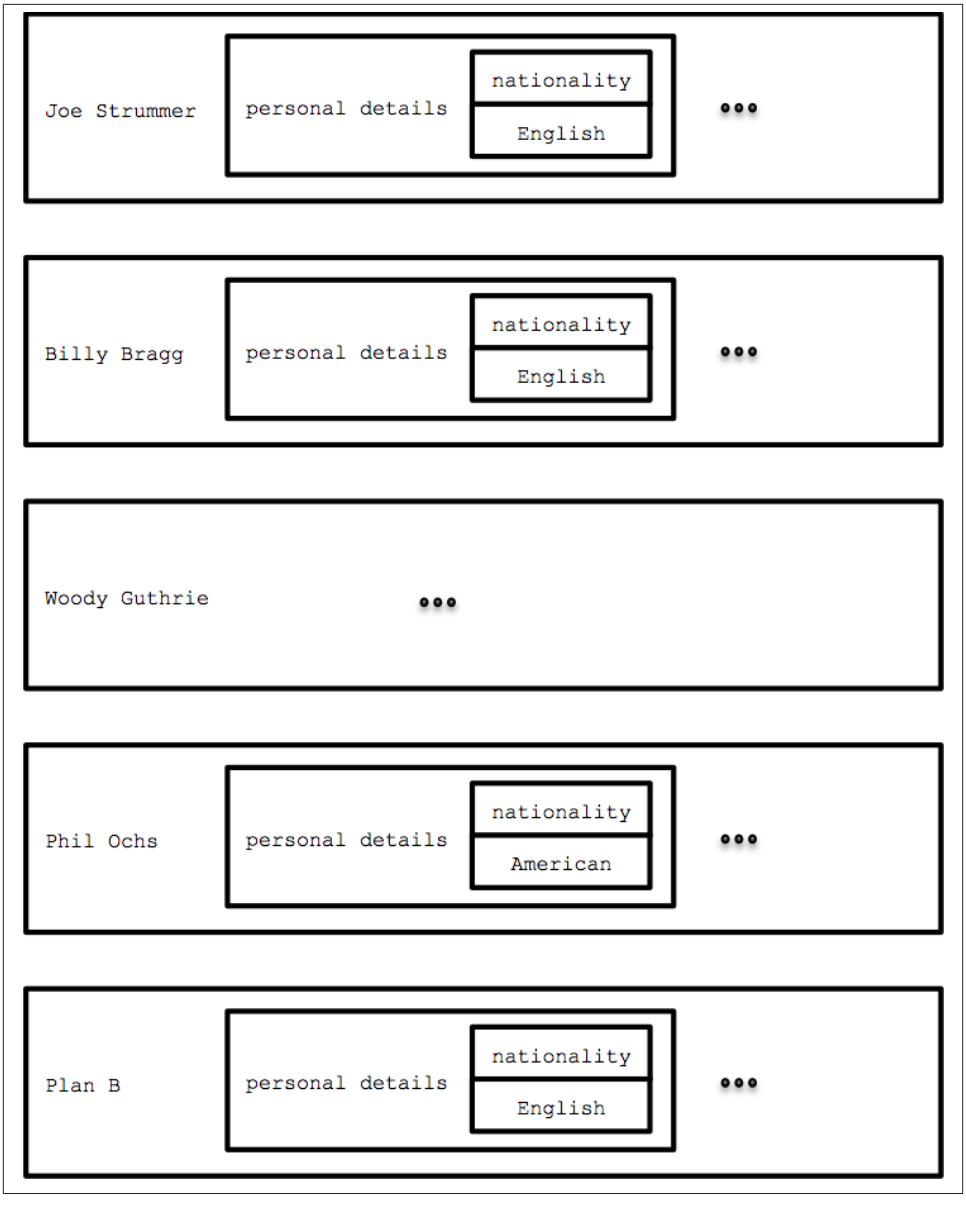


Figure 2-6. Keys form a natural index through rows in a column store

But it's not just the more expressive data model (compared to document and key-value stores at least) which are exciting about column store, but their operational characteristics too. For example, although Apache Cassandra is famed for being based on an Amazon Dynamo-like infrastructure for distribution, scale, and failover, under the

covers there are several storage engines (including the default) which have shown to be very competent at dealing with high write loads — the kind of peak write load problem when popular TV shows encourage interaction for example.

All in all, the column stores present themselves as reasonably expressive and operationally very competent. And yet at the end of the day, they're still simply *aggregate stores* just like document and key-value databases and as such querying is still going to be substantially augmented by processing to reify useful information at scale.

Query Versus Processing in Aggregate Stores

In focussing on the data models for (non-graph) NOSQL stores, we've highlighted each model and some of the similarities and differences between them. But on balance, the differences have been far fewer than the similarities. In fact these stores are sufficiently similar to be abstracted into one larger super-class of store which Fowler and Sadalage [12] have chosen to call *aggregate stores*, stemming from the observation that each of these stores persists standalone complex records which reflect the notion of an *Aggregate* in Domain-Driven design [13].

The details the underlying storage strategy differs a great deal between each aggregate store, yet they all have a great deal in common when it comes to their query model. While aggregate stores might provide features like indexing, simple document linking, or query languages for simple ad-hoc queries, it's commonplace to identify and extract (a subset of) data from the store before piping it through some processing external infrastructure. Typically that infrastructure is a map-reduce framework and it is used in order to reify deep insight into data which cannot be inferred by considering each aggregate individually.

Map-reduce, like BigTable, is another technique published by Google [14] and rapidly implemented in the open source community with Apache Hadoop and its ecosystem being the frontrunner map-reduce framework for the rest of us.

Map-reduce is a relatively simple parallel programming model where data is split and operated upon in parallel before being gathered back together and aggregated for provide focussed information. For example, if we wanted to count how many Americal artists are in a database of recording artists we would extract al of the artist records and discard all those representing non-American artists in the map phase, before counting the remaining records in the reduce phase to obtain our answer.

12. *http://martinfowler.com/bliki/NosqlDistilled.html*
13. *http://domaindrivendesign.org/*
14. *http://research.google.com/archive/mapreduce.html*

Of course doing this kind of operation on a large database — even where we have many machines to execute map and reduce code, and fast network infrastructure to move data around — is an enormous endeavour. Instead the usual approach is to try to use the features of the data store to provide a more focussed data set (e.g. using indexes or other ad-hoc queries) asnd then map-reduce that smaller data set to arrive at our answer.

Sometimes processing data in this way can prove to be beneficial, after all map-reduce is highly parallelizable and frameworks like Hadoop are a commodity. However though enormous processing throughput can be achieved this comes at the expense of both latency (extracting and re-assimilating the data) and risk since data is ripped out of its safe home in the database and crunched through some external machinery that has no sense of the underlying consistency model the database enforces.

Moving Onwards and Upwards

We've seen the three popular aggregate store types—key-value, document, and column store—in this chapter and understood their strengths and weaknesses. We've discussed how the aggregate data model often results in data stores that have excellent operational characteristics, but we've also seen how the query and processing models supported by those stores highlight the paucity of the underlying standalone aggregate data model presented to users, and the (latent) machinery that needs to be deployed to redress that shortcoming.

Aggregate stores are not natively suited to most domains where data is interconnected (the canonical case, if you give it a moment's consideration). You can use them that way, but then it's *your* job as the developer to fill in where the underlying data model leaves off. While the data might be big, it's not necessarily smart.

However it's now time for us to move on from the world of simple aggregates and into a world where data is habitually interconnected as we delve into the domain of graph databases.

CHAPTER 3

Graphs and Connected Data

In this chapter we've going to immerse ourselves in the world of graphs and connected data and make the case for graph databases as the sanest means for graph storage and querying. We're going to define *connected data* and show the benefits of working with graphs to solve complex data and business problems, and demonstrate how graphs are a natural model for a wide variety of domains. But before we dive into that, let's revisit why contemporary NOSQL stores and relational databases can struggle when used for managing graphs and connected data.

The Aggregate Model: Lacking Relationships

As we saw in the previous chapter, aggregate databases — key-value, document, and column stores — are popular members of the NOSQL family for storing sets of disconnected documents/values/columns. However by convention, it's possible to fake relationships (or at least pointers) in these stores by embedding the identifier of one aggregate inside well-known field in another. As we can see in as we see in Figure 3-1, as humans it's deceptively easy for us to infer relationships between our aggregates by visually inspecting them.

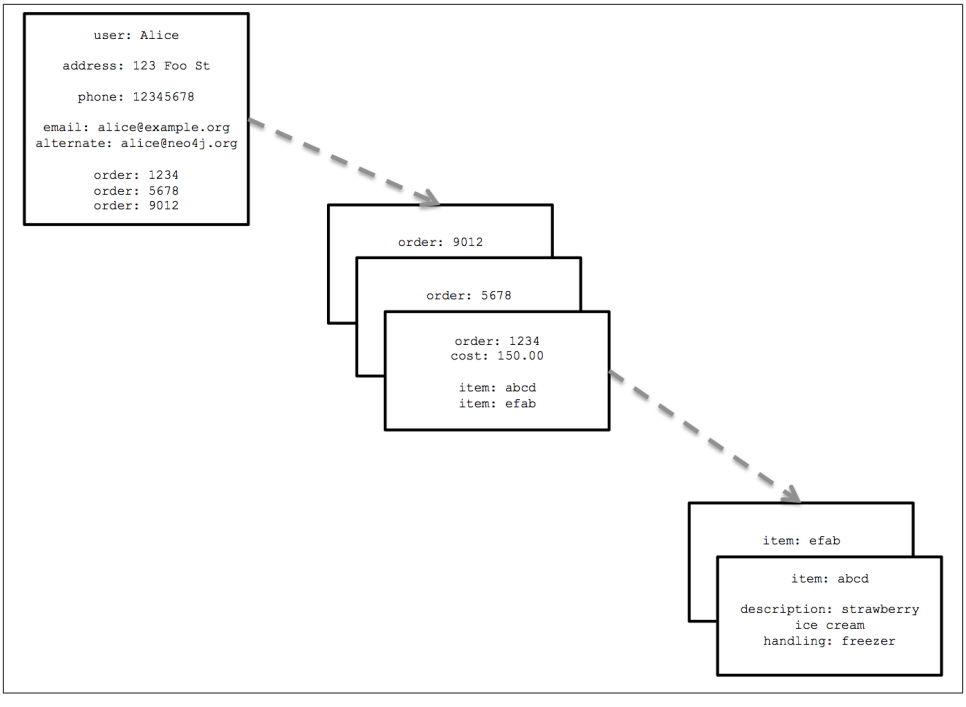


Figure 3-1. Reifying relationships in an aggregate store

In Figure 3-1 we can (somewhat reasonably) infer that some property values are really references to foreign aggregates elsewhere in the database. But that inference doesn't come for free, since support for relationships does not exist at all in this data model. Instead it's left to developers do the hard work to infer and reify useful knowledge out of these flat, disconnected data structures [1]. It's also left to developers to remember to update or delete those foreign aggregate references in tandem with the rest of the data, otherwise the store will quickly contain overwhelming numbers of dangling references, both harmful to data quality and query performance.

Links and walking

The Riak key-value store allows each of its stored values to be augmented with link metadata. Each link is one-way, pointing from one stored value to another. Riak allows any number of these links to be *walked* (in Riak terminology), making the model helpfully somewhat connected. However as a native key-value store (rather than graph da-

1. Remember in most aggregate stores, it's inside the aggregates themselves are where structure is given to data, often as nested maps.

tabase) at query time link walking is powered by map-reduce and is relatively latent, suited for simple graph-structured programming rather than general graph algorithms.

But there's another weak point in this scheme, because in the absence of more identifiers to "point" back (the foreign aggregate "links" are not reflexive of course) we also lose the ability to be able to run other interesting queries on the database. For example with the current structure it is expensive to ask of the database who has bought a particular product in order to perhaps recommend it based on customer profile. In those cases we can export the data set and process it via external compute infrastructure (likely Hadooop) in order to brute-force compute the result. Another option is to retrospectively insert the backwards pointing foreign aggregate references before then querying for the result. Either way the results will be latent.

Sometimes it might be tempting to think that there is equivalence between aggregate stores and graph databases. This is a forlorn hope since aggregate stores do not maintain consistency of connected data nor provide index-free adjacency [2], and therefore are slow when used for graph problems. This results in an inherently latent compute mode where data is crunched outside the database rather than queried within it.

It's easy to see how these aggregate model limitations impact us even in a simple social domain [3]. In Figure 3-2 we see a (small) social network — a trivial example of a graph tacitly encoded inside documents.

2. *http://en.wikipedia.org/wiki/Graph_database*

3. Social graph is the poster child for graphs in general, but as we'll see there are other interesting and valuable use cases we can address

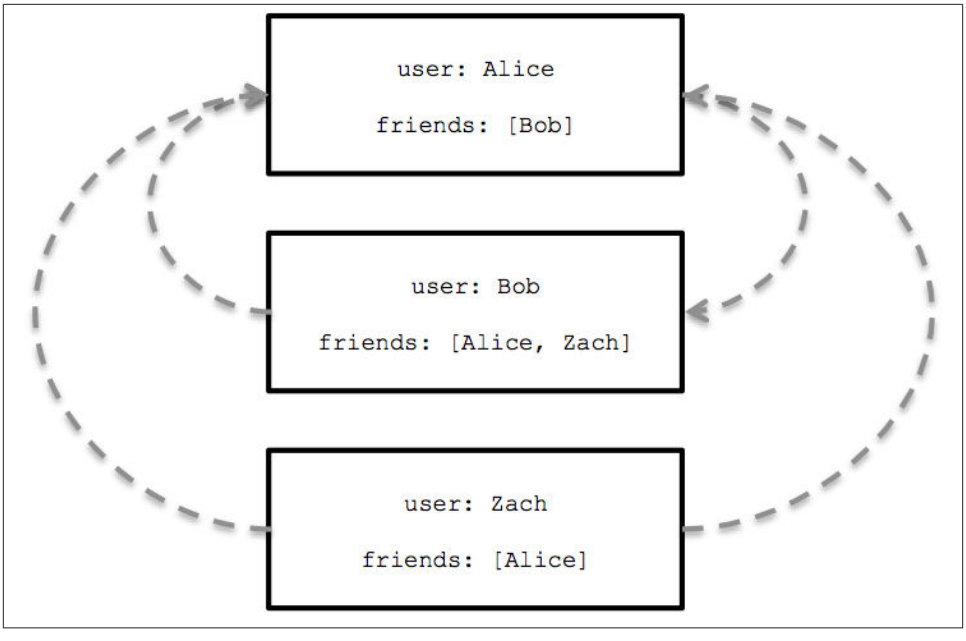


Figure 3-2. A small social network encoded in an aggregate store

In Figure 3-2 to find immediate friends of a user is easy in theory — assuming data access has is disciplined so that the identifiers stored in the `friends` property are consistent with other record IDs in the database. In this case we simply look up immediate friends by their ID, which requires numerous index lookups (one for each friend) but no brute-force scans of the entire data set. In doing this, for example, we'd find that `Bob` considers `Alice` and `Zach` to be friends.

But friendship isn't always reflexive, so what if we'd like to ask, for example, "who is friends with Bob?" rather than "who are Bob's friends?" That's a tougher proposition and in this case our only option would be to brute force across the whole data set looking for `friends` entries that contain `Bob`.

Brute-force processing

Brute force computing an entire data set is *O(n)* in terms of complexity since all *n* aggregates in the data store must be considered. That's far too costly for most reasonable sized data sets where we'd prefer *O(log n)* or better where possible [4].

4. Remember a *O(log n)* algorithm is somewhat efficient because it discards half the potential workload on each iteration.

> Conversely a graph database provides constant order lookup in this case, we'd simply find the node in the graph that represents Bob, and then follow any incoming *friend* relationships to other nodes which represent people who consider Bob to be their friend. This is far cheaper than brute-forcing (unless **everybody** is friends with Bob) since far fewer members of the network are considered — only those that are connected to Bob.

To prevent the need for constant processing of the entire data set, we'd have to further denormalize the storage model by adding backward links. In this case we'd have to add a second property, perhaps called `friended_by` to list the perceived incoming friendship relations. This doesn't come for free. For instance we have to pay the initial and ongoing cost of increased write latency, and increased disk space utilization for storing that additional metadata. *Traversing* those links is expensive because each hop requires an index lookup, as aggregates have no notion of locality, unlike graphs which naturally provide index-free adjacency through real — not reified — relationships. That is, by implementing a graph structure atop a non-native store, we may get some of the benefits in terms of partial connectedness but at substantial cost.

That substantial cost is amplified when you consider traversing deeper than just one hop. Friends are easy enough, but imagine trying to compute — in real time — friends of friends, or friends of friends of friends. That's impractical with this kind of database since traversing a fake relationship is not cheap enough. This doesn't only limit your chances of expanding your social network, but in other use cases will reduce profitable recommendations, will miss faulty equipment in your data center, will let fraudulent purchasing activity slip through the net. Most systems try to maintain the appearance of such graph-like processing, but inevitably it's done in batches and doesn't provide that high-value real-time interaction that users demand.

The Relational Model: Also Lacking Relationships

Things aren't all that rosy in the relational database world either. For several decades, developers have tried to accommodate increasingly connected, semi-structured datasets inside relational databases. But whereas relational databases were initially designed to codify paper forms and tabular structures—something they do exceedingly well—they struggle when attempting to model the ad hoc, exceptional relationships that crop up in the real world. It's an ironic twist that relational databases are in fact so poor at dealing with relationships. While relationships do at least exist in the vernacular of relational databases, they exist only as a means of joining tables and are completely free of semantics (like direction, name). Worse still, as outlier data multiplies, and the amount of semi-structured information increases, the relational model becomes burdened with large join tables, sparsely populated rows and lots of null-checking logic. The rise in connectedness translates in the relational world into increased joins, which impede

performance and make it more difficult to evolve an existing database in response to changing business needs.

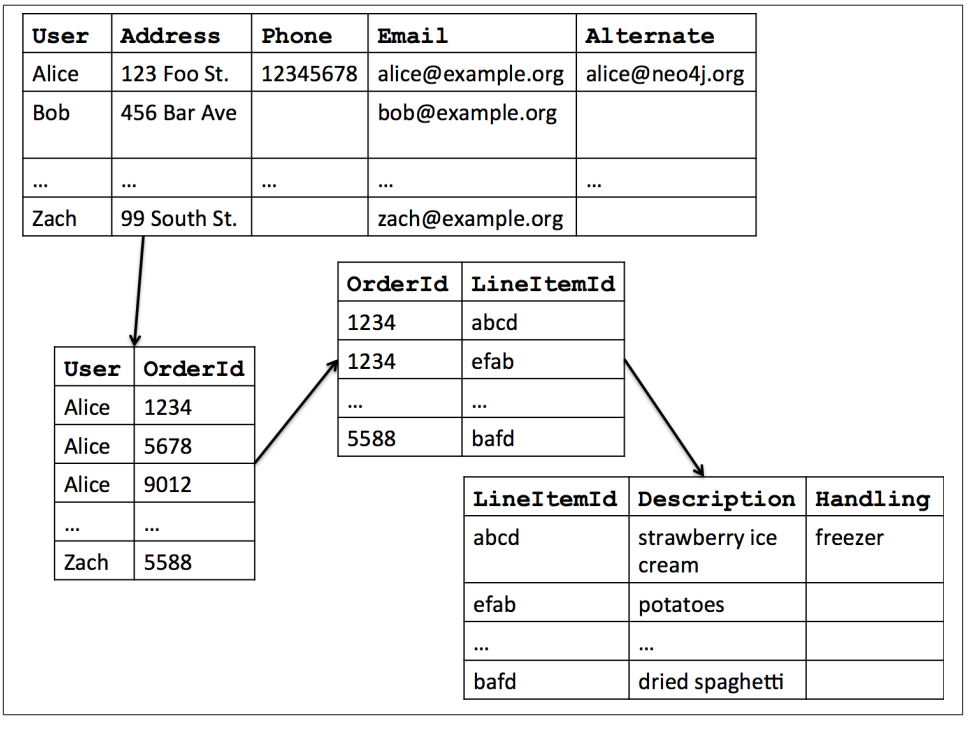


Figure 3-3. Semantic relationships are hidden in a relational database

Even in a simple relational schema like that shown in Figure 3-3, many of these problems are immediately apparent.

- Manifest accidental complexity and additional development and maintenance overheads because of join tables and maintaining foreign key constraints *just to make the database work*.
- Sparse tables require special checking in code while schema gives false confidence about perceived data structure.
- Several *expensive* joins needed just to discover what a customer bought.
- No cheap way of asking reciprocal queries. For example it's acceptable from a time perspective to ask "what products did a customer buy?`` but not at all reasonable to ask "which customers bought this product?``, which is the basis of recommendation systems and numerous other valuable applications.

Even in now commonplace domains like social networks, relational databases can struggle. For modestly popular people, a relational database will be able to report a set of friends relatively easily from a simple join-table arrangement like that shown in Figure 3-4.

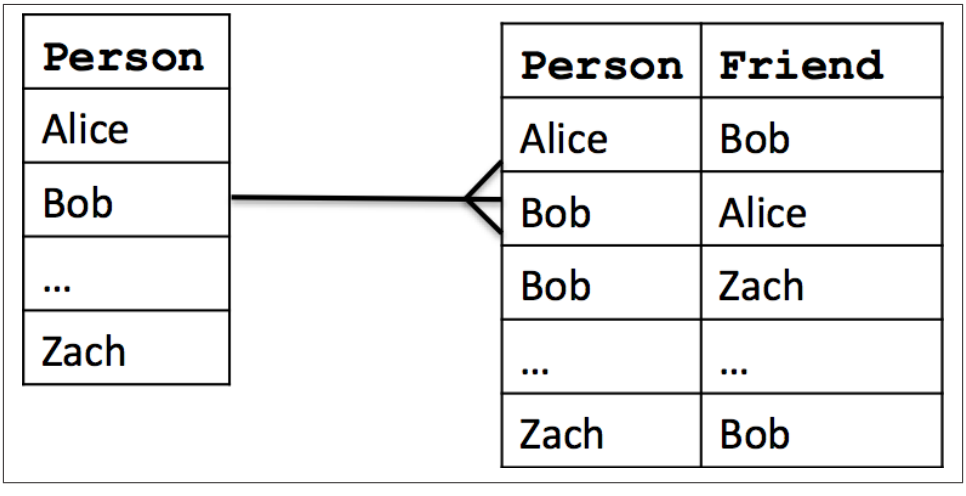


Figure 3-4. Modeling friends and friends-of-friends in a relational database

Firstly, let's satisfy ourselves that easy query of asking "who are Bob's friends" is indeed an easy case:

Example 3-1. Bob's friends

```
SELECT PersonFriend.friend
FROM Person JOIN PersonFriend
  ON Person.person = PersonFriend.person
WHERE Person.person = 'Bob'
```

Which results in `Alice` and `Zach` — great! This isn't such an onerous query for the database either, since it constrains the number of rows under consideration by filtering on with `Person.person='Bob'`.

Now we can ask the reciprocal query which is who is friends with Bob, since you'll recall friendship isn't always a reflexive relationship:

Example 3-2. Who is friends with Bob?

```
SELECT Person.person
FROM Person JOIN PersonFriend
  ON Person.person = PersonFriend.person
WHERE PersonFriend.friend = 'Bob'
```

This query results only in the row for `Alice` being returned, implying that `Zach` doesn't consider `Bob` to be a friend, sadly. The reciprocal query is still easy to implement, but for the database to work out who is friends with `Bob` it requires more effort since all of the rows in the `PersonFriend` table must be considered.

Things degenerate further when we ask "who are my friends of friends". Firstly things get a little tougher simply because of query complexity, since hierarchies are not a particularly natural or pleasant thing to express in SQL, relying on recursive joins [5].

Example 3-3. Alice's friends-of-friends

```
SELECT pf1.person as PERSON, pf3.person as FRIEND_OF_FRIEND
FROM PersonFriend pf1 INNER JOIN Person
  ON pf1.person = Person.person
INNER JOIN PersonFriend pf2
  ON pf1.friend = pf2.person
INNER JOIN PersonFriend pf3
  ON pf2.friend = pf3.person
WHERE pf1.person = 'Alice' AND pf3.person <> 'Alice'
```

This query is computationally complex, not to mention it only deals with Alice's friends of friends and goes no further out into Alice's social network. While "who are my friends-of-friends-of-friends" is feasible, beyond that to 4, 5, or 6 degrees of friends is not suited to relational computation due to computational and space complexity of recursively joining tables, either through clever SQL or by having application code explicitly enumerate the joins.

In a relational database, we work against the grain. We have often brittle schemas which rigidly describe the structure of the data only to have those schemas' intended behavior overridden by code that ostensibly obeys the laws but bends all the rules. Consequently a web of sparsely populated tables propagates through the database and a plethora of code to handle each of the exceptional cases is spread through the application. This increases coupling and all but destroys any semblance of cohesion.

And for most of us, relational technology simply doesn't evolve or scale easily enough. It's time to move on.

Connected Data and Graph Databases

In the previous example with aggregate and relational stores we were dealing with *implicitly* connected data. That is, as users we could infer semantic dependencies between entities but the data model — and as a consequence the databases themselves — could not recognize or assist us in taking advantage of those connections. As a result

5. Some relational databases provide syntactic sugar for this, for instance Oracle supports `CONNECT BY` which simplifies the query, but not the underlying computational complexity.

the workload of reifying a network out of the flat, disconnected data fell to us and we had to deal with the consequences of slow queries, latent writes across denormalized stores, etc.

In reality what we want from our data is a cohesive picture of the data set to suit the current operation, including the connections between the data. In the graph world, connected data is represented in high-fidelity. Where there are connections in the domain, there are connections in the data. For example, consider the (expanded [6]) social network in Figure 3-5.

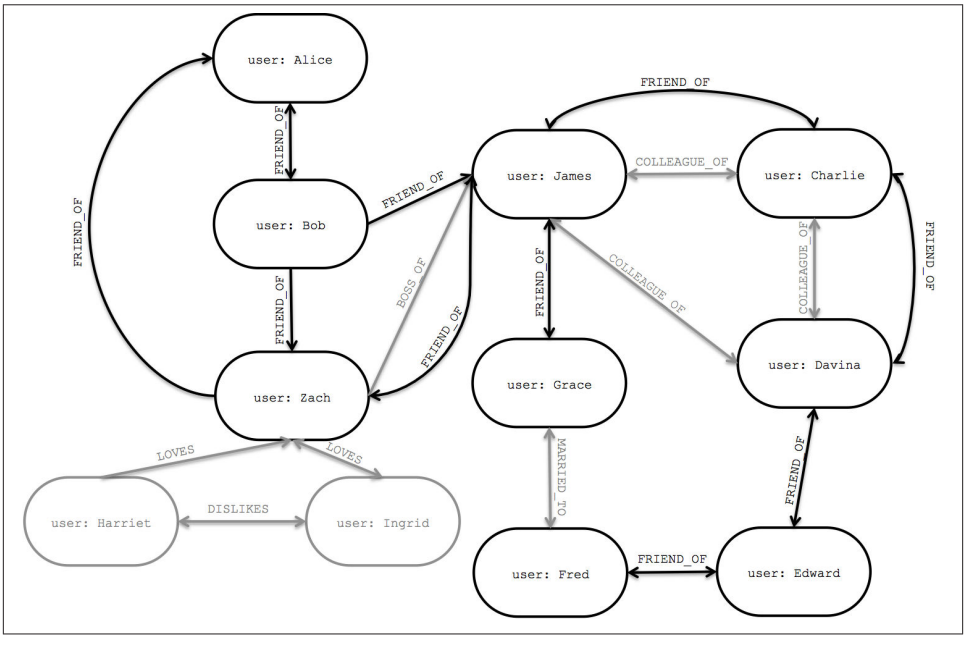


Figure 3-5. Easily modeling friends, colleagues, workers, and (unrequited) lovers in a graph

Our social network, like so many real-world cases of connected data, the connections between entities don't exhibit uniformity across the domain — the domain is then said to be semi-structured. Social networks are a popular example of a densely-connected, semi-structured network, one that resists being captured by a one-size-fits-all schema or conveniently split across disconnected aggregates. That our simple network of friends has both grown (there are now potential friends up to six degrees away), and has been enriched to describe other semantics between participants is testament to the scalability

6. We can expand the network in this example because the graph data model is more expressive and performant than the other models we've considered

of graph modeling. New *nodes* and new *relationships* have been added without compromising the existing social network nor prompting any migrations to occur — the original data and its intent remain intact.

But now we have a much richer picture of the network. We can see who `LOVES` someone (and whether that love is returned), who works with whom via `COLLEAGUE_OF`, who to suck up to through `BOSS_OF`, who's off the market because they're `MARRIED_TO` someone else even introduce an antisocial element into our otherwise social network with `DISLIKES` [7]. With this dataset like this at our disposal, we can now look at the performance advantages of graph databases for connected data.

In a graph database path operations—traversing the graph-- are not only highly aligned with the data model (relationships naturally form paths) but also highly efficient. For example in their book "Neo4j in Action" [8] Partner and Vukotic performed an experiment using a relational store and Neo4j. In this experiment, the authors wished to validate (or repute) the notion that Neo4j is substantially quicker for connected data than a relational store. The domain they chose was to find friends-of-friends at a maximum depth of 4 in a graph — that is for any person in a social network, is there a route to some other person in the graph at most 4 hops away. For a social network containing 1,000,000 people each with approximately 50 friends, the results heavily suggest that graph databases are the best choice for connected data as we can see in Table 3-1.

Table 3-1. Finding extended friends in a relational database versus efficient finding in Neo4j

Depth	RDBMS Execution time (s)	Neo4j Execution time (s)	Records returned
2	0.016	0.01	~2500
3	30.267	0.168	~110,000
4	1543.505	1.359	~600,000
5	Unfinished	2.132	~800,000

At depth 2 (friends-of-friends) both databases perform well enough to insert the query into a user-facing system. The difference of milliseconds would not be noticeable, though even at this stage it's illustrative to see that the Neo4j query runs in less than 2/3 the time that the relational query. But by the time we reach depth 3 (friend-of-friend-of-friend) it's clear that the relational database can no longer deal with the query in a reasonable timeframe — taking over 30 seconds to complete the query would be completely unacceptable for an online system. Conversely Neo4j's response time remains

7. You can infer that Ingrid and Zach both `LOVES` one-another and that Harriet unrequitedly `LOVES` Zach, that Ingrid and Harriet are probably within their rights to `DISLIKE` each other.

8. *http://www.manning.com/partner/*

relatively flat, just a fraction of a second to perform the query and definitely quickly enough for an online system.

At depth 4 the relational database exhibits terrible latency for an online system, to the extent where it's practically useless. Neo4j timing's are a marginally less good too, but is still at the periphery of being acceptable for a responsive online system. Finally at depth 5 the relational database simply takes too long to complete the query while Neo4j remains relatively modest at around 2 seconds, and yet this timing can be highly trimmed should we (sensibly) choose to return only a handful of the many thousands of possible distant friends .

Aggregate and relational stores both perform poorly for connected data

Both aggregate stores and relational databases perform poorly we move away from modestly sized set operations — operations that both kinds of databases *should* be good at. When we try to mine path information out of the graph like the classic finding of friends and friends-of-friends that we're all used to from social media platforms and applications things demonstrably slow down. We don't mean to unduly beat up on either aggregate stores or relational databases, they have a fine technology pedigree for the things they're good at but for managing connected data they fall short as we've discussed. Anything more than a shallow traversal of immediate friends or possibly friend-of-friend will be slow because of the number of index lookups involved. Graphs on the other hand offer index-free adjacency so that traversing connected data is extremely rapid.

Friends and social networks are all well and good, but at this point you might reasonably wonder whether finding such remote friends is a valid use-case. After all, it's rare that we'd be so desperate for company that we'd traverse so far out of the core of our social network. The point is that we can substitute social networks for any other domain and experience similar performance, modeling and maintenance benefits. Whether those domains are music or data center management, bio-informatics or football statistics, network sensors or time-series of trades, graphs are a powerful source of insight. So let's avail ourselves of the power of graphs for connected data with a contemporary example: social recommendation of products not only based on a user's purchase history but on the histories of their friends, neighbours and other people like them, bringing together

several independent facets of the user's lifestyle to make more accurate and more profitable recommendations.

Productive analysis

One of the most appealing aspects of the graph data model is its intuitiveness. People without any background in data modeling — which is to say, relational modeling — easily grasp the graph approach. The typical "whiteboard" or "back of the napkin" representation of a data problem is a graph. More often than not such representations can be directly translated into a graph data model. One of the most important contributions of graphs to informatics is how they make data modeling and data interrelationships available to the layperson.

Let's start off by modeling the classic purchase history of a user as connected data. In a graph, this can be as simple as linking between the user and their orders, and inserting links between orders to provide a purchase history as we see in Figure 3-6.

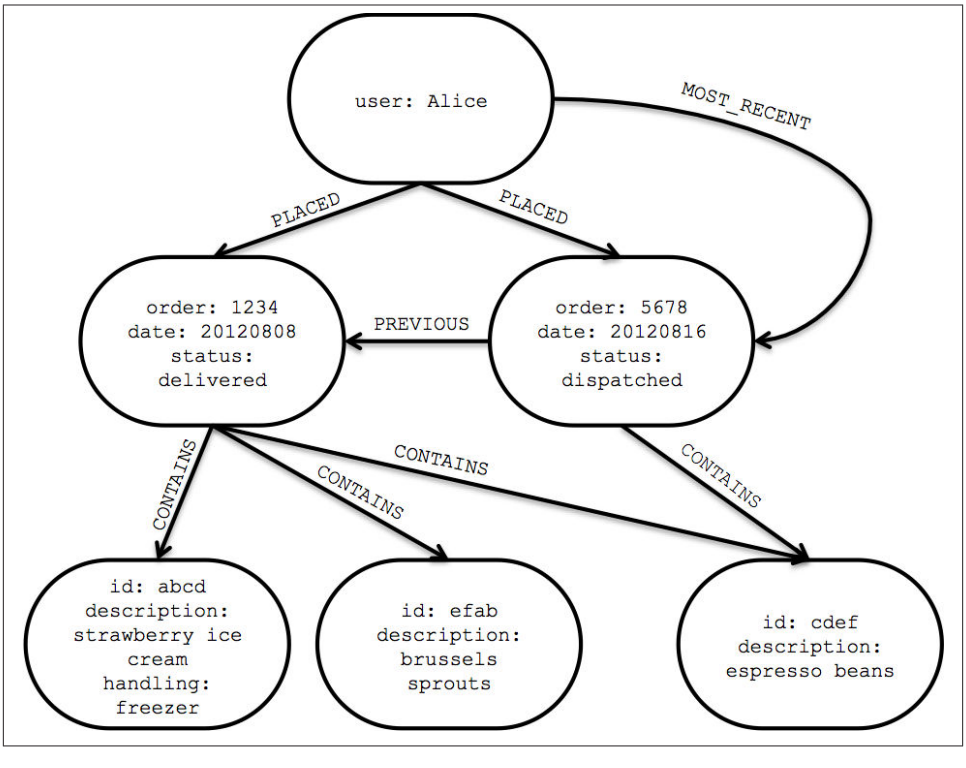


Figure 3-6. Modeling a user's order history in a graph

The kind of graph shown in Figure 3-6 provides a great deal of insight into customer behavior. We can see all the orders that the user PLACED and we can easily reason about what each order CONTAINS. So far so good, but we can take the opportunity to enrich this graph to support well-known access patterns. For example, users often want to see their order history, so we've added a linked list structure into the graph such that the user's most recent order is found by following the outgoing MOST_RECENT relationship, and we can iterate through the lsat going further back in time by following the PREVIOUS relationships [9].

Having even this simple data allows us to start to make recommendations. If we notice that users who buy strawberry ice cream, also buy espresso beans then we can start to recommend those beans to users who normally only buy the ice cream. But this is a rather one-dimensional recommendation even if we traverse lots of orders to ensure there's a strong correlation between strawberry ice cream and espresso beans. We can do **much** better, and the way we do that is to intersect the purchasing graph with graphs from other domains. Since graphs are naturally multi-dimensional structures, it's quite straightforward to ask a much more sophisticated question of the data to gain access to a fine-tuned market segment. For example to ask "all of the flavors of ice cream that people living near a user like who also enjoy espresso but dislike Brussels sprouts."

From our point of view, enjoying and disliking items could be easily considered isomorphic to having ordered those items repeatedly or infrequently (perhaps once only). How would we define "living near?" Well it turns out that, perhaps unsurprisingly, geospatial coordinates are easily graphed. The most likely implementation of this is to create an R-Tree [10] to represent bounded boxes around geographies. Using such a structure it's natural to describe overlapping hierarchies of locations. For example, we can express that London is in the UK, but we can also express that the postal code SW11 1BD is in Battersea which is a district in London (which is in south-eastern England, which in turn is in Great Britain [11]). And since UK postal codes are fine-grained, we can use that boundary to target people with somewhat similar tastes.

9. There's no need for a reciprocal NEXT relationship unless we'd we want to. Traversing the PREVIOUS relationship from head to tail provides the same affordance.

10. *http://en.wikipedia.org/wiki/R-tree*

11. The various terms associated with the geography and nationhood in the British Isles is intricate, but this will help the unitiated: *http://www.visualnews.com/2011/02/03/the-difference-between-great-britain-england-the-united-kingdom-and-a-whole-lot-more/*

Such pattern matching queries are extremely difficult to write in SQL, and laborious to write against aggregate stores, and tend to perform very poorly. Graph databases on the other hand are optimized for precisely these types of traversal and pattern matching queries which be written very easily using a query language (such as Cypher [12]in Neo4j), and can be expected to perform very well (milliseconds response time).

This graph isn't only useful for recommendations on the buyer's side, it's also useful operationally for the seller. For example, given certain buying patterns (products, cost of typical order, etc) we can establish whether a particular transaction might constitute fraud. Patterns outside of the norm for a given user can easily be detected in a graph and be flagged for further attention (using well-known similarity measures from the graph data-mining literature), thus reducing the risk for the seller.

From the data practitioner's point of view, it's clear that the graph database's sweet spot lies in handling data sets that are so sophisticated that they are unwieldy when treated in any other form than as a graph. We therefore define the term *connected data* to describe *data that is best handled as a graph*.

Schema-Free Development and Pain-Free Migrations

Notwithstanding the fact that just about anything can be modeled as a graph, we live in a pragmatic world, and novel, powerful data modeling techniques do not in themselves provide sufficient justification for replacing standard, well-known platforms. There must also be an immediate and very significant practical benefit. In the case of graph databases as we've seen, this motivation exists in the form of a set of use cases and data patterns whose performance improves by one or more orders of magnitude when implemented in a graph, and whose latency is much lower compared to batch processing aggregates. However when we factor in the increase in performance with the benefits of agility, schemaless flexibility, and ease of development, the case for graph databases can readily be made.

Developers today want to connect data as the domain dictates, thereby allowing structure and schema to emerge rather than having to pre-define it upfront. Moreover, they want to be able to evolve their model in step with the rest of their application, using a technology aligned with the incremental and iterative software delivery practices that have come to the fore in recent years. The graph database data model provides the flexibility necessary to help IT *move at the speed of business*, while today's graph database

12. *http://docs.neo4j.org/chunked/milestone/cypher-query-lang.html*

technologies provide the APIs and query languages necessary for widespread adoption by mainstream programmers.

This has generally positive implications for developer productivity and project risk. For instance, we don't need to model exhaustively ahead of time — a practice which is all but foolhardy in the face of changing business requirements since the graph model is naturally additive. As new facts come to light we **design** the graph structures for them and pour in the new data in with the existing. Such an additive mindset tends to mean that we don't migrate graphs often either, we migrate much less often, in fact only when we're performing breaking changes on existing structures.

There are implications, of course, for development and maintenance of applications since graph users cannot rely on the (rotten) crutch of a relational schema to try to fix some level of governance into the database—often much to the confusion of traditional data and application architects. Instead that governance is typically applied in the applications themselves through tests which assert the business rules (and by implication the way data is stored and queried) are valid over time. This isn't something to get excited about, that tests drive out the behavior of a system is uncontroversial in the 21st century and that tests providing ongoing governance (particularly around regressions) is also recognized as beneficial by the developer community at large.

Connecting the Dots

In this chapter we've highlighted the fact that data is most useful when it's connected. We've seen how other kinds of data stores leave connectedness as an exercise in data processing to the developer, and contrasted that with graph databases where connectedness is first-class citizen. Finally we've show how the expressive and flexible nature of graphs makes them ideal for high-fidelity domain analysis, frictionless development and graceful system maintenance

CHAPTER 4

Working with Graph Data

In previous chapters we've described the substantial benefits of the graph database when compared both with document, columnar and key-value NOSQL stores and with traditional relational databases. But having chosen to adopt a graph database, the question arises: how do we model the world in graph terms?

In this chapter we'll look at graph modeling. We'll focus in particular on the expressiveness of the graph model and the ease with which it can be queried. Because graphs are extremely flexible, lending themselves to many different domain representations, there's often no absolute right or wrong way to model with graphs, just some layouts that work better than others for a given problem. We'll highlight these good practices while also covering some classic pitfalls of graph modeling for the unwary.

Models and Goals

Before we dig deeper into modeling with graphs, a word on models in general. Modeling is an abstracting activity motivated by a particular need or goal. We model in order to bring specific facets of an unruly domain into a space where they can be structured and manipulated. There are no natural representations of the world the way it "really is," just many purposeful selections, abstractions and simplifications, some of which are more useful than others for a satisfying a particular goal.

Graph representations are no different in this respect. What perhaps differentiates them from many other data modeling techniques, however, is the close affinity between the logical and physical graph models. Relational data management techniques require us to deviate from our natural language representation of the domain: first by cajoling our representation into a logical model; and then by forcing it into a physical model. These transformations introduce semantic dissonance between our conceptualisation of the world and the database's instantiation of that model. With graph databases this gap shrinks considerably.

Graph modeling naturally fits with the way we tend to abstract the salient details from a domain using circles and boxes, and then describe the connections between these things by joining them with lines. Today's graph databases, more than any other database technologies, are "whiteboard friendly." The typical whiteboard view of a problem *is* a graph. What we sketch in our creative and analytical modes maps closely to data model we implement inside the database. In terms of expressivity, graph databases reduce the impedance mismatch between analysis and implementation that has plagued relational database implementations for many years. What is particularly interesting about such graph models is the fact that they not only communicate how we think things are related, they also clearly communicate the kinds of questions we want to ask of our domain. As we'll see throughout this chapter, graph models and graph queries are really just two sides of the same coin.

Relational Modeling in a Systems Management Domain

This book is primarily concerned with graphs, graph databases, graph modeling and graph queries. To introduce graph modeling, however, it's useful to compare how we work with graphs versus how we work in relational world, since RDBMS systems and their associated data modeling techniques are familiar to most developers and data professionals.

To help compare graph and relational modeling techniques, we'll examine a simple data-center management domain. In this domain, data centers support many applications on behalf of many customers using a whole plethora of infrastructure, from virtual machines to physical load balancers. A micro-level view of this domain is shown in Figure 4-1.

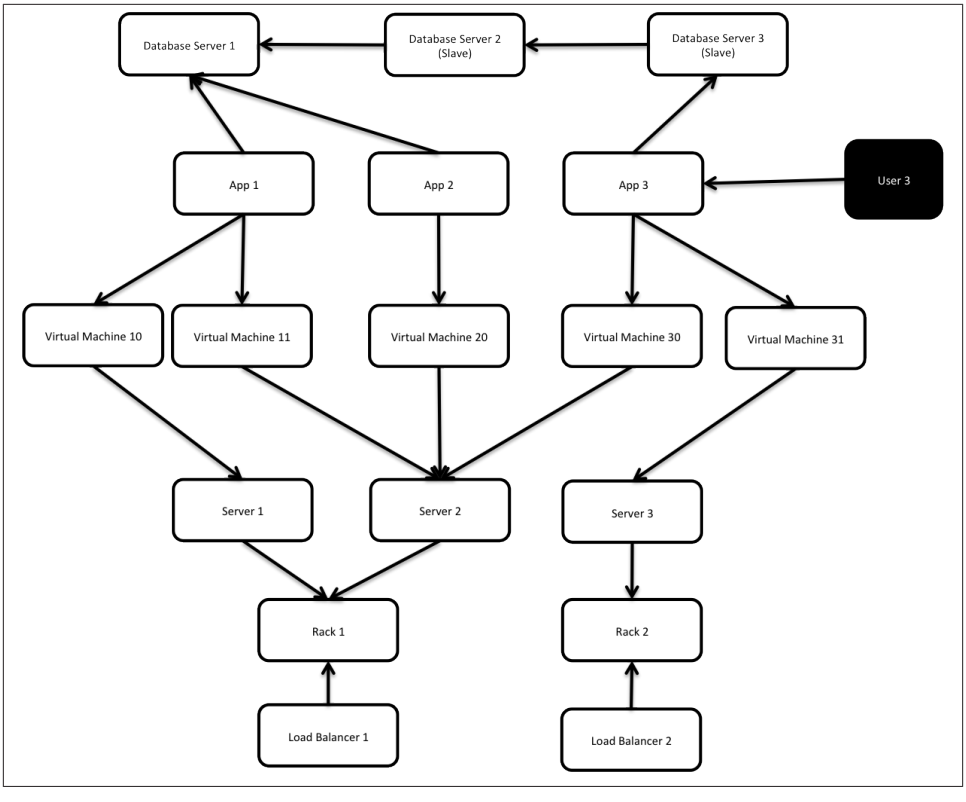


Figure 4-1. Simplified snapshot of application deployment within a data center

In Figure 4-1 we see a somewhat simplified view of several applications and the data center infrastructure necessary to support them. The applications, represented by nodes `App 1`, `App 2` and `App 3`, depend on a cluster of databases labelled `Database Server 1, 2, 3`. While users logically depend on the availability of an application and its data, there is additional physical infrastructure between the users and the application; this infrastructure includes virtual machines (`Virtual Machine 10, 11, 20, 30, 31`), real servers (`Server 1, 2, 3`), racks for the servers (`Rack 1, 2`), and load balancers (`Load Balancer 1, 2`), which front the apps. In between each of the components there are of course many networking elements: cables, switches, patch panels, NICs, power supplies, air conditioning and so on—all of which can fail at inconvenient times. To complete the picture we have a straw-man single user of application 3, represented by `User 3`.

As the operators of such a system, we have two primary concerns:

- Ongoing provision of functionality to meet (or exceed) a service-level agreement, including the ability to perform forward-looking analyses to determine single

points of failure, and retrospective analyses to rapidly determine the cause of any customer complaints regarding the availability of service;

- Billing for resources consumed, including the cost of hardware, virtualisation, network provision and even the costs of software development and operations (since these are a simply logical extension of the system we see here).

If we are building a data center management solution, we'll want to ensure that the underlying data model allows us to store and query data in a way that efficiently addresses these primary concerns. We'll also want to be able to update the underlying model as the application portfolio changes, as the physical layout of the data centre evolves or as virtual machine instances migrate. Given these needs and constraints, let's see how the relational and graph models compare.

Modeling for relational data

The initial stage of modeling in the relational world is similar to the first stage of many other data modeling techniques: that is, we seek to understand and agree on the entities in the domain, how they interrelate and the rules that govern their state transitions. Most of this tends to be done informally, often through whiteboard sketches and discussions between subject matter experts and system and data architects, until the participants reach some kind of agreement—expressed perhaps in the form of a diagram such as the one in Figure 4-1.

The next stage captures that agreement in a more rigorous form such as an entity-relationship diagram. This transformation of the model using a more strict notation provides us with a second chance to refine a common domain vocabulary that can be shared with relational database specialists [1]. In our example, we've captured the domain in the E-R diagram Figure 4-2.

Ironically E-R diagrams immediately demonstrate the shortcomings of the relational model for capturing a rich domain. Although E-R diagrams allow relationships to be named (something which relational stores disallow, and which graph databases fully embrace) they only allow *single, undirected* named relationships between entities. In this respect, the relational model is a poor fit for real-world domains where relationships between entities are both numerous and semantically rich and diverse.

1. Such approaches aren't always necessary: adept relational users often move directly to table design and normalization without first describing an intermediate E-R diagram.

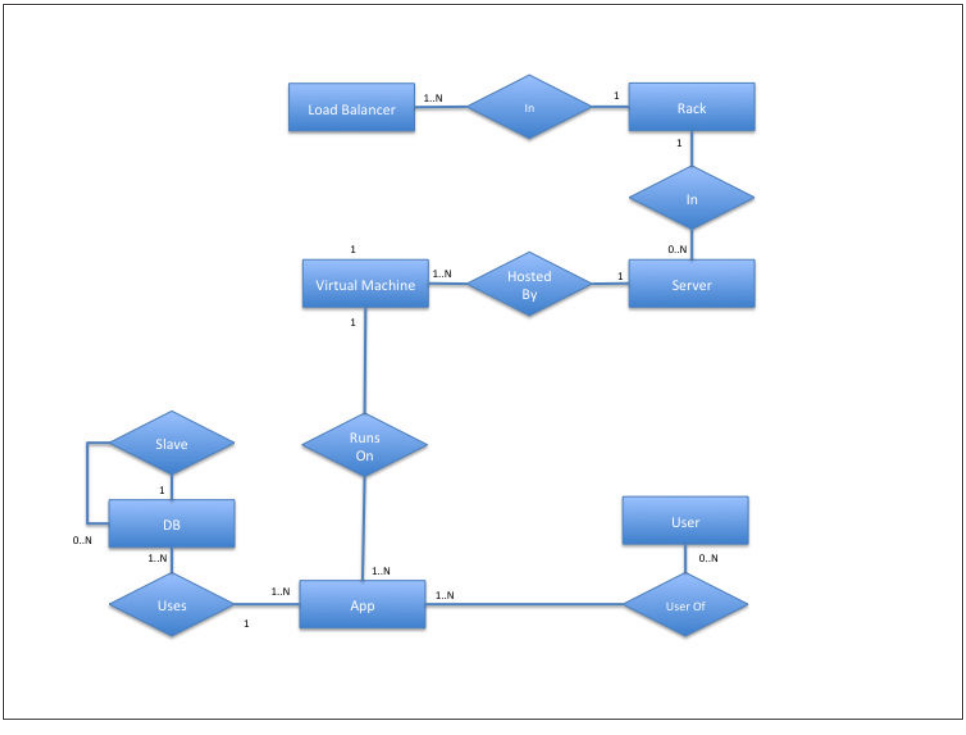


Figure 4-2. An Entity-Relationship diagram for the data center domain

Once we have arrived at a suitable logical model, we map it into tables and relations, which are then normalized to ensure high fidelity and zero data redundancy. In many cases this step can be as simple as transcribing the E-R diagram into a tabular form and then loading those tables via SQL commands into the database. But even the simplest case serves to highlight the idiosyncrasies of the relational model. For example, in Figure 4-3 we see that a great deal of accidental complexity creeps into the model in the form of foreign key constraints (everything annotated `[FK]`), which support one-to-many relationships, and join tables (e.g. `AppDatabase`), which support many-to-many relationships—and all this before we've added a single row of real user data. These constraints are model-level metadata that exist simply so that we can reify relations between tables at query time. Yet the presence of this structural data is keenly felt, for it clutters and obscures the domain data with data that serves the database, not the user.

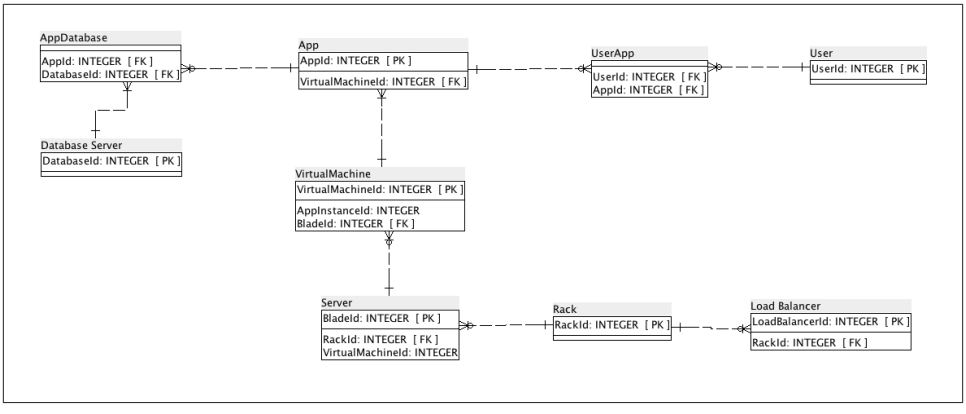


Figure 4-3. Tables and relationships for the data center domain

We have now arrived at a high fidelity, normalized model that is relatively faithful to the domain. This model, though imbued with substantial accidental complexity, contains no duplicate data. But our design work is not yet complete. One of the challenges of the relational paradigm is that normalized models generally aren't fast enough for real-world needs. For many production systems, a normalized schema, which in theory is fit for answering any kind of ad hoc question we may wish to pose the domain, must in practice be further adapted and specialised for specific access patterns. In other words, to make relational stores perform well enough for regular application needs, we have to abandon any vestiges of true domain affinity and accept that we have to change the user's data model to *suit the database engine not the user*. This technique is given the name *denormalization*.

Denormalization involves duplicating data (substantially in some cases) in order to gain query performance. Take as an example users and their contact details. A typical user often has several email addresses, which, in a fully normalised model, we would store in a separate 'EMAIL` table. To reduce joins and the performance penalty imposed by joining between two tables, however, it is quite common to inline this data in the `USER` table, adding one or more columns to store users' most important email addresses.

While denormalization may be a safe thing to do (assuming developers understand the denormalized model and how it maps to their typically domain-centric code, *and* have robust transactional support from the database), it is usually not a trivial task. For the best results, we usually turn to a true RDBMS expert to munge our normalized model into a denormalized one that is aligned with the characteristics of the underlying RDBMS and physical storage tier. In doing this, we accept that there may be substantial data redundancy.

We might be tempted to think that all this design-normalize-denormalize effort is acceptable because it is a one-off task. This school of thought suggests that the cost of the

work is amortized across the entire lifetime of the system (which includes both development and production) such that the effort of producing a performant relational model is comparatively small compared to the overall cost of the project. This is an appealing notion, but in many cases it doesn't match reality, for systems change not only during development, but also during their production lifetimes.

The amortized view of data model change, in which costly changes during development are eclipsed by the long-term benefits of a stable model in production, assumes that systems spend the majority of their time in production environments, and that these production environments are stable. While it may be the case that most systems spend most of their time in production environments, these environments are rarely stable. As business requirements change or regulatory requirements evolve, so must our systems and the data structures on which they are built.

Data models invariably undergo substantial revision during the design and development phases of a project, and in almost every case, these revisions are intended to accommodate the model to the needs of the applications that will consume it once it is in production. So powerful are these initial influences over the model, it becomes near impossible to modify the application and the model at a later point in time to accommodate things they were not originally designed to do.

The technical mechanism through which we evolve a database is known as *migration*. The migration technique has been popularized by productive application development frameworks such as Rails.[2] Migrations provide a structured, step-wise approach to applying a set of database refactorings consistently so that a database can be responsibly evolved to meet the changing needs of the applications that use it.[3]

The problem with the denormalized model is its resistance to the kind of rapid evolution the business demands of its systems. As we've seen with the data center example, the changes imposed on the whiteboard model over the course of implementing a relational solution create a gulf between the conceptual world and the way the data is physically laid out; this dissonance all but prevents business stakeholders from actively collaborating in the further evolution of a system. On the development side, the difficulties in translating changed business requirements into the underlying database structure cause the evolution of the system to lag the evolution of the business. Without expert assistance and rigorous planning, migrating a denormalized database poses several risks. If the migrations fail to maintain storage-affinity, performance can suffer. Just as serious, if

2. See: *http://guides.rubyonrails.org/migrations.html*
3. See: *http://databaserefactoring.com/*

deliberately duplicated data is left orphaned after a migration, we risk compromising the integrity of the data as a whole.

Relational databases—with their rigid schemas and complex modeling characteristics—are not an especially good tool for supporting rapid change. To face up to these challenges we need a model that is closely aligned with the domain without sacrificing performance, that supports evolution while providing enough structural affordances to guide the designer, and that maintains the integrity of the data as it undergoes rapid change and growth. To that end, it's time for us to get to grips with *property graphs* and graph modeling.

The Property Graph Model

The rise of graph databases like Neo4j helps bridge the divide between businesses and *their* data. In terms of domain affinity, Euler's graph theory (from the first half of the eighteenth century) is an early example of structured business analysis, whereby a real-world problem (the Seven Bridges of Königsberg) is purposely abstracted to a data model that can generate useful insight. What graph databases provide is a platform for storing and querying that data to generate insight on-demand.

While there are numerous kinds of graphs, Neo4j, like several other notable graph databases, has settled on the *property graph* as its graph model. The property graph balances simplicity and expressiveness. (The next most-popular alternative, the triple store, popularized by RDF, is theoretically elegant but generally considered less practicable, in large part because in a triple store everything is decomposed into individual nodes and relationships.) Property graphs sacrifice some graph purity for pragmatism by grouping properties into nodes, thereby making them easier to work with. The main abstractions in a property graph are *nodes*, *relationships* and *properties*.

- Nodes are placeholders for properties. Think of nodes as documents that store properties in the form of arbitrary key-value pairs. The keys are strings and the values can be arbitrary data types.
- Relationships connect nodes. Relationships in Neo4j are **always** connected and directed (that is, they always have a *start node* and an *end node*). In addition, every relationship has a label. A relationship's label and its direction add semantic clarity to the connections between nodes. Like nodes, relationships can also house properties. The ability to add properties to relationships is particularly useful for providing additional metadata for graph algorithms, and for constraining queries at run time.

These simple primitives are all we need to create some sophisticated and semantically rich models. For example, in Figure 4-4 we have high-quality information about Shakespeare and some of his plays, together with details of one of the companies that has

recently performed the plays, and a theatrical venue, with geospatial data. We've even added a review. In all, the graph describes and connects at last three different domains. In the diagram we've distinguished these three domains with different coloured nodes: green for the literary nodes, white for the theatrical nodes, and red for the geospatial nodes.

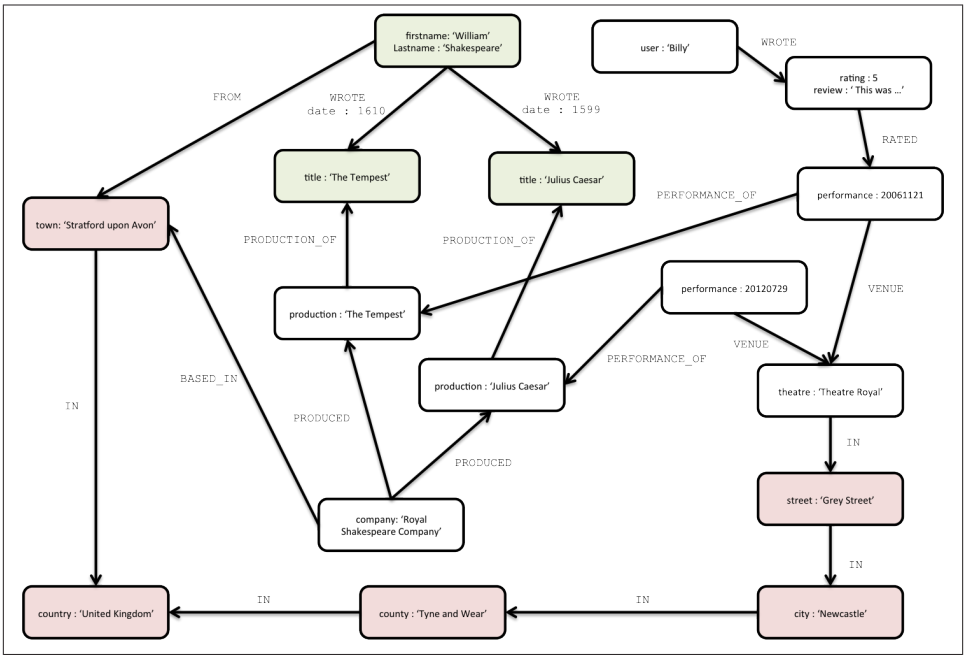


Figure 4-4. An example property graph

Looking first at the literary nodes, we have a node that represents Shakespeare himself, with properties `firstname : 'William'` and `lastname : 'Shakespeare'`. This node is connected to a pair of nodes representing the plays *Julius Caesar* (`title : 'Julius Caesar'`) and *The Tempest* (`title : 'The Tempest'`) via relationships labelled `WROTE`.

Reading this subgraph left-to-right, following the direction of the relationship arrow, conveys the fact that *William Shakespeare wrote Julius Caesar*. If we're interested in provenance, the `WROTE` relationship has a `date` property, which tells us that the play was written in 1599 [4]. It's a trivial matter to see how we could add the rest of Shakespeare's works—the plays and the poems—into the graph simply by adding more nodes to represent each work, and joining them to the Shakespeare node via more `WROTE` relationships.

4. History's actually a little vague on this, but 1599 is a good enough approximation for this example

By *traversing* the WROTE relationship—even by tracing that arrow with our finger—we're effectively doing the kind of work that a graph database performs, albeit at human latency rather than at microsecond-level. As we'll see later, this simple *traversal* primitive is the building block for arbitrarily sophisticated graph queries.

Turning next to the theatrical nodes, you can see that we've added some information about the Royal Shakespeare Company (often known simply as the *RSC*) in the form of a node with the key `company` and value `Royal Shakespeare Company`. The theatrical domain is, unsurprisingly, connected to the literary domain: in our graph, this is reflected by the fact that the RSC has `PRODUCED` versions of *Julius Caesar* and *The Tempest*. The graph also captures details of specific performances: the RSC's production of *Julius Caesar*, for example, was performed on the 29 July 2012. [5] If we're interested in the performance venue, we simply follow the outgoing `VENUE` relationship from the performance node to find that the play was performed at the Theatre Royal.

The graph also allows us to capture reviews of specific performances. In our sample graph we've included just one review, for the 29 July performance, written by the user `Billy`. We can see this in the top right of Figure 4-4 with the interplay of the performance, rating and user nodes. In this case we have a node representing Billy (`user : 'Billy'`) whose outgoing `WROTE` relationship connects to a node representing his review. The review node contains a numeric `rating` property and a free-text `review` property. The review is linked to a specific performance through an outgoing `RATED` relationship. To scale this up to many users, many reviews and many performances we simply add more nodes and more identically-named relationships to the graph.

The third domain, that of geospatial data, comprises a simple hierarchical tree of places. This geospatial domain is connected to the other two domains at several points in the graph. The town of Stratford upon Avon (with property `town : 'Stratford upon Avon'`) is connected to the literary domain as a result of its being Shakespeare's birthplace (Shakespeare is `FROM` Stratford). It is connected to the theatrical domain insofar as it is home for the RSC (the RSC is `BASED_IN` Stratford). To learn more about Stratford upon Avon's geography, we can follow its outgoing `IN` relationship to discover it is in the United Kingdom.

The property graph makes it possible to capture more complex geospatial data. Looking at the Theatre Royal, for example, we see that it is located in `Grey Street`, which is `IN` the city of Newcastle, which is `IN` the county of Tyne and Wear, which ultimately is `IN` the United Kingdom—just like Stratford upon Avon.

5. Part of the RSC's summer season which they traditionally spend touring

As you can see from this relatively simple example, property graphs make it easy to unite different domains—each with its own particular entities, properties and relationships—in a way that not only makes each domain accessible, but also generates insight from the connections between domains.

Unlike document, column, or key-value stores, graph databases don't require us to infer connections between documents using contrived properties such as foreign keys, or out-of-band processing like map-reduce. Instead, relationships are a first-class citizen in the data model. By assembling the simple abstractions of nodes and relationships into connected structures, graph databases allow us to build arbitrarily sophisticated models with high domain affinity. Modeling is simpler and at the same time more expressive than what we experience with both traditional relational database and the other NOSQL stores.

Diagrams are good for communicating the fundamentals of the property graph model, but when it comes to using a database, we need to create, manipulate and query data. Having familiarised ourselves with the property graph model, it's now time to look at query languages.

The Cypher Query Language

Modeling expressivity is only one of the reasons graph databases appeal to the needs of today's IT professionals. The graph model really shines when the interpretation of the domain requires an understanding of how, or in what way, or to what degree entities are related or connected to one another. That is, graph databases address the increasing need to not only store but also to query connected data. In Neo4j the humane and expressive query language *Cypher* is the primary means of accessing the database. Both in this chapter and throughout this book we'll use Cypher extensively to illustrate graph queries and graph constructions and so at this point it's worthwhile taking a brief detour to learn the basics [6] of the language.

Other query languages

Other graph databases have other means of querying data. Most, including Neo4j, support the RDF query language SPARQL [7] and the imperative, path-based query language

6. This section is not intended to be a reference document for Cypher, merely a friendly introduction to enable us to explore more interesting graph query scenarios. For reference documentation see: *http://docs.neo4j.org/chunked/milestone/cypher-query-lang.html*

7. *http://www.w3.org/TR/rdf-sparql-query/*

Gremlin. [8] Our interest, however, is in the expressive power of the *declarative* model, and so in this book we'll focus almost exclusively on Cypher.

Cypher Philosophy

Cypher is a humane query language for developers and database operations professionals alike. By *humane* we mean a language that is aligned with how we intuitively describe graphs—which, as we've already seen, is often through a diagram. The interesting thing about graph diagrams is that they tend to contain specific instances of nodes and relationships, rather than classes or archetypes. Even very large graphs are illustrated using smaller subgraphs made from real nodes and relationships. In other words, we describe graphs using *specification by example*.

Cypher enables a user (or an application acting on behalf of a user) to ask the database to find data that matches a pattern. Colloquially, we say that we ask the database to "find things like this"--and the way we specify the things we want the database to find is to draw a picture of them, in ASCII art. We have to employ a few tricks to get around the fact that a query language has only one dimension (text proceeding from left to right), whereas a graph diagram can be laid out in two dimensions, but on the whole, Cypher patterns follow very naturally from the way we draw graphs on the whiteboard.

Like most query languages, Cypher is composed of clauses. A reasonably simple query is made up of `START`, `MATCH` and `RETURN` clauses (we'll describe the other clauses you can use in a Cypher query later in this chapter):

- `START`: Specifies one or more starting points in the graph. These starting points are obtained via index lookups or, more rarely, accessed directly based on element IDs.
- `MATCH`: This is the *specification by example* part. Using ASCII characters to represent nodes and relationships (round brackets for nodes, dashes, and greater-than and less-than signs for relationships), we *draw* the data we're interested in.
- `RETURN`: Specifies which nodes, relationships and properties in the matched data should be returned to the client.

Here's an example of a Cypher query that uses these three clauses to find the plays written by Shakespeare:

```
START bard=node:authors('firstname:"William" AND lastname:"Shakespeare"')
MATCH (bard)-[:WROTE]->(plays)
RETURN plays
```

8. *https://github.com/tinkerpop/gremlin/wiki/*

At the heart of this query is the simple pattern `(bard)-[:WROTE]->(plays)`. This pattern describes two nodes, one of which we've bound to an identifier `bard`, the other to an identifier `plays`. These nodes are connected by way of a `WROTE` relationship, which extends from `bard` to `plays`. The dashes and greater-than sign that surround the relationship name (`-->`) indicate relationship direction. The rounded brackets help further discriminate the node identifiers.

Now this pattern could, in theory, occur many times throughout our graph data. To localise the query, we need to anchor some part it to one or more places in the graph. What we've done with the `START` clause is lookup a real node in the graph—the node representing Shakespeare. We bind this Shakespeare node to the `bard` identifier, which then carries over to the `MATCH` clause. This has the effect of anchoring our pattern to a specific point in the graph. Cypher can then *rotate* the pattern around this anchor point to discover actual nodes to bind to the `plays` identifier. While `bard` will always be anchored to the one real Shakespeare node in our graph, `plays` will (in our sample data) be bound to a list of nodes representing the plays *Julius Caesar* and *The Tempest*. These nodes are returned to the client in the `RETURN` clause.

The other clauses we can use in a Cypher query include:

- `WHERE`: Provides criteria for filtering portions of a pattern.
- `CREATE` and `CREATE UNIQUE`: Creates nodes and relationships.
- `DELETE`: Removes nodes, relationships and properties.
- `SET`: Set property values.
- `FOREACH`: Performs an updating action for each element in a list.
- `WITH`: Divides a query into multiple, distinct parts.

If these clauses look familiar—especially if you're a SQL developer—that's great! Cypher isn't intended to cause headaches when it's written or read; it's intended to be familiar enough to help you rapidly along the learning curve. At the same time, it's different enough to emphasize that we're dealing with graphs **not** sets. With that in mind, let's look in more detail at the kind of Cypher queries we can write for the graph in Figure 4-4.

Creating Graphs

To create the Figure 4-4 graph, we use a number of `CREATE` statements to build the overall structure. These statements are executed by the Cypher runtime within a single transaction such that once the statements have executed, we can be confident the graph is

present in the database (or not at all, should the transaction fail). As we might expect, Cypher has a humane and visual way of building graphs: [9]

```
CREATE shakespeare = { firstname : 'William', lastname : 'Shakespeare' },
       juliusCaesar = { title : 'Julias Caesar' },
       (shakespeare)-[:WROTE { date : 1599 }]->(juliusCaesar),
       theTempest = { title : 'The Tempest' },
       (shakespeare)-[:WROTE { date : 1610}]->(theTempest),
       rsc = { company : 'Royal Shakespeare Company' },
       production1 = { production : 'Julius Caesar' },
       (rsc)-[:PRODUCED]->(production1),
       (production1)-[:PRODUCTION_OF]->(juliusCaesar),
       performance1 = { performance : 20120729 },
       (performance1)-[:PERFORMANCE_OF]->(production1),
       production2 = { production : 'The Tempest' },
       (rsc)-[:PRODUCED]->(production2),
       (production2)-[:PRODUCTION_OF]->(theTempest),
       performance2 = { performance : 20061121 },
       (performance2)-[:PERFORMANCE_OF]->(production2),
       performance3 = { performance : 20120730 },
       (performance3)-[:PERFORMANCE_OF]->(production1),
       billy = { user : 'Billy' },
       rating = { rating: 5, review : 'This was awesome!' },
       (billy)-[:WROTE]->(rating),
       (rating)-[:RATED]->(performance1),
       theatreRoyal = { theatre : 'Theatre Royal' },
       (performance1)-[:VENUE]->(theatreRoyal),
       (performance2)-[:VENUE]->(theatreRoyal),
       greyStreet = { street : 'Grey Street' },
       (theatreRoyal)-[:IN]->(greyStreet),
       newcastle = { city : 'Newcastle' },
       (greyStreet)-[:IN]->(newcastle),
       tyneAndWear = { county: 'Tyne and Wear' },
       (newcastle)-[:IN]->(tyneAndWear),
       uk = { country: 'United Kingdom' },
       (tyneAndWear)-[:IN]->(uk),
       stratford = { town : 'Stratford upon Avon' },
       (stratford)-[:IN]->(uk),
       (rsc)-[:BASED_IN]->(stratford)
```

In the Cypher code above, we're doing two different things: creating nodes and their properties, and relating nodes (with relationship properties where necessary). For example `CREATE shakespeare = { firstname : 'William', lastname : 'Shakespeare' }` creates a node representing William Shakespeare. The newly created node is assigned to the identifier `shakespeare`, which remains available for the duration of

9. Normally it's applications that issue database commands: humans don't normally have to see so many `CREATE` statements!

the current scope [10]. This `shakespeare` identifier is used later in the code to attach relationships to the underlying node. For example, the clause `CREATE (shakespeare)-[:WROTE { date : 1599 }]->(juliusCaesar)"` creates a `WROTE` relationship *from* Shakespeare *to* the play Julius Caesar. With enough of these, we can build some very interesting and expressive graphs.

Unlike the relational model, these commands don't introduce any accidental complexity into the graph. The information meta-model—that is, the structuring of nodes through relationships—is kept separate from the business data, which lives exclusively as properties. We no longer have to worry about foreign key and cardinality constraints polluting our real data, since both are explicit in the graph model as the nodes and the semantically rich relationships that interconnect them.

We can modify the graph at a later point in time in two different ways. We can, of course, continue using the `CREATE` clause to simply add to the graph. But we can also use `CREATE UNIQUE`, which has the semantics of *ensuring* a particular sub-graph structure—some of which may already exist, some of which may be missing—is in place once the command has executed. In practice, we tend to use `CREATE` when we're adding to the graph and don't mind duplication, and `CREATE UNIQUE` when duplication is not permitted by the domain.

Beginning a Query

Now that we have a graph, we can start to query it. In Cypher we always begin our queries from one or more well-known starting points in the graph—what are called *bound* nodes. (We can also start a query by binding to one or more relationships, but this happens far less frequently than binding to nodes.). This is the purpose of the `START` clause: to give us some starting points, usually drawn from underlying indexes, from which we can explore the remainder of the graph.

For instance, if we wanted to discover more about the performances at the Theatre Royal, we would use that as our starting point. However, if we were more interested in the works of reviewers or the county of Tyne and Wear, we'd use those nodes to start instead. Since we want to know about all things Shakespeare that have taken place in the Theatre Royal in Newcastle, we'll start our query with these three nodes.

```
START theater=node:venues(theatre = 'Theatre Royal'), newcastle=node:cities(city = 'Newcastle'),
      bard=node:authors('firstname:"William" AND lastname:"Shakespeare"')
```

10. The identifiers we introduce are only available within the current scope. Should we wish to give long-lived names to nodes (or relationships) then we simply add them to an index.

This START clause identifies all nodes that have a property key venue and property value Theatre Royal in a node index called venues. The results of this index lookup are bound to the identifier theater. (What if there are many Theatre Royal nodes in this index? We'll deal with that shortly.) In a second index lookup, we find the node representing the city of Newcastle; we bind this node to the identifier newcastle.

To find the Shakespeare node itself, we use a more sophisticated query syntax: we declare that we want to retrieve the node that has properties firstname:William *and* last name:Shakespeare [11]. We bind the result of this lookup to bard.

From now on in our query, wherever we use the identifiers theater, newcastle and bard in a pattern, that pattern will be anchored to the real nodes associated with these identifiers. In effect, the START clause localises the query, giving us starting points from which to match patterns in the surrounding nodes and relationships.

Declaring information patterns to find

The MATCH clause in Cypher is where the magic happens. Much as the CREATE clause tries to convey intent using ASCII art to describe the desired state of the graph, so the MATCH clause uses the same syntax to describe patterns to discover in the database. We've already looked at a very simple MATCH clause; now we'll look at a more complex pattern that finds all the Shakespeare performances at Newcastle's *Theatre Royal*:

```
MATCH (newcastle)<-[:IN*1..4]-(theater)<-[:VENUE]-()-[p:PERFORMANCE_OF]->()-[:PRODUCTION_OF]->(pla
```

This MATCH pattern uses several syntactic elements we've not yet come across. As well as including bound nodes based on index lookups (discussed above), it uses pattern nodes, arbitrary depth paths, and anonymous nodes. Let's take a look at these in turn:

- The identifiers newcastle, theater and bard carry over from the START clause, where they were bound to nodes in the graph using index lookups, as described above.
- If there are several Theatre Royals in our database (the cities of Plymouth, Bath, Winchester and Norwich all have a Theatre Royal, for example), then theater will be bound to all these nodes. To restrict our pattern to the Theatre Royal in Newcastle, we use the syntax <-[:IN*1..4]-, which means the theater node can be no more than four outgoing IN relationships away from the node representing the city of Newcastle. By providing a variable depth path, we allow for relatively fine-grained address hierarchies (comprising, for example street, district or borough, and city).
- The syntax (venue)<-[:VENUE]-() uses the *anonymous* node, hence the empty parentheses. Knowing the data as we do, we expect the anonymous node to match

11. This is in fact Lucene query syntax. Lucene is Neo4j's default index provider.

performances, but because we're not interested in using the details of individual performances elsewhere in the query or in the results, we don't name the node or bind it to an identifier.

- We use the anonymous node again to link the performance to the production (`()-[:PERFORMANCE_OF]->()`). If we were interested in returning details of performances and productions, we would replace these occurrences of the anonymous node with identifiers: `(performance)-[:PERFORMANCE_OF]->(production)`.
- The remainder of the `MATCH` is a straightforward `(play)<-[w:WROTE]-(bard)` node-to-relationship-to-node pattern match. This pattern ensures we only return plays written by Shakespeare. Since `(play)` is joined to the anonymous production node, and by way of that to the performance node, we can safely infer that it has been performed in Newcastle's Theatre Royal. In naming the play node we bring it into scope so that we can use it later in the query.
- Finally, while most relationships are not bound to identifiers (for example, `-[:VENUE]-` doesn't have an identifier), two relationships `-[w:WROTE]-` and `-[p:PERFORMANCE_OF]-` do include identifiers (`w` and `p` respectively). This allows us to refer to those relationships later in the query (in the `WHERE` and `RETURN` clauses).

At this point our query looks like this:

```
START theater=node:venues(theatre = 'Theatre Royal'), newcastle=node:cities(city = 'Newcastle'),
      bard=node:authors('firstname:"William" AND lastname:"Shakespeare"')
MATCH (newcastle)<-[:IN*1..4]-(theater)<-[:VENUE]-()-[p:PERFORMANCE_OF]->()-[:PRODUCTION_OF]->(pla
RETURN play
```

Running this query yields all the Shakespeare plays that have been performed at the Theatre Royal in Newcastle. That's great if we're interested in the entire history of Shakespeare at the Theatre Royal, but if we're only interested in specific plays, productions or performances, we need somehow to constrain the set of results.

Constraining Matches

We constrain graph matches using the `WHERE` clause to stipulate one or more of the following:

- That certain additional nodes and relationships must be present (or absent) in the matched subgraphs.
- That specific properties on matched nodes and relationships must be present (or absent)--irrespective of their values.
- That certain properties on matched nodes and relationships must have specific values.

Compared to the MATCH clause, which describes structural relationships and assigns identifiers to parts of the pattern, WHERE is more of a tuning exercise to refine the current pattern match. For example, if we want to restrict the range of plays that we're interested in to Shakespeare's *final period*, we can supplement our MATCH with the following WHERE clause:

```
WHERE w.date > 1608
```

Adding this clause means that for each successful match, the Cypher execution engine will check that the WROTE relationship between the Shakespeare node and the matched play has a date property with a value greater than 1608—the generally accepted beginning of Shakespeare's late period. Matches with a WROTE relationship whose date value is greater than 1608 will pass the test; these plays will then be included in the results. Matches that fail the test will not be included in the results. By adding this clause we ensure that only plays from Shakespeare's late period are returned.

Processing Results

Cypher's RETURN clause allows us to perform some processing on the matched graph data before returning it to the user (or the application) that executed the query.

The simplest thing we can do is to return the plays that matched our criteria:

```
RETURN play
```

We can enrich this result in several ways: by aggregating, ordering, filtering and limiting the returned data. For example, if we're only interested in how many plays match our criteria, we apply the count aggregator:

```
RETURN count(play)
```

If we want to rank the plays by the number of performances, we count a different value, and use ORDER BY:

```
RETURN play, count(p) AS performance_count ORDER BY performance_count DESC LIMIT 10
```

This RETURN clause counts the number of PERFORMANCE_OF relationships using the identifier p (which is bound to the PERFORMANCE_OF relationships in the MATCH clause) and aliases the result as performance_count. It then orders the results based on performance_count, with the most frequently performed listed first, and limits the final results to the top 10.

Query Chaining

Before we leave our brief tour of Cypher, there is one more feature that it is useful to know about—the WITH clause. Sometimes it's just not practical (or possible) do everything you want in a single MATCH clause. The WITH clause allows us to chain together several matches, with the results of the previous MATCH being piped into the next.

As queries become more complex, `WITH` instils a sense of discipline, in large part because it insists on separating read-only clauses from write-centric `SET` operations.

The Neo4j Core API and Traverser Framework

Besides Cypher, which we use for graph pattern matching, Neo4j has several lower-level APIs, which developers can use to perform **graph traversals**. A traversal is effectively a set of instructions for following a path, from one or more starting points to some informational goal elsewhere in the graph. Traversals are the building blocks of other graph query approaches. In Neo4j, traversals can be programmed either imperatively against the low-level **Core API**, or declaratively against the **Traverser Framework**.

The Core API allows developers to fine-tune their queries so that they exhibit high affinity with the underlying graph. A well-written Core API query is often faster than any other approach. The downside is that such queries can be verbose, requiring considerable developer effort. Moreover, their high affinity with the underlying graph makes them tightly coupled to its structure: when the graph structure changes, they can often break. Cypher can be more tolerant of structural changes—things such as variable length paths help mitigate variation and change.

The Traverser Framework is both more loosely coupled than the Core API (since it allows the developer to declare informational goals), and less verbose, and as a result a query written using the Traverser Framework typically requires less developer effort than the equivalent written using the Core API. Because it is a general-purpose framework, however, the Traverser Framework tends to perform marginally less well than a well-written Core API query.

If we find ourselves in the **unusual situation** of coding at with the Core API or Traverser Framework (and thus eschewing Cypher and its affordances), it's because we are working on an edge case where we need to finely craft an algorithm that cannot be expressed effectively using Cypher's pattern matching. Choosing between the Core API and the Traverser Framework is a matter of deciding whether the higher abstraction/lower coupling of the Traverser Framework is sufficient, or whether the close-to-the-metal/higher coupling of the Core API is in fact necessary for implementing an algorithm correctly and in accordance with our performance requirements.

Graph Modeling in a Systems Management Domain

Now that we understand the property graph model and the basics of the Cypher query language, we're ready to revisit the systems management domain from Figure 4-1. In this section we'll walk through the analysis and modeling steps necessary to build a graph representation of the domain.

In the early stages of analysis the work required of us is similar to the relational approach: using lo-fi methods such as whiteboard sketches, we describe and agree upon the domain. After that, however, the methodologies start to diverge. Instead of transforming the domain model into tables, we enrich it, with the aim of producing a high-fidelity representation of the domain suited to what we are trying to achieve in that domain. That is, for each entity in our domain we ensure that we've captured both the properties and the connections to neighbouring entities necessary to support our application goals.

Helpfully, domain modeling is completely isomorphic to graph modeling. By ensuring the correctness of the domain model we're implicitly improving the graph model, since in a graph database **what you sketch on the whiteboard is typically what you store in the database**. Remember, the domain model is not a transparent, context-free window onto reality: rather, it is a purposeful abstraction of those aspects of our domain relevant to our application goals. By enriching the domain model, we effectively produce a graph model attuned to our application's data needs.

In graph terms what we're doing is ensuring each node has the appropriate properties so that it can fulfil its specific domain responsibilities. But we're also ensuring that every node is in the correct semantic context; we do this by creating named and directed (and often attributed) relationships between the nodes to capture the structural aspects of the domain. For our data center scenarion, the resulting graph model looks like Figure 4-5.

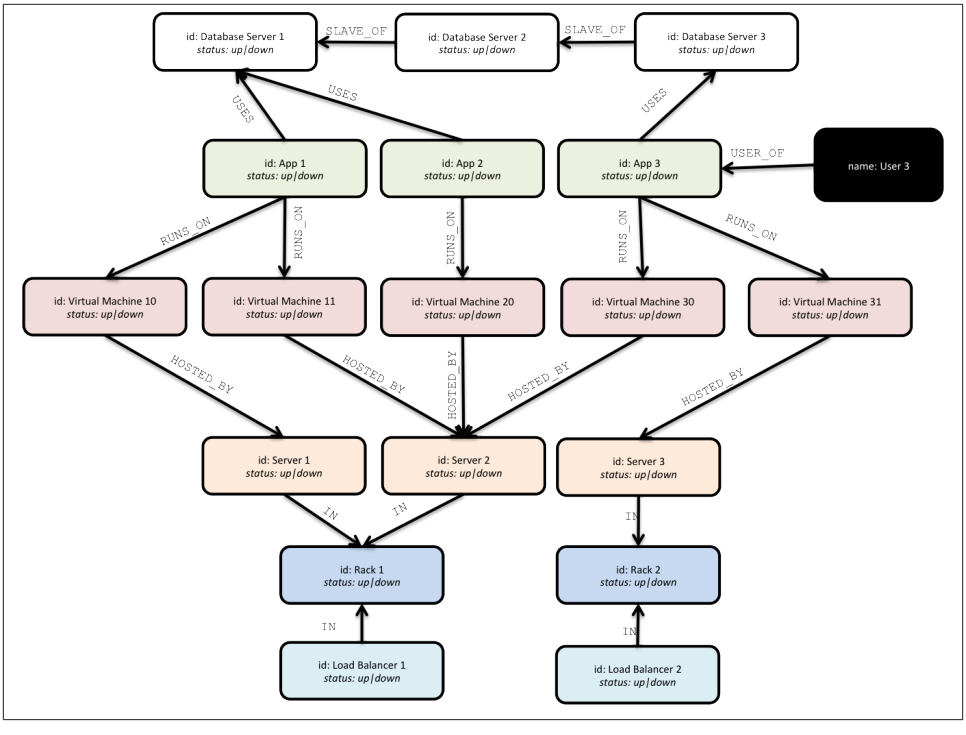


Figure 4-5. Example graph for the data center deployment scenario

And logically, that's **all** we need to do. No tables, no normalization, no de-normalization. If we have a high-fidelity domain model then moving that into the database is, logically at least, child's play.

Modeling guidelines

While there is usually a very close affinity between the domain model and the resulting graph model, there are occasions where the two will necessarily diverge slightly. These are our rules of thumb for refining the graph model:

- Use nodes to represent *things* in the domain; use relationships to represent the structural relationships between those things.
- Every entity—that is, everything with an identity—should be a separate node.
- Complex, multi-part property values—addresses, for example—should usually be pulled out into separate nodes, even though they typically don't have their own global identity.

- Actions should be modeled in terms of their products, which should be represented as nodes. Prefer `(alice)-[:WROTE]->(review)-[:OF]->(film)` to `(alice)-[:REVIEWED]->(film)`.

Be diligent about discovering and capturing domain entities. As we shall see later in the chapter, it's relatively easy to transpose roles that ought to be assigned to nodes to sloppily-named relationships instead. This is particularly true when modeling actions or activities. Look for the thing—the result or product—that can be represented as a node. Even the smallest domain entities can require their own nodes.

As described in the next section, reviewing the model with an eye to queryability will help determine which of these guidelines apply in any particular case.

Testing the model

Once we've refined our domain model, the next step is to test how suitable it is for answering realistic queries. While graphs are great for supporting a continuously evolving structure (and therefore for correcting any erroneous earlier design decisions), there are a number of design decisions which, once they are baked into our application, can hamper us further down the line. By reviewing the domain model and the resulting graph model at this stage we can avoid these pitfalls; thereafter, changes to the graph structure will likely be driven solely by changes in the business, rather than by the need to mitigate a poor design decision.

In practice there are two techniques that we can apply here. The first, and simplest, is just to check that the graph reads well. Pick a start node, and then follow relationships to other nodes, reading each node's role and each relationship's name as you go. Doing so should create some sensible sentences. For our data center example, we can read off sentences like "Load balancer 1 fronts the App, which consists of App Instance 1, 2 and 3, and the Database, which resides on Database Machine 1 and Database Machine 2," and "Blade 3 runs VM 3 which hosts App Instance 3." If reading the graph in this way makes sense, we can be reasonably confident it has good domain affinity and fidelity.

To further increase our confidence, we also need to consider the queries we'll run on the graph. Here we adopt a *design for queryability* mindset. To validate that the graph supports the kinds of queries we expect to run on it, we must describe those queries. This requires us to understand our end-users' goals; that is, the use cases to which the graph is to be applied. In our data center scenario, for example, one of our use cases involves end-users reporting that an application or service is unresponsive. To help these users, we must identify the cause of the unresponsiveness and then resolve it. To determine what might have gone wrong we need to identify what's on the path between the user and the application, and also what the application depends on to deliver functionality to the user. Given a particular graph representation of the data center domain,

if we can craft a Cypher query that addresses this use case, we can be even more certain that the graph meets the needs of our domain.

Continuing with our example use case, let's assume that we can update the graph from our regular network monitoring tools, thereby providing us with a near real-time view of the state of the network [12]. When a user reports a problem, we can limit the physical fault-finding to problematic network elements between the user and the application and the application and its dependencies. In our graph we can find the faulty equipment with the following query:

```
START user=node:users(id = 'User 3')
MATCH (user)-[*1..5]-(asset) WHERE asset.status! = 'down'
RETURN distinct asset
```

That is, starting from the user who reported the problem, we `MATCH` against all assets in the graph along an undirected path of length 1 to 5. We store asset nodes with a `status` property whose value is `'down'` in our results. The exclamation mark in `asset.status!` ensures that assets that lack a `status` property are not included in the results. `RETURN distinct asset` ensures that unique assets are returned in the results, no matter how many times they are matched.

Given that such a query is readily supported by our graph, we gain confidence that the design is fit for purpose.

Common Modeling Pitfalls

While graph modeling is a very expressive way of mastering the complexity in a problem domain, expressivity in and of itself is no guarantee that a particular graph is fit for purpose. In fact there have been occasions where even those of us who work with graphs all the time make mistakes. In this section we'll take a look at a model that went wrong. In so doing, we'll learn how to identify problems early in the modeling effort, and how to fix them.

Email Provenance Problem Domain

In this case we were working on a problem involving the analysis of email communications. Communication pattern analysis is a classic graph problem. Normally, we'd interrogate the graph to discover subject matter experts, key influencers, and the communication chains through which information is propagated. On this occasion, however, instead of looking for positive role models (in the form of experts) we were search-

12. With a large physical network, we might use Complex Event Processing to process streams of low-level network events, updating the graph only when the CEP solution raises a significant domain event. See: *http://en.wikipedia.org/wiki/Complex_event_processing*

ing for rogues: that is, suspicious patterns of email communication that fall foul of corporate governance—or even break the law.

A Sensible First Iteration?

In analyzing the domain we learnt about all the clever patterns that potential wrongdoers adopt to cover their tracks: using BCC, using aliases—even conducting conversations with those aliases to mimic legitimate interactions between real business stakeholders. Based on this analysis we produced a representative graph model that seemed to capture all the relevant entities and their activities.

To illustrate this early model, we'll use Cypher's `CREATE` clause to generate some nodes representing users and aliases. We also generate a relationship that shows that Alice is one of Bob's known aliases. (We'll assume that the underlying Neo4j instance is indexing these nodes so that we can later look them up and use them as starting points in our queries.) Here's the Cypher query to create our first graph:

```
CREATE alice = {username : 'alice'}
CREATE bob = {username : 'bob'}
CREATE charlie = {username : 'charlie'}
CREATE davina = {username : 'davina'}
CREATE edward = {username : 'edward'}

CREATE (alice)-[:ALIAS_OF]->(bob)
```

The resulting graph model makes it easy to observe that *Alice is an alias of Bob*, as we see in Figure 4-6.

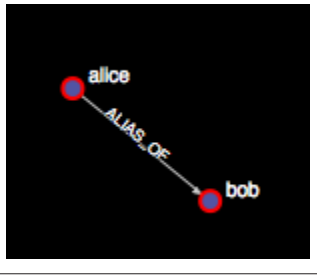


Figure 4-6. Users and aliases

Now we join the users together through the emails they've exchanged:

```
CREATE (bob)-[:EMAILED]->(charlie)
CREATE (bob)-[:CC]->(davina)
CREATE (bob)-[:BCC]->(edward)
```

At first sight this looks like a reasonably faithful representation of the domain. Each clause lends itself to being read comfortably left to right, thereby passing one of our

informal tests for correctness. For example, it's deceptively easy to read the sentence "Bob emailed Charlie". The limitations of this model only emerge when it becomes necessary to determine exactly *what* was exchanged by the potential miscreant Bob (and his alter-ego Alice). We can see that Bob CC'd or BCC'd some people, but we can't see the most important thing of all: the email itself.

This first modeling attempt results in a star-shaped graph. Bob is represented by a central node; his actions of emailing, copying and blind-copying are represented by relationships that extend from Bob to the nodes representing the recipients of his mail. The problem is, as we see in Figure 4-7, the most critical element of the data (the actual *email*) is missing.

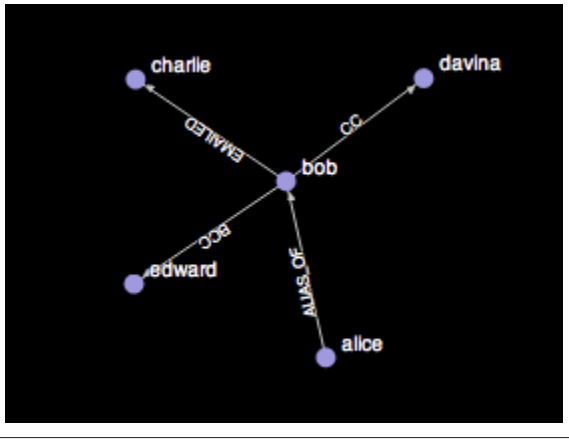


Figure 4-7. Missing email node leads to lost information

That such a structure is lossy becomes evident when we pose the following query:

```
START bob=node:users(username, "Bob"), charlie=node:users(username, "Charlie")
MATCH (bob)-[e:EMAILED]->(charlie)
RETURN e
```

This query returns the `EMAILED` relationships between Bob and Charlie (there will be one for each email that Bob has sent to Charlie). This tells us that emails have been exchanged, but it tells us nothing about the emails themselves. We might think we can remedy the situation by adding properties to the `EMAILED` relationship to represent an email's attributes, but that's just playing for time. Even with properties attached to each `EMAILED` relationship, we would still be unable to correlate between the `EMAILED`, `CC` and `BCC` relationships; that is, we would be unable to say which emails were copied and blind-copied, and to whom.

The fact is we've unwittingly made a simple modeling mistake, caused mostly by a lax use of English rather than any shortcomings of graph theory. Our everyday use of lan-

guage has lead us to focus on the verb "emailed" rather than the email itself, and as a result we've produced a model lacking domain insight.

In English, it's easy and convenient to shorten the phrase "Bob sent an email to Charlie" to "Bob emailed Charlie". In most cases that loss of a noun (the actual email) doesn't matter since the intent is still clear. But when it comes to our forensics scenario, these elided statements are problematic. The intent remains the same, but the details of the number, contents and recipients of the emails that Bob sent have been lost through having been folded into a relationship `EMAILED`, rather than being modeled explicitly as nodes in their own right.

Second Time's the Charm

To fix our lossy model, we need to insert email nodes to represent the real emails exchanged within the business, and expand our set of relationship names to encompass the full set of addressing fields that email supports. Now instead of creating lossy structures like this:

```
CREATE (bob)-[:EMAILED]->(charlie)
```

-we'll instead create more detailed structures, like this:

```
CREATE email = {id : '1234', content : "Hi Charlie, ... Kind regards, Bob"}

CREATE (bob)-[:SENT]->(email)
CREATE (email)-[:TO]->(charlie)
CREATE (email)-[:CC]->(davina)
CREATE (email)-[:CC]->(alice)
CREATE (email)-[:BCC]->(edward)
```

This results in the kind of graph structure we see in Figure 4-8:

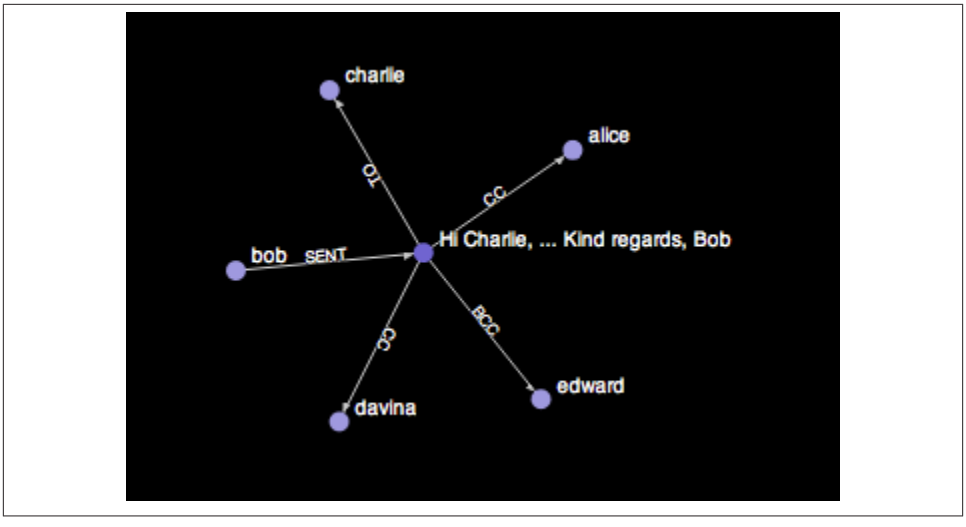


Figure 4-8. High fidelity modeling yields a classic star graph for a single email

Of course in a real system there will be many more of these emails with their intricate web of interactions for us to explore. It's quite easy to imagine that over time many more `CREATE` statements are executed as the email server logs the interactions, like so:

```
CREATE email_1234 = {id : '1234', content : "email contents"}
CREATE (bob)-[:SENT]->(email_1234)
CREATE (email_1234)-[:TO]->(charlie)
CREATE (email_1234)-[:TO]->(davina)
CREATE (email_1234)-[:CC]->(alice)
CREATE (email_1234)-[:BCC]->(edward)

CREATE email_2345 = {id : '2345', content : "email contents"}
CREATE (bob)-[:SENT]->(email_2345)
CREATE (email_2345)-[:TO]->(davina)
CREATE (email_2345)-[:BCC]->(edward)

CREATE email_3456 = {id : '3456', content : "email contents"}
CREATE (davina)-[:SENT]->(email_3456)
CREATE (email_3456)-[:TO]->(bob)
CREATE (email_3456)-[:CC]->(edward)

CREATE email_4567 = {id : '4567', content : "email contents"}
CREATE (charlie)-[:SENT]->(email_4567)
CREATE (email_4567)-[:TO]->(bob)
CREATE (email_4567)-[:TO]->(davina)
CREATE (email_4567)-[:TO]->(edward)

CREATE email_5678 = {id : '5678', content : "email contents"}
CREATE (davina)-[:SENT]->(email_5678)
CREATE (email_5678)-[:TO]->(alice)
```

```
CREATE (email_5678)-[:BCC]->(bob)
CREATE (email_5678)-[:BCC]->(edward)
```

In turn this leads to interesting graphs, as we see in Figure 4-9.

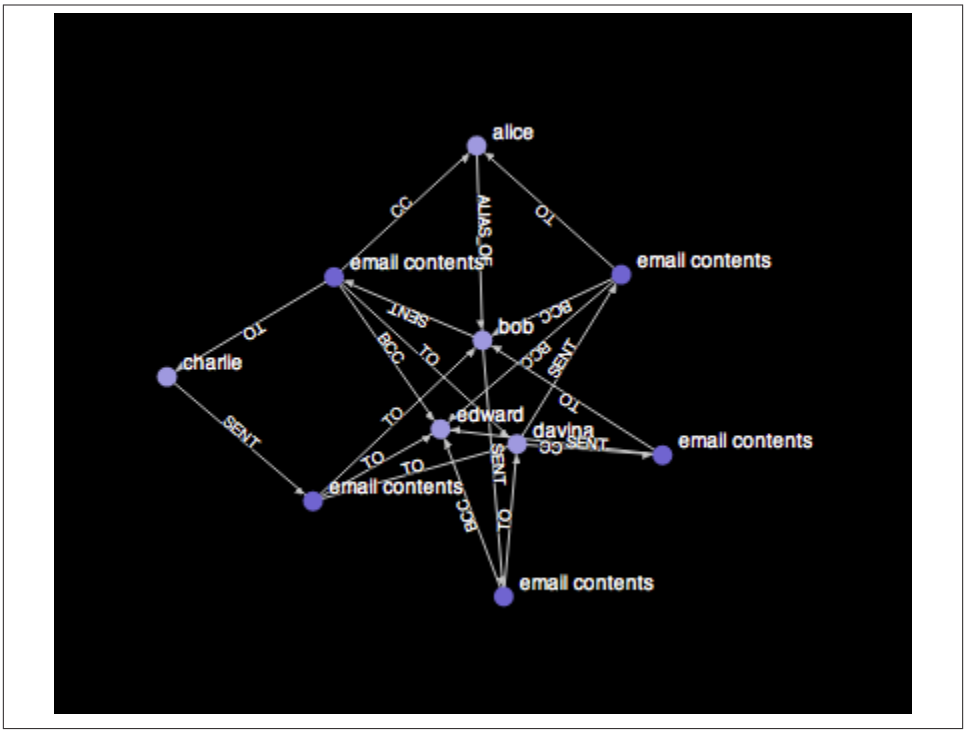


Figure 4-9. A high fidelity graph of email interactions

Now we can query this high-fidelity graph to identify potentially suspect behaviour:

```
START bob=node:users(username, 'Bob'), charlie=node:users(username, 'Charlie'), davina=node:users(
MATCH (bob)-[:SENT]->(email)-[:TO]->(charlie),
      (bob)-[:SENT]->(email)-[:CC]->(davina)
      (bob)-[:SENT]->(email)-[:CC]->(alice)
RETURN email
```

Here we retrieve (and process) those emails that Bob sent to Charlie while copying Davina and Alice (trying to throw us off the scent, since we know Alice is one of Bob's aliases). And since both Cypher and the underlying graph database have graph affinity these queries—even over large datasets—run very quickly.

Evolving the Domain

As with any database, our graph serves a system that is bound to evolve over time (if it's at all useful). So what should we do when the graph evolves? How do we know what breaks, or indeed how to we even tell that something has broken? The fact is, we can't avoid *migrations* in a graph database: they're a fact of life just like with any data store. But in a graph database they're often simpler.

In a graph, to add new facts or compositions, we tend to add nodes and relationships rather than changing the model in place. Adding to the graph using *new* kinds of relationships will not affect any existing queries, and is completely safe. Changing the graph using *existing* relationship types, and changing the properties (not just the property values) of existing nodes *might* be safe, but we need to run a representative set of queries to maintain confidence that the graph is still fit for purpose after the the structural changes. However these activities are precisely the same kind of actions we perform during normal database operation, so in a graph world a migration really is just business as normal.

At this point we have a useful graph of who sent a particular email and who received it (by whatever means). But of course one of the joys of email is that recipients can forward or reply to an email they've received, to increase interaction and knowledge sharing, and in some cases even to leak critical business information. Given we're looking for suspicious communication patterns, it makes sense for us to also take into account forwarding and replies.

At first glance, there would appear to be no need to use database migrations to update our graph to support our new use case. The simplest additions we can make involve adding `FORWARDED` and `REPLIED_TO` relationships into the graph. Doing so won't affect any pre-existing queries: these new relationships will be ignored because the pre-existing queries aren't coded to recognise them.

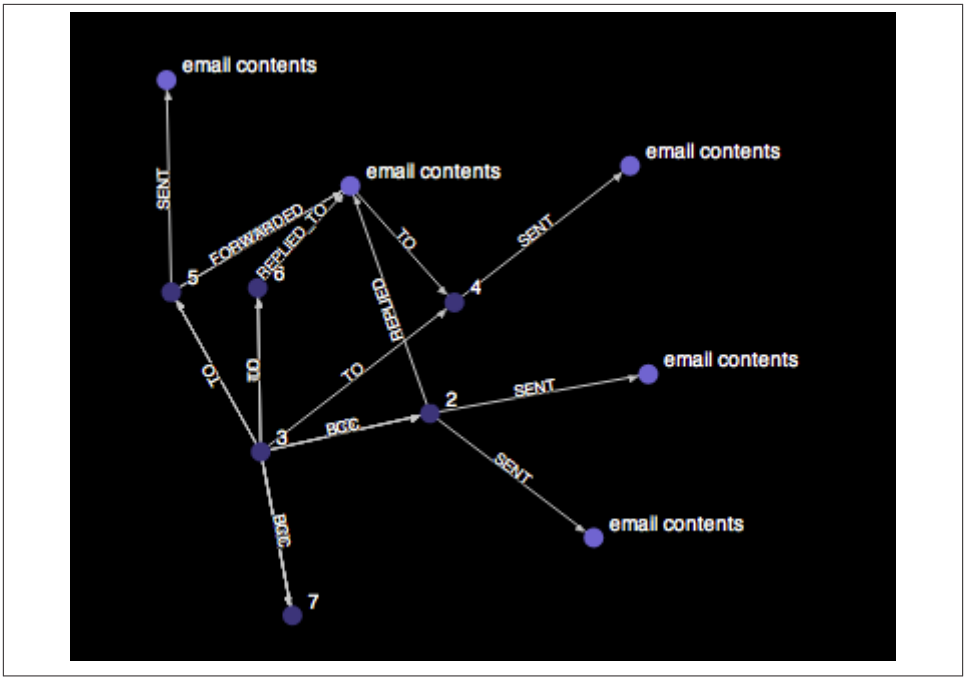


Figure 4-10. A naive, lossy approach fails to appreciate that forwarded and replied emails are first-class entities

However, this approach quickly proves inadequate. Adding `FORWARDED` or `REPLIED` relationships is naive and lossy in much the same way as our original use of an `EMAILED` relationship. To illustrate this, consider the following `CREATE` statements:

```
START email = node:emails(id, 1234)
CREATE (alice)-[:REPLIED_TO]->(email)
CREATE (davina)-[:FORWARDED]->(email)-[:TO]->(charlie)
```

In the first `CREATE` statement we're trying to record the fact that Alice replied to a particular email. The statement makes logical sense when read from left to right, but the sentiment is lossy—we can't tell whether Alice replied to all the recipients of the `email_1234` or directly to the author. All we know that is that some reply was sent. The second statement also reads well from left to right: Davina forwarded `email_1234` to Charlie. The problem is, we already use the `TO` relationship to indicate that a given email has a `to` header that identifies the primary recipients. Reusing `TO` here makes it impossible to tell who was a recipient and who received a forwarded version of an email.

To resolve this problem, we have to consider the fundamentals of the domain. A reply to an email is itself a new email, albeit with some relationship to a previous message. Whether the reply is to the original sender, all recipients, or a subset can be easily mod-

eled using the same familiar TO, CC and BCC relationships, while the original email itself can be referenced via a REPLY_TO relationship. Here's a revised series of writes resulting from several email actions:

```
CREATE email = {id : '0', content : "email"}
CREATE (bob)-[:SENT]->(email)
CREATE (email)-[:TO]->(charlie)
CREATE (email)-[:TO]->(davina)

CREATE reply_1 = {id : '1', content : "response"}
CREATE (reply_1)-[:REPLY_TO]->(email)
CREATE (davina)-[:SENT]->(reply_1)
CREATE (reply_1)-[:TO]->(bob)
CREATE (reply_1)-[:TO]->(charlie)

CREATE reply_2 = {id : '2', content : "response"}
CREATE (reply_2)-[:REPLY_TO]->(email)
CREATE (bob)-[:SENT]->(reply_2)
CREATE (reply_2)-[:TO]->(davina)
CREATE (reply_2)-[:TO]->(charlie)
CREATE (reply_2)-[:CC]->(alice)

CREATE reply_11 = {id : '11', content : "response"}
CREATE (reply_11)-[:REPLY_TO]->(reply_1)
CREATE (charlie)-[:SENT]->(reply_11)
CREATE (reply_11)-[:TO]->(bob)
CREATE (reply_11)-[:TO]->(davina)

CREATE reply_111 = {id : '111', content : "response"}
CREATE (reply_111)-[:REPLY_TO]->(reply_11)
CREATE (bob)-[:SENT]->(reply_111)
CREATE (reply_111)-[:TO]->(charlie)
CREATE (reply_111)-[:TO]->(davina)
```

This creates the graph in Figure 4-11 where numerous replies and replies-to-replies have been recorded.

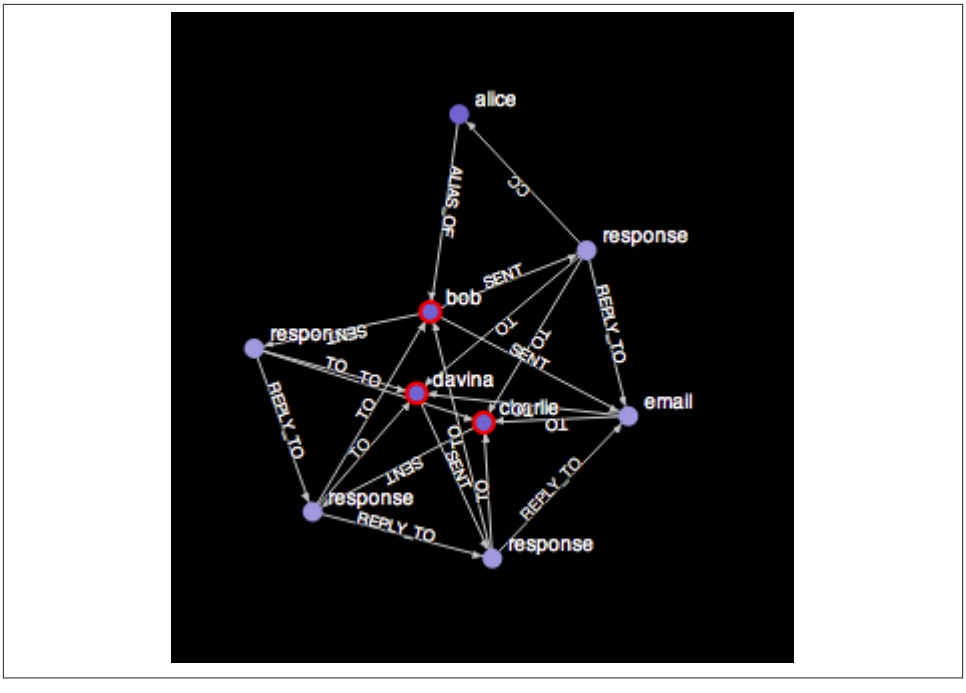


Figure 4-11. Explicitly modeling replies in high-fidelity

Now it is easy to see who replied to Bob's original email. First locate the email of interest, then match against incoming `REPLY_TO` relationships (since there may be multiple replies) and from there match against incoming `SENT` relationships to find the sender(s). In Cypher this is simple to express. In fact, Cypher makes it easy to look for replies to replies to replies, and so on to an arbitrary depth (though we limit it to depth 4 here):

```
START email = node:emails(id, '6789')
MATCH (email)<-[:REPLY_TO*1..4]-()<-[:SENT]-(replier)
RETURN replier
```

It's a similar pattern for a forwarded email: a forwarded email is simply a new email that happens to contain some of the text of the original email (possibly with additions by the forwarder). In that case we should model the new email explicitly, just as in the reply case. But from an application point of view we should also be able to reference the original email from the forwarded mail so that we always have high fidelity provenance data. And the same applies if the forwarded mail is itself forwarded, and so on. For example, if Alice (Bob's alter-ego) emails Bob to try to establish separate concrete identities, and then Bob (wishing to perform some subterfuge) forwards that email onto Charlie who forwards it onto Davina, we actually have three emails to consider. Assuming the users (and their aliases) are already in the database, in Cypher we'd write that audit information into the database as follows:

```
CREATE email = {id : '0', content : "email"}
CREATE (alice)-[:SENT]->(email)-[:TO]->(bob)

CREATE email_101 = {id : '101', content : "email"}
CREATE (email_101)-[:FORWARD_OF]->(email)
CREATE (bob)-[:SENT]->(email_101)-[:TO]->(charlie)

CREATE email_102 = {id : '102', content : "email"}
CREATE (email_102)-[:FORWARD_OF]->(email_101)
CREATE (charlie)-[:SENT]->(email_102)-[:TO]->(davina)
```

On completion of those writes, this particular sub-graph in our database will resemble Figure 4-12, continuously adding new nodes to represent physical domain entities just as in the case where we modeled replies.

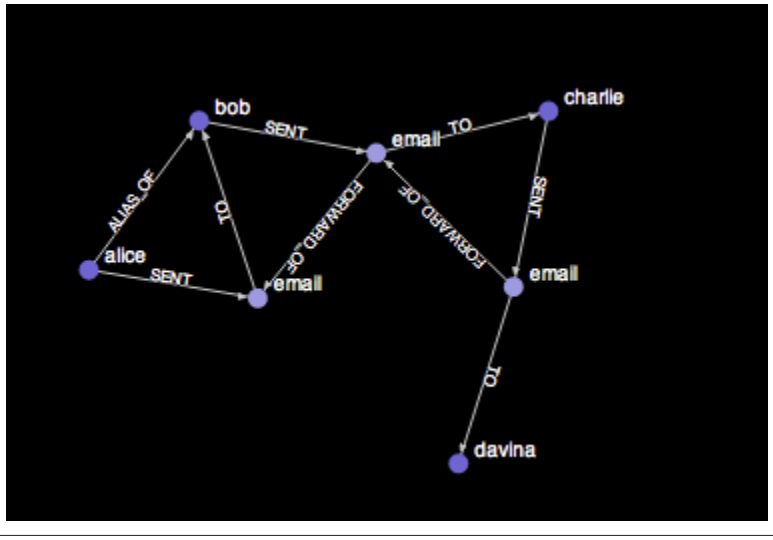


Figure 4-12. Explicitly modeling email forwarding in high-fidelity

Again as we modeled queried replied emails, we can query this graph too and, for example, determine the length of an email chain.

```
START email = node:emails(id, '0')
MATCH (email)<-[f:FORWARD_OF*]-()
RETURN count(f)
```

The above query starts at the given original email and then matches against all incoming `FOWARD_OF` relationships to any depth. In this case those relationships are bound to a variable `f` and we simply return the number of times that relationship has matched by counting the number of times `f` has been bound using Cypher's `count` function. In this example, we see the original email has been forwarded twice.

Avoiding Anti-Patterns

In the general case, don't encode entities (nouns) into relationships. Use relationships to convey semantics about how entities are related, and the quality of those relationships, not to directly encode verbs. And remember that domain entities aren't always clear in the way humans communicate, so think carefully about the nouns that you're actually dealing with.

It's also important to realize that graphs are a naturally additive structure. Pouring in facts in terms of domain entities and how they interrelate is natural, even if it might at first feel like you're flooding the database with a great deal of painstaking data. In general it's a both bad practice and futile to try to conflate data elements at write time to preserve query-time efficiency, so model naturally and in high-fidelity and trust in the graph database to do the right thing.

Good graph databases like Neo4j maintain fast query times even when storing vast amounts of data. Learning to trust your graph database is important when learning to structure (and not denormalize) your graphs.

Summary

Modeling with graphs is one of the most expressive and powerful tools at our disposal as software professionals, providing high affinity between domain and data and removing the need for complex normalization and denormalization and the correspondingly complex data management code. However, modeling with graphs is a new technique for most of us, and so we should take care to ensure that the graphs we're creating read well (for queries) and don't conflate entities and actions (losing useful domain knowledge). While there are no absolute rights or wrongs to graph modeling, by following the guidance in this chapter, we can have confidence that our graph data will serve our systems' needs over many iterations and keep pace with code evolution.

CHAPTER 5

Graph Databases

In this chapter we discuss the characteristics of graph database management systems ("graph databases" for short, or Graph DBMSs), showing how graph databases differ from other means of storing and querying complex, semi-structured, densely-connected data. Throughout this chapter we're going to ground the discussion in the architecture and open source code of Neo4j - the most popular graph database.

Graph databases: a definition

A graph database is a storage engine which supports a graph data model backed by native graph persistence, with access and query methods for the graph primitives to make querying the stored data pleasant and performant. However there are numerous graph data models and while the rest of this book focuses on the most common and pragmatic model — the property graph, or simply *graph database* — there are other kinds of stores which lay claim to the name "graph database" including triple stores and hypergraph stores.

Hypergraphs

Hypergraphs are a generalized graph model where a relationship (called a hyper-edge) can connect any number of nodes. More specifically while the property graph model permits a relationship have a single start and end node, the hypergraph model allows any number of nodes at a relationship's start and end. For some problems hypergraphs could be a useful model. For example in Figure 5-1 we see that Alice and Bob are the owners of three vehicles which is expressed through a single hyper-edge (rather than potentially six relationships in a property graph).

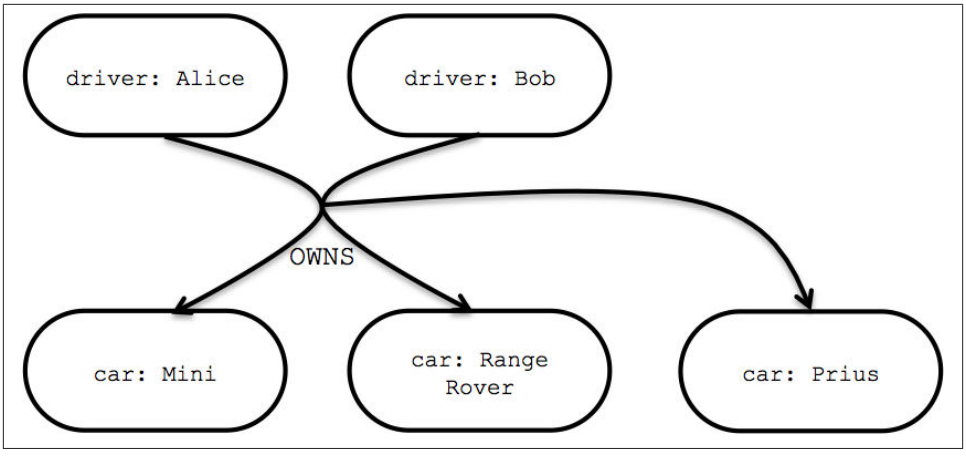


Figure 5-1. A simple (directed) hypergraph

However we saw in the anti-patterns discussion Chapter 4 how high-fidelity is important to graph modeling. While in theory hypergraphs are high-fidelity, in practice it's very easy to produce lossy models with them. To illustrate this point, let's consider the property graph in Figure 5-2.

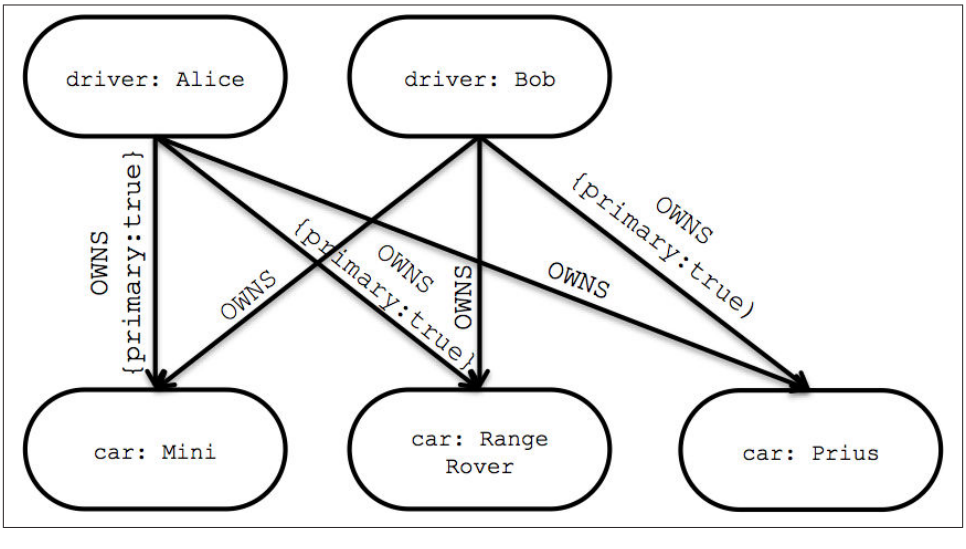


Figure 5-2. A property graph is semantically fine-tuned

The Figure 5-2 property graph contains several `OWNS` relationships to express the same information that required only a single relationship in Figure 5-1. Yet in using several relationships, not only are we able to use a familiar and very explicit modeling technique,

but we're also able to fine-tune that model. For example we've taken the opportunity to distinguish the "primary driver" for each vehicle for insurance purposes by adorning the appropriate relationships, something which can't be done with a single hyper-edge.

It's a fair to say that hypergraphs are a more general model than property graphs since hyper-edges are multidimensional. However it is possible to represent the information of any hypergraph as a property graph (albeit using more relationships and intermediary nodes) because the two models are isomorphic. Whether a hypergraph or a property graph is best for you is going to depend on your modeling mindset and the kinds of applications you're building. Anecdotally, for most purposes property graphs seem to have the best balance of pragmatism and modeling efficiency, hence their overwhelming popularity in the graph database space. However for situations where meta-intent is desired (e.g. I like the fact that you liked that car) then hypergraphs require fewer primitives than property graphs.

Triples

The triple-store family of databases comes from the Semantic Web [1] movement, where researchers are interested in large-scale knowledge inference by adding semantic markup on links between Web resources. To-date there's relatively little of the Web which is usefully marked-up so running queries across the semantic layer is uncommon. Instead most effort in the semantic Web appears to be invested in harvesting useful data and relationship information from the Web (or other more mundane data sources like applications) and depositing it in *triple stores* for querying.

Triple stores are a specialized kind of database into which harvested *triples* are deposited. Triple typically provide SPARQL [2] capabilities to reason about stored RDF [3] data. In turn, the stored triples of the form *subject-predicate-object* provide a set of facts such that we can easily capture knowledge via numerous pithy statements like "Ginger dances with Fred" and "Fred likes ice cream." Individually a single triple is semantically rather poor, but en-masse they provide a rich data set from which to harvest knowledge and *infer* connections.

While RDF — the lingua franca of triple stores and the Semantic Web — can be serialized several ways, using the RDF/XML format we can begin to see how triples come together to form linked data in RDF encoding of a simple three-node graph.

1. *http://www.w3.org/standards/semanticweb/*
2. *http://www.w3.org/TR/rdf-sparql-query/*
3. *http://www.w3.org/RDF/*

RDF encoding of a simple three-node graph.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns="http://www.example.org/ter
        <rdf:Description rdf:about="http://www.example.org/ginger">
                <name>Ginger Rogers</name>
                <occupation>dancer</occupation>
                <partner rdf:resource="http://www.example.org/fred"/>
        </rdf:Description>
        <rdf:Description rdf:about="http://www.example.org/fred">
                <name>Fred Astaire</name>
                <occupation>dancer</occupation>
                <likes rdf:resource="http://www.example.org/ice-cream"/>
        </rdf:Description>
</rdf:RDF>
```

W3C support

The logical representation of the triple store doesn't imply any triple-like implementation internally for a store. However the various triple stores are almost all unified by their support for Semantic Web technology like RDF and SPARQL. While there's nothing particularly special about RDF as as means of serializing linked data from a format perspective, it is endorsed by the W3C and therefore benefits from being widely understood and well documented. The query language SPARQL benefits from similar W3C patronage.

In the graph database space there is a similar raft of innovation around graph serialization formats (e.g. GEOFF [4]) and inferencing query languages (e.g. the Cypher query language [5] that we use throughout this book). The key difference is that such innovation does not enjoy the patronage of a well-regarded body like the W3C, though they do benefit from strong engagement within their user and vendor communities.

However triple stores are not graph databases, though they ostensibly deal in data which — once processed — tends to be logically linked. Native storage of triples all but precludes the kind of rapid traversal of relationships that are the bedrock of graph databases, graph algorithms and graph-oriented programming. Instead a strength of the triple model is that it is easy to scale horizontally for storage since triples are independent artefacts. Yet native triple storage all but implies latent at query time for the same reason — the linked data has to be reified from independent facts into a connected structure. Given those assertions, the sweet spot for a triple store is generally not OLTP (that is for responsive online transaction processing systems), instead triples may be of use for analytics where latency is a secondary consideration.

4. *http://geoff.nigelsmall.net/*
5. *http://docs.neo4j.org/chunked/milestone/cypher-query-lang.html*

Though it's clear that graph databases emphasize traversal performance and the ability to execute graph algorithms, it is possible to use a graph database as a backing store behind a RDF/SPARQL endpoint. For example the Blueprints SAIL [6]API provides an RDF interface to several graph databases, including Neo4j. In practice this implies a level of functional isomorphism between graph databases and triple stores. However each store type is suited to different kinds of workload, with graph databases being optimized for graph workloads and especially rapid traversals.

Graph-Native Stores

We've discussed the property graph model several times throughout this book, and by now we're familiar with the notion of nodes containing properties connected by named, directed relationships which themselves can contain properties. However there a numerous ways to encode and store a property graph, some of which take advantage of their natural *index-free adjacency* model which is which is crucial for fast, efficient graph traversals.

The term index-free adjacency [7] has been advanced as a means of reasoning about non-graph native databases from more general-purpose graph databases. In a non-native store (global) indexes are used to link together nodes, where as in a native graph store each node acts as a kind of micro-index of other nearby nodes (at low computational cost). In a native property graph, there is no requirement to maintain large overarching indexes to describe connectivity (at greater computational cost).

Using indexes to discover adjacent nodes is a more expensive operation case than using the relationships incident on a node. Depending on the implementation, index lookups are typically $O(\log n)$ in algorithmic complexity versus $O(1)$ for looking up relationships. To traverse a network, the cost of an indexed approach at $O(m \log n)$ dwarfs the cost of $O(m)$ for an index-free adjacency approach for a traversal of m steps.

6. *https://github.com/tinkerpop/blueprints/wiki/Sail-Implementation*
7. The main citation for the term is at *http://en.wikipedia.org/wiki/Graph_database*, but it is believed to have originated from Marko Rodriguez, a well-known graph advocate.

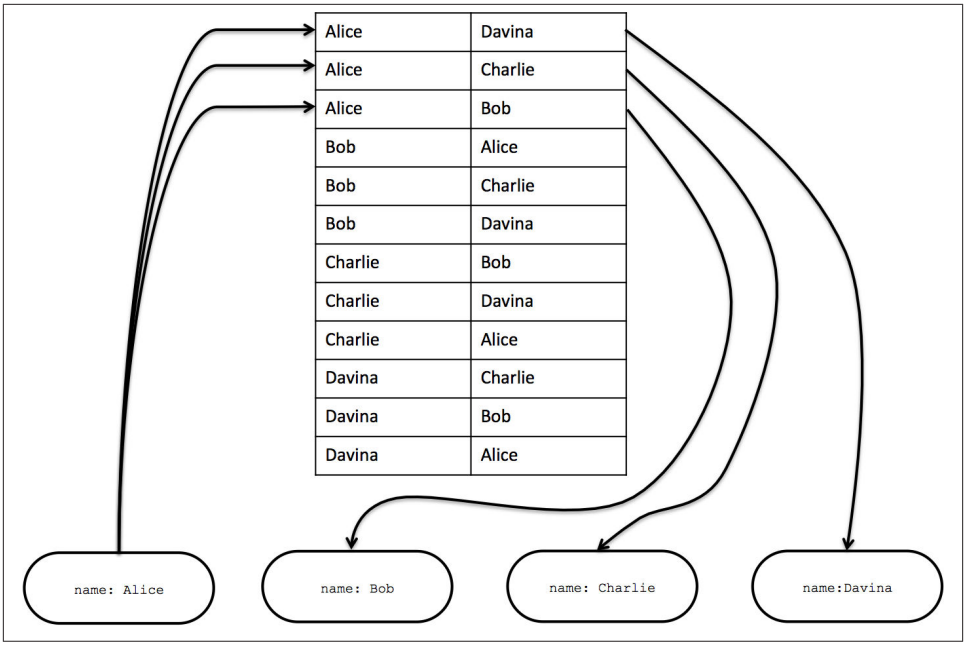


Figure 5-3. Non-native graph stores use indexing to traverse between nodes

For example in Figure 5-3 to find Alice's friends costs an index lookup at cost *O(log n)* which may be an acceptable cost for occasional lookups. But to find who is friends with Alice, the costs are far more onerous. For example, it's *O(log n)* cost to find out who Alice's friends are, but *O(m log n)* to find out who is friends with Alice as we have to interrogate the index for each node that could potentially be a friend with Alice.

Index-free adjacency leads to lower-cost "joins"

When using indexes to fake connections between records, there is no *actual* relationship stored in the database. This is problematic when we try to traverse in the "opposite" direction compared to how the index was constructed. Since we have to brute force search through the index — an O(n) operation — joins like this are simply too costly to be of any practical use.

While the cost of index lookups might not be considered too onerous for very small networks like Figure 5-3 or the kind of example often seen in demos, in the general case it is too costly to be practical since it directly limits the scope and richness of the result set for a responsive application. It is for that reason that performant graph databases like Neo4j tend towards index-free adjacency to ensure high performance traversals as an essential design goal.

Recall that in a general-purpose graph database relationships can be traversed in either direction (tail to head, or head to tail) extremely cheaply. The graph in Figure 5-4 helps illustrate this point in contrast to the far more expensive index-bound approach.

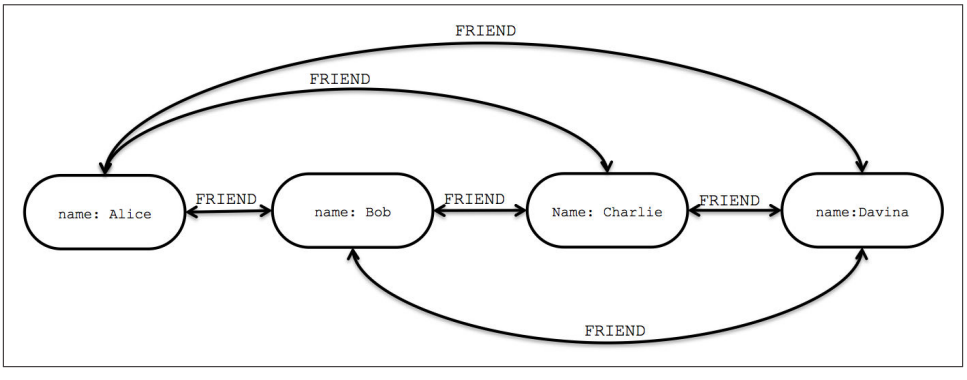


Figure 5-4. Neo4j uses relationships, not indexes for fast traversals

As we can see in Figure 5-4, to discover who Alice's friends are in a graph, we simply follow her outgoing `FRIEND` relationships at *O(1)* cost each. For who is friends with Alice, we simply follow all her incoming `FRIEND` relationships to their source at *O(1)* cost each once again. This observation should lead to very efficient graph traversals in theory, but such high-performance traversals only become reality when they are supported by an architecture designed for that purpose as we shall now discuss.

Architecture

Given that index-free adjacency is the key to high performance traversals and thus high performance queries and writes, one key aspect of the design of a graph database is the way in which graphs are stored. An efficient, native graph storage format supports extremely rapid traversals for arbitrary graph algorithms — an important reason for using graphs — and is a key differentiator between a native graph database and other non-native graph databases (e.g. triple stores).

For illustrative purposes we'll use the Neo4j database as a canonical example of how a graph database is architected. We choose Neo4j for a variety of reasons — it's very popular, it's graph-native, and it's open source so our discussion here can be readily tied back to corresponding code.

Firstly let's contextualize our discussion by looking at Neo4j's high level architecture, presented in Figure 5-5. We'll work bottom-up from the files on disk through to the programmatic APIs and up to the Cypher query language. Along the way we'll discuss the performance and dependability characteristics and the design decisions that enable Neo4j to be a performant, reliable graph database.

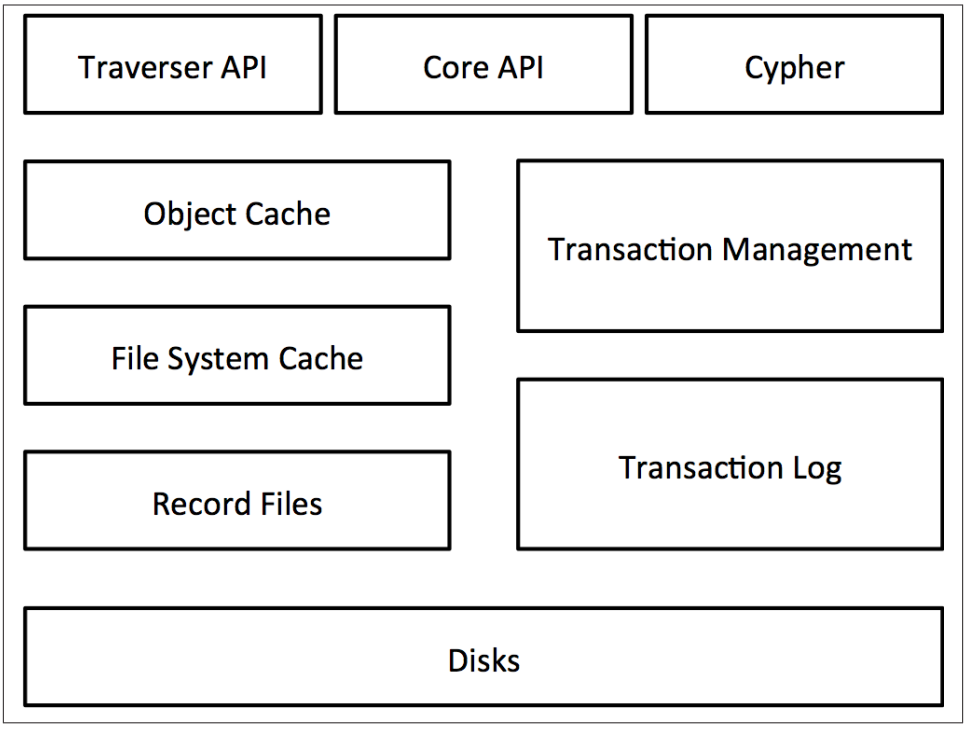


Figure 5-5. Neo4j architecture

Neo4j stores graph data spread across a number of *store files*. Each store file contains the data for a specific part of the graph (e.g. nodes, relationships, properties). The division of storage responsibilities — particularly the separation of graph structure from property data — exists for performant graph traversals even if though it means the user's view of their graph and the actual records on disk are structurally dissimilar. Let's start our exploration of physical storage by looking at the structure of nodes and relationships on disk as shown in Figure 5-6.

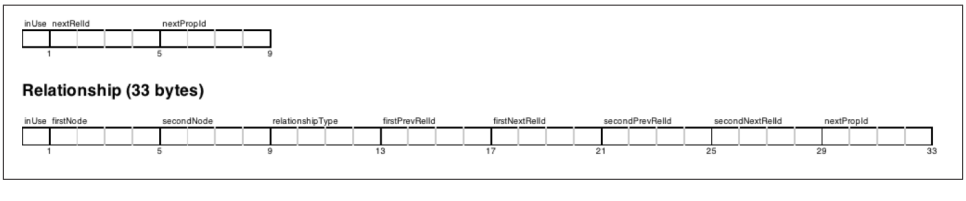


Figure 5-6. Neo4j Node and Relationship Store File Record Structure

The node store file is where, unsurprisingly, node records are stored. Every node created in the user-level graph ends up in this node store, and is stored in the physical file

`neostore.nodestore.db`. Like most of the Neo4j store files, the node store is a fixed-size record store where each record is 9 bytes in length. Fixed record sizes enable fast lookups for nodes within the store file — if we have a node with id `100` then we know its record begins 900 bytes into the file and so the database can directly compute the record location at cost *O(1)* without performing a search at cost *O(log n)*.

Inside a node record, the first byte is the in-use flag which tells the database whether the record is currently used or can be reclaimed to store new records (with Neo4j's `.id` files being used to keep track of unused records). The next four bytes are the of the first relationship connected to the node. The last four nodes bytes are the id of the first property for the node. The node record is pretty lightweight, it's really just a couple of pointers to lists of relationships and properties.

Correspondingly, relationships are stored in the relationship store file 'neostore.relationshipstore.db`. Like the node store file, the relationship store contains fixed-size (in this case 33 bytes) records and each record contains the id of the nodes at the start and end of each relationship, a pointed to the relationship type (stored in the relationship type store) and pointers for the relationship chains for the start and end node (as a doubly linked list) since a relationships logically belongs to both nodes and therefore should appear in the list of both nodes' relationships.

Both the node and relationship stores are only concerned with the structure of the graph, not the property data it holds. Furthermore, both node and relationship stores use fixed-sized records such that any individual record's location within the store file can be rapidly computed given its id. These are critical design decisions that underlines Neo4j's commitment to high performance traversals.

In Figure 5-7 we see how the various store files interact on disk. The node records contain only a pointer to their first property and their first relationship (in what is often termed the _relationship chain). From here, we can follow the (doubly) linked-list of relationships until we find the one we're interested in, the `LIKES` relationship from `Node 1` to `Node 2` in this case. Once we've found the relationship record of interest, we can simply read its properties if there are any via the same singly-linked list structure as node properties, or we can examine the node records that it relates via its start node and end node IDs. These IDs, multiplied by the node record size, of course give the immediate offset of both nodes in the node store file.

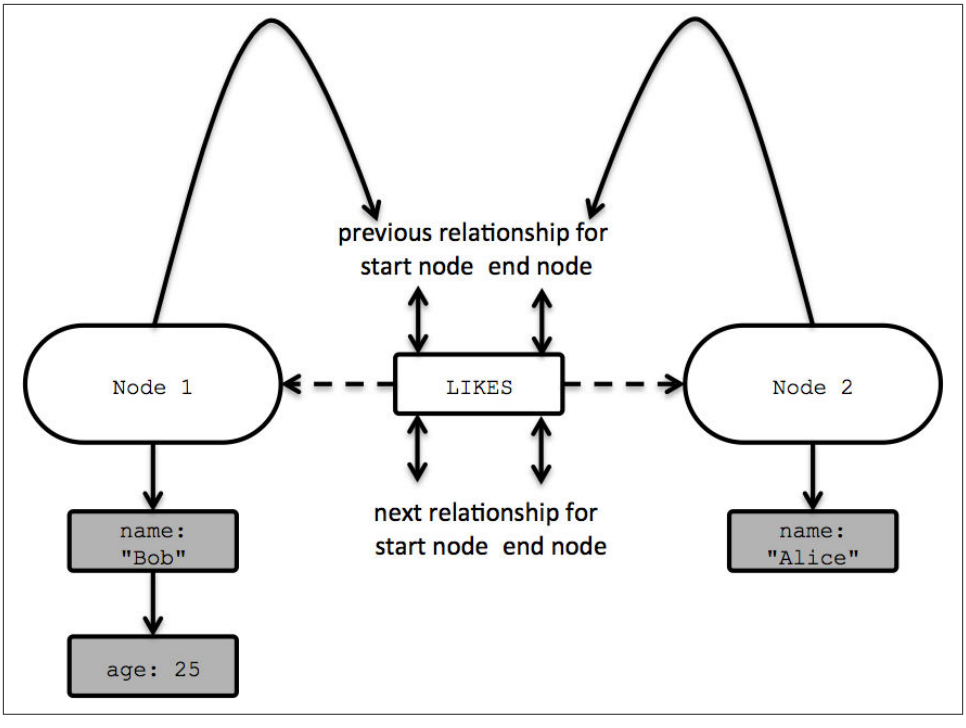


Figure 5-7. How a graph is physically stored in Neo4j

Doubly linked lists in the relationship store

Don't worry if the relationship store structure seems a little complex at first; it's definitely not as simple as the node store or property store.

It's helpful to think of a relationship record as "belonging" to two nodes—the start node and the end node of the relationship. Clearly, however, we don't want to store two relationship records, because that would be wasteful. And yet it's equally clear that the relationship record should somehow belong to both the start node and the end node.

That's why there are pointers (aka record ids) for two doubly-linked lists: one is the list of relationships visible from the start node; the other is the list of relationships visible from the end node. That each list is doubly-linked simply allows us to rapidly iterate through that list in either direction, and insert and delete relationships efficiently.

Choosing to follow a different relationship involves iterating through a linked list of relationships until we find a good candidate (e.g. matching the correct type, or having some matching property value). Once we have a suitable relationship we're back in business simply multiplying id by record size, and thereafter chasing pointers.

With fixed sized records and pointer-like record ids, traversals are implemented simply by ``pointer chasing'' around a data structure which can be performed at very high speed. To traverse a particular relationship from one node to another the database performs several cheap id computations [8]:

1. From a given node record, locate the first record in the relationship chain by computing its offset into the relationship store by multiplying its id by the fixed relationship record size (remember that's 33 bytes at time of writing). This gets us into right record in the relationship store.
2. From the relationship record look at the *second node* field for the id of the second node. Multiply that id by the node record size (9 bytes at time of writing) to locate the correct node record in the store.

Should we wish to constrain the traversal to relationships with particular types, then we'd add a lookup in the relationship type store. Again this is a simple multiplication of id by record size (5 bytes at time of writing) to find the offset for the appropriate relationship type record in the relationship store.

In addition to the node and relationship stores which contain the graph structure, we have the property store files that store the user's key-value pairs. Recall that Neo4j, as a property graph database, allows properties — name-value pairs — to be stored in both nodes and relationships and so the property store is referenced from both node and relationship records.

Records in the property store are physically stored in the `neostore.propertystore.db` file, and in that file each (fixed-size) entry contains fields representing the property type [9], a pointer into the property index store file (`neostore.propertystore.db.index`), a pointer to a DynamicStore record (being either in the DynamicStringStore in `neostore.propertystore.db.strings` or the DynamicArrayStore in `neostore.propertystore.db.arrays` depending on the property type) and finally the id of the next property in the chain [10].

Inlining and optimizing property store utilization

Neo4j supports store optimizations where some properties can be inlined into the property store file directly (`neostore.propertystore.db`). This can happen where property

8. in fact these computations are much cheaper than searching global indexes as we'd do if faking a graph in a non-graph native database
9. Neo4j allows any primitive JVM type, plus strings and arrays of all the previous.
10. Recall that properties are held as a singly linked-list on disk compared to the doubly linked-list for relationships.

data can be encoded to fit within the size normally allocated for a pointer to a record (minus a little overhead for descriptive metadata). In practice this means that data like phone numbers and zip codes can be inlined in the property store file directly rather than being pushed out to the dynamic stores. This results in reduced I/O operations and improved throughput since only a single file access is required.

In addition to inlining certain compatible property values, Neo4j also maintains space discipline on property names. For example in a social graph, there will likely be many nodes with properties like `first_name` and `last_name`. It would be wasteful if each property name was written out to disk verbatim, and so instead property names are indirectly referenced from the property store through the property index file. The property index allows all properties with the same name to share a single record, and thus for repetitive graphs — a very common use case — a great space, and subsequently I/O, saving is achieved.

Having an efficient storage layout is only half of the picture. Even though the store files are optimized for rapid traversals, we live in a world where disk storage vastly outsizes main memory capacity. However disks have seek times in single digit milliseconds for spinning disks which — although fast by human standards — are ponderously slow in computing terms. Though Solid State Disks (SSDs) are far better (since there's no significant seek penalty waiting for platters to rotate) the path between CPU and disk is still more latent than the path to L2 cache or main memory where we'd ideally like to operate on our graph.

However we can use in-memory caching to mask the lower performance characteristics of mechanical/electronic mass storage devices and probabilistically provide low latency access to the graph. In Neo4j there is a two-tiered caching architecture which provides this functionality.

The lowest tier in the Neo4j cache stack is the *filesystem cache*. This cache is page-affined, dividing the store into discrete regions with the cache holding a fixed number of those regions per store file. The actual amount of memory to be used to cache each store file's pages can be fine-tuned, though Neo4j will use sensible default values based on the capacity of the underlying hardware in the absence of input from the user [11].

Pages are evicted based on a least frequently used policy, with the default implementation delegating responsibility for to underlying operating system's memory mapped file system such that the operating system itself determines which segments of virtual memory will be in real memory and which will be flushed out to disk. The filesystem cache is particularly beneficial where related parts of the graph are written together so that they occupy the same page. This is a common pattern for writes since we tend to persist whole subgraphs to disk rather than random nodes and relationships.

11. See: *http://docs.neo4j.org/chunked/stable/configuration-caches.html* for tuning options

If the filesystem cache reflects the write characteristics of typical usage, then the *high level* or *object* cache is all about optimizing for arbitrary read patterns. The high-level cache stores nodes, relationships, and properties for rapid, in-memory graph traversal. Unlike the store files where the node records are light compared to the relationship records, in the high level cache nodes are richer and much closer to what we expect as programmers since they hold properties and references to their relationships. Conversely the relationship objects refer only to their properties. Relationships for each node are grouped by RelationshipType and direction for fast lookup of specific relationships based on those characteristics. As with the underlying stores, all lookups are by id meaning they are very performant as we can see in Figure 5-8.

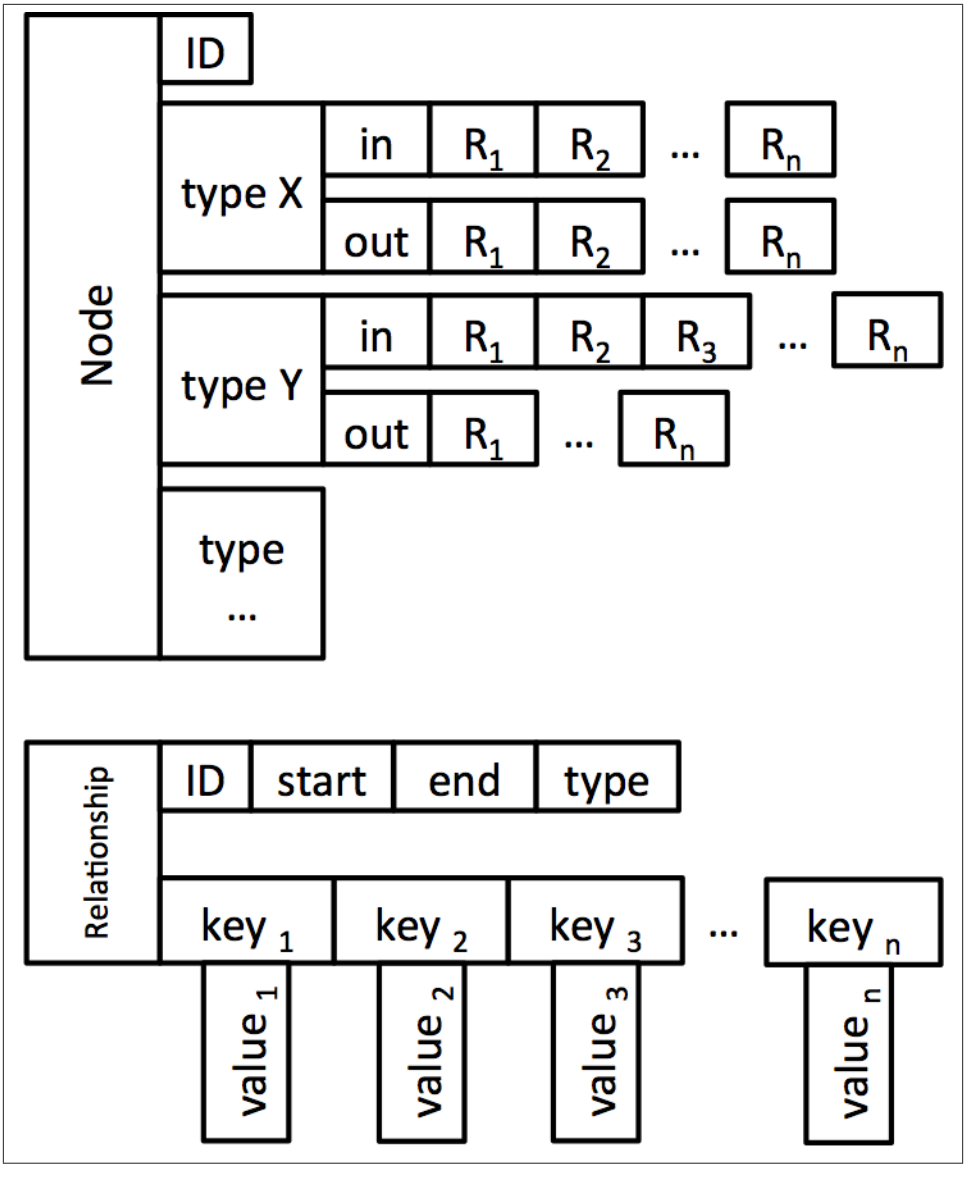


Figure 5-8. Nodes and relationships in the object cache

Neo4j Programmatic APIs

While the filesystem and caching infrastructure is fascinating, it is all for nothing without the APIs to drive it. Though we've focussed on the high-level query language *Cypher* for much of this book, Neo4j also offers programmatic APIs, which Cypher uses,

but which are also available to developers. Logically it's easy to think of these APIs in a stack where at the top of the stack we prize expressiveness and declarative programming, while at the bottom of the stack we prize precision and imperative style, as depicted in Figure 5-9.

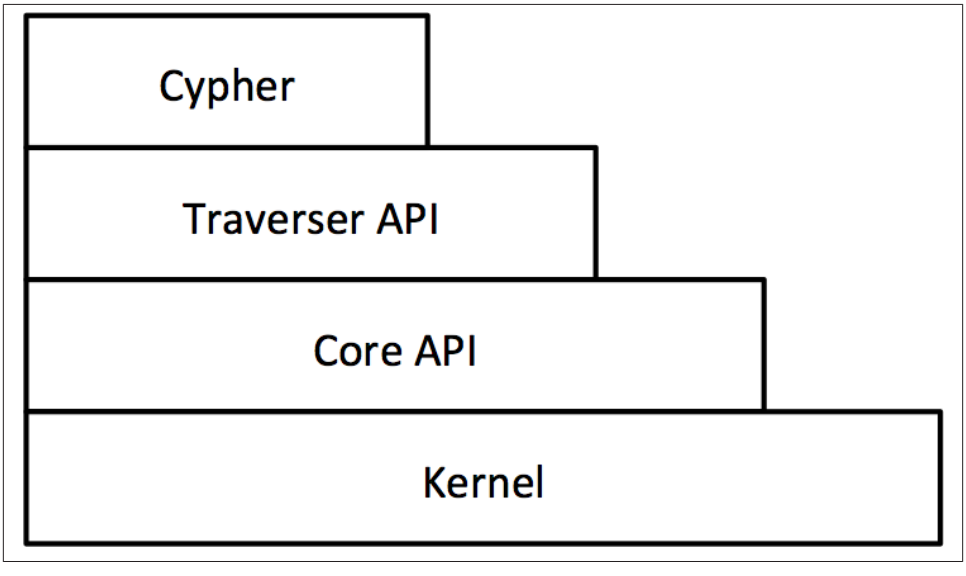


Figure 5-9. Logical view of the user-facing APIs in Neo4j

We'll step through these from the bottom up in the following sections.

Kernel API

Though it's not too often the focus of development, the lowest level API that regular Neo4j users access are transaction event handlers [12] in the kernel. These allow user code to listen to transactions as they flow through the kernel and react (or not) based on the data content and lifecycle stage of each transaction by user code.

Kernel transaction event handlers

A typical use-case for transaction event handlers is to prevent physical deletion of records. A handler can be set up to intercept deletion of a node and instead simply mark

12. See: *http://docs.neo4j.org/chunked/stable/transactions-events.html*

that node as deleted logically (or in a more sophisticated manner move the node "back in time`` by creating timestamped archive relationships).

Core (or "Beans") API

Neo4j's core API is a popular imperative Java API which exposes the graph primitives of node, relationship, and properties to the user. As a rule, it's a lazily evaluated API for reads which means a relationships are only traversed as quickly as the API caller can consume the data being returned. For writes, the Core API provides transaction management capabilities to ensure atomic, consistent, isolated, and durable persistence.

In the following code, we see a snippet of code borrowed from the Neo4j tutorial in which we try to find human companions from the Doctor Who universe.[13]

```
// Index lookup for the node representing the doctor is omitted for brevity

Iterable<Relationship> relationships = doctor.getRelationships( Direction.INCOMING,
                                                                COMPANION_OF );

for ( Relationship rel : relationships )
{
    Node companionNode = rel.getStartNode();
    if ( companionNode.hasRelationship( Direction.OUTGOING, IS_A ) )
    {
        Relationship singleRelationship = companionNode
                                            .getSingleRelationship( IS_A,
                                                    Direction.OUTGOING );
        Node endNode = singleRelationship.getEndNode();
        if ( endNode.equals( human ) )
        {
            // Found one!
        }
    }
}
```

The specific domain in the code in this snippet isn't too important, but it's safe to say the code is very imperative (and somewhat verbose compared the the equivalent Cypher query). We simply loop round the Doctor's companions and check to see if any of the companion nodes have an `IS_A` relationship to the node representing the human species. If so we do something with it.

As an imperative API, the Core API can be fine-tuned to the underlying graph structure. However compared to the higher-level APIs (and particularly Cypher) more code is needed to achieve an equivalent goal. Nonetheless the affinity between the Core API

13. See: *https://github.com/jimwebber/neo4j-tutorial*. Doctor Who is the world's longest-running science-fiction show and a firm favorite of the Neo4j team.

and the underlying record store is plain to see — with the structures used at the store and cache level being represented relatively faithfully to user code.

Neo4j Traversal API

The traversal API is a declarative Java API whereby the user provides a set of constraints about which parts of the graph the traversal is allowed to visit. These constraints are expressed in terms of relationship type, direction and breadth versus depth-first search, coupled with a user-defined path evaluator which is triggered on each node encountered. In the following code snippet we see an example of the traverser API in action.

```
Traversal.description()
  .relationships( DoctorWhoRelationships.PLAYED, Direction.INCOMING )
  .breadthFirst()
  .evaluator( new Evaluator()
  {
    public Evaluation evaluate( Path path )
    {
      if ( path.endNode().hasRelationship( DoctorWhoRelationships.REGENERATED_TO ) )
      {
        return Evaluation.INCLUDE_AND_CONTINUE;
      }
      else
      {
        return Evaluation.EXCLUDE_AND_CONTINUE;
      }
    }
  } );
```

The declarative nature of the traversal API is plain to see in the snippet, where we declare that only `PLAYED` relationships in the `INCOMING` direction may be traversed, and that we traverse the graph in a breadth-first manner (visiting all nearest neighbors before going further outwards).

Even though the traversal API is declarative for graph structure, we still rely on the imperative Core API both to support within each traversal to determine whether further hops through the graph are necessary or to modify the graph. In the previous code snippet this is the function of the `Evaluator` which, given a path, allows user code to determine an action on the graph and whether to permit a further traversal. Again the native graph structures inside the database bubble close to the surface here, with the graph primitives of node, relationship and property taking center stage in the API.

The traversal API isn't just for user code, the current Cypher implementation uses the traverser framework too!

We've now seen the graph-centric APIs, and how they are sympathetic to the structures used in the lower levels of the Neo4j stack to support rapid graph traversals. Accordingly, we can now move begin to discuss why we can *depend* on Neo4j to store our data reliably, and not just idiomatically and rapidly.

Non-Functional Characteristics

At this point we've understood a great deal about how Neo4j is designed and what it means to construct a native graph database. But to be considered *dependable* any data storage technology must provide some level of guarantee as to the durability and accessibility of the stored data.[14]

Relational databases are traditionlly evaluated on the number of transactions per second they can process. It is assumed that these transactions uphold the ACID properties (even in the presence of failures) such that data is consistent and recoverable. For non-stop processing and managing large volumes, a database is expected to scale so that many instances are available to process queries and updates, with the loss of an individual instance not unduly affecting the running of the cluster as a whole.

At a high level at least, much the same applies to graph databases: they need to guarantee consistency, and to recover gracefully from crashes and prevent data corruption; further, they need to scale out to provide high availability, and scale up for performance. In the following sections we'll explore what each of these means to a graph database and show why Neo4j's architecture means it can be considered dependable from a systems engineering point of view.

Transactions

Transactions have been a bedrock of dependable computing systems for decades, and though many NOSQL stores aren't transactional [15] they remain a fundamental abstraction for dependability in contemporary graph databases (including Neo4j).

Transactions in Neo4j are semantically identical to traditional database transactions. Writes occur within a transaction context, taking write locks (for consistency) on any necessary nodes and relationships, and on successful completion of the transaction only (for atomicity) the changes made to the graph and/or index is flushed to disk (for durability) and the write locks released. Should the transaction fail for some reason then

14. The formal definition of dependability is the "trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers" as per *http://www.dependability.org/*

15. Because there's an unvalidated assumption that transactional systems scale less well. There is some truth to this, insofar as (distributed) two-phase commit can to exhibit unavailability problems in pathological cases, but for the most part transactions don't limit scalability.

the writes will be discarded and the write locks released maintaining the graph in its previous consistent state.

Should two or more transactions attempt to mutate the same data concurrently, Neo4j will detect that potential deadlock and serialize the transactions. Furthermore writes within a single transactional context will not be visible to other transactions (for isolation).

Transactions in Neo4j

In Neo4j the transaction implementation is conceptually straightforward. Each transaction is represented as an in-memory object whose state represents writes to the database. This object is supported by a lock manager that applies write locks to nodes and relationships as they are created, updated, and deleted. On transaction rollback, the transaction object is simply discarded and the write locks released, whereas on successful completion the transaction is committed to disk.

Committing data to disk in Neo4j uses a *Write Ahead Log* approach whereby changes are appended as actionable entries in the active transaction log. On transaction *commit* (assuming a positive response to the *prepare* phase) a commit entry will be written to the log, causing it to be flushed to disk, thereby making the changes durable. Once the disk flush has occurred the changes are applied to the graph itself. After all the changes have been applied to the graph, any write locks associated with the transaction are released.

One of the advantages of transactional behavior from a database system is that once the transaction has committed, the system is in a state where changes are guaranteed to be in the database even if a fault causes a (non-pathological) failure. This, as we shall now see, confers substantial advantages for recoverability and hence ongoing provision of service.

Recoverability

Databases are no different from any other software in that they are susceptible to bugs in their implementation [16], in the hardware they run on and in the power and cooling infrastructure that hardware needs. Though diligent engineers try to minimize the possibility of failure in all of these, at some point it's inevitable the database will crash (though the mean time between failures should be very long indeed).

In a well designed system, a database server crash is an annoyance but does not impinge availability (though it may affect throughput). However when that failed server resumes

16. Even Neo4j cannot be provably bug-free.

operation it must not source corrupt data to its users irrespective of how inconvenient or problematic the nature or timing of the crash.

In case of an unclean shutdown (perhaps caused by a fault or even an overzealous operator), Neo4j checks in the most recently active transaction log and replays any transactions it finds against the store. It's possible that some of those transactions *may* have already been applied to the store, but since replaying is an idempotent action, the net result is the same: that the store is consistent with all successfully committed transactions prior to the failure when recovery completes.

Of course, recovering locally is fine if all we have is a single database instance, but generally we run databases in clusters (which we'll discuss next) to assure high availability to applications. Fortunately clustering confers an additional benefit to the recovering database that, not only will it be consistent with respect to the time of failure when it recovers, but it can quickly catch up to be consistent with other instances in the cluster. That is, once local recovery has completed, then a replica can ask other members of the cluster (typically the master) for any newer transactions that it has missed which it then applies to its own data set via transaction replay.

Recoverability deals with the capability of the database to set things right after a fault has arisen. Even so, a good database still needs to be highly available to meet the increasingly sophisticated needs of data-heavy applications.

Availability

Since Neo4j is both transactional and recoverable, it bodes well for availability since the ability to recognize and (if necessary) repair an instance after crashing means that data quickly becomes available again without human intervention. And of course, more live instances increases the overall availability of the database to process queries.

However it's uncommon that we want individual disconnected database instances in a typical production scenario. In those cases databases (of all kinds) tend to be clustered for high availability. In Neo4j's case a master-slave cluster arrangement is used so that a complete replica of the graph is stored on each machine [17].

For writes, the classic write-master with read-slaves is a popular topology whereby all database writes are directed at the master, while read operations are directed at slaves. This provides asymptotic scalability for writes (up to the capacity of a single spindle) but allows for near linear scalability for reads (accounting for the modest overhead in managing the cluster).

17. This is a simplification, in reality the master and some slaves have a completely up to date copy of the graph, while other slaves will be catching up typically being milliseconds behind.

Although write-master with read-slaves is a classic deployment topology, Neo4j also supports writing through slaves. In this scenario, the slave first ensures that it is consistent with the master (it is said to "catch up") and then the write is synchronously transacted across both instances. This is useful where we want immediate durability in two database instances but at the cost of higher write latency owing to the forced catchup phase, and offers additional deployment flexibility. However it does *not* imply that writes are distributed around the system, since all writes must still pass through the master at some point.

Other replication options in Neo4j

In Neo4j version 1.9 onwards it's also possible to specify an arbitrary number of replicas to be written before a transaction is considered complete in a best-effort manner. This an be combined with writing through slaves to provide "at least two" level of durability.

Another aspect of availability is contention for access to resources. A query that contends for exclusive access (e.g. for writes) to a particular part of the graph may suffer from high enough latency to appear unavailable. We've seen similar contention with coarse-grained table-level locking in RDBMSs where writes are latent even when there's logically no contention.

Fortunately in a graph, access patterns tend to be more evenly spread, especially where idiomatic graph-local queries are executed. A graph local operation is one that starts at one or more given places in the graph and traverses the surrounding subgraphs from there. The starting points for such queries tend to be things that are especially significant in the domain, such as users or products, and so the overall query load tends to be distributed with low contention. In turn the perception is one of greater responsiveness and higher availability.

The benefits of idiomatic queries

Jackie Stewart, the Formula 1 racing driver, is reputed to have said that to drive a car well you don't need to be an engineer but you do need mechanical sympathy with your car. That is, the best performance comes as a result of the driver and car working together harmoniously.

In much the same way, Neo4j queries are mechanically sympathetic to the database when they are framed as idiomatic, graph local queries that begin their traversal from one or more start points. The underlying infrastructure, including caching and store access, is optimized to support this kind of workload.

Being idiomatic has beneficial side effects. For example, because caching is aligned with idiomatic searches, queries that are themselves idiomatic tend to exploit caches better

and run faster than non-idiomatic queries. In turn, faster running queries free up the database to run more of them, which means higher throughput and the sense of better availability from the client's point of view because there's less waiting around.

Unidiomatic queries (e.g. those which pick random nodes/relationships rather than traversing) exhibit the opposite characteristics: they disrespect the underlying caching layers and therefore run more slowly as more disk I/O is needed. Since the queries run slowly, the database can process fewer of them per second, which means the availability of the database to do useful work diminishes from the client's point of view.

Therefore in Neo4j, we should always strive for idiomatic operations so that we are mechanically sympathetic to the database and so maximise its performance.

Our final observation on availability is that scaling for cluster-wide replication has a positive impact not just in terms of fault-tolerance, but also in terms of responsiveness. Since there are many machines available for a given workload, query latency is low and availability is maintained. But as we'll now discuss, scale itself is more nuanced than simply the number of servers we deploy.

Scale

The topic of scale has become more important now that we're in an era where we are aware of enormous and ever-increasing data volumes that surround us. In fact, a substantial motivation for the NOSQL movement has been to address the problems of data at scale which have proven difficult to solve with relational databases. In some sense, graph databases are no different since they also need to scale to meet the workload demands of modern applications. But scale isn't a simple value like transactions per second, it's an aggregate value that we measure across multiple axes.

For graph databases, we will decompose our broad discussion on scale into three key themes: * Graph size * Response time * Read and write throughput

Starting with Graph size, Neo4j can store tens of billions of nodes, relationships, and properties in a single (non-distributed) graph. Specifically the upper bound for the current (at time of writing) release of Neo4j (1.9) is as follows: * Number of nodes: 34 billion * Number of relationships: 34 billion * Number of properties: 68 billion

These (relatively generous) upper bounds result from a technical drive to balance efficiency versus scale. One reason Neo4j is so fast for graph traversals is that it makes extensive use of fixed record sizes behind the scenes as we have discussed. Each record includes a number of pointers: the larger the pointers, the larger the fixed record sizes,

and the more capacious the graph can be. However as these pointers increase in size, memory and disk requirements increase [18].

At time of writing, the Neo4j team has publicly expressed its intention to support 100B+ nodes/relationships/properties in a forthcoming release. At some point the fixed limit will be so large as to be logically infinite, but retain the benefits of fixed-sized records and pointers for high-speed traversals.

In terms of latency, graph databases don't suffer the same problems as traditional RDBMS where the more data you have in in tables — and in indexes — the longer join operations take [19]. Instead with Neo4j, most queries follow a pattern where an index is used simply to find a starting node (or nodes), and the remainder of the traversal uses a combination of pointer chasing and pattern matching to search the datastore. Where this is very different from a relational database is that performance does not depend on the total size of the data set, but only on the data being queried. This leads to performance times that are nearly constant (i.e. are related to the size of the result set), even as the size of the data set grows [20].

Capacity and latency leave only one remaining axis along which we measure database performance: IO throughput (for mature databases, this is measured in transactions per second). It's a simple trap to fall into to immediately think that we need to scale databases horizontally, especially since we can also scale graph databases vertically to great effect. Vertical scale is in fact an important distinguishing feature of graphs. Because when it comes to vertical scaling, not all databases are created equal. In fact graphs tend to scale very well vertically because they need fewer writes to store the same high-fidelity data because of the incredible expressiveness of the data model - graphs scale vertically by doing less work for the same outcome.

For example, imagine a publishing platform where we'd like to read the latest piece from an author. In a RDBMS we typically select the author's works by joining the authors table to a table of publications on matching author id, and then ordering them by publication date and limiting to the newest handful. Depending on the characteristics of the ordering operation that might be a *O(log(n))* operation which isn't so very bad.

18. This means Neo4j can grow to accommodate a very large number of records, but it tries to stay in the sweet spot for most users, maintaining a careful balance of disk footprint versus storage headroom.

19. This simple fact of life is one of the key reasons that performance tuning is nearly always the very top issue on a relational DBA's mind.

20. As we discussed in Chapter 4, it's still sensible to tune the structure of the graph to suit the queries, even if we're dealing with lower data volumes

However as we can easily see in Figure 5-10, the equivalent graph operation is *O(1)* — meaning constant performance irrespective of dataset size. With a graph we simply follow the outbound relationship called `WROTE` from the author to the work at the head of a list (or tree) of published articles. Should we wish to find older publications, we simply follow the `PREV` relationships and iterate through a linked list (or alternatively recurse through a tree). Writes follow suit because we always insert new publications at the head of the list (or root of a tree), which is another constant time operation. This compares favourably to the alternatives, particularly since it naturally maintains constant time performance for reads.

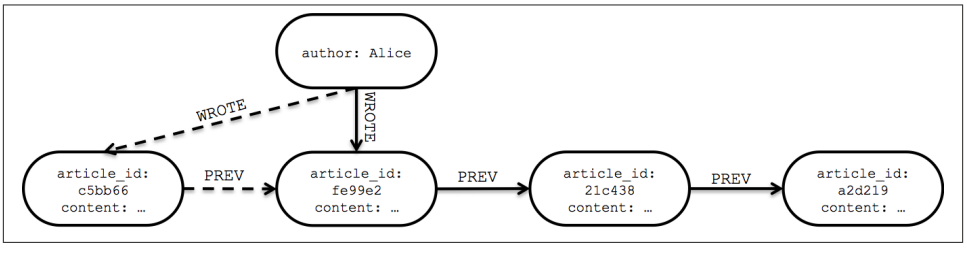


Figure 5-10. Constant time operations for a publishing system

Of course it may be the case that eventually a single machine won't have sufficient IO throughput to serve all the queries directed to it. In that case it's straightforward with Neo4j to build a cluster that allows both high availability and the opportunity to spread read load across many machines. For typical workloads with more reads than writes this solution architecture can be ideal.

But finally, should we exceed the capacity of a cluster then graphs can be still spread across database instances if the application builds in *sharding* logic. Sharding involves the use of a synthetic identifier to join records across database instances at the application level. How well this will perform depends very much on the shape of the graph. Some graphs lend themselves very well to this. Mozilla for instance use the Neo4j graph database as part of their next-generation cloud browser: Pancake. Rather than having a single large graph, they store a large number of small independent graphs, each tied to an end user. This makes it very easy to scale with no performance penalty.

Of course not all graphs have such convenient boundaries. If your graph is large enough that it needs to be broken up and no natural boundaries exist, then the approach you would use is much the same as one would use with a NOSQL store like MongoDB: we create synthetic keys, and relate records via the application layer using those keys plus some application-level resolution algorithm. The main difference from the MongoDB approach is that Neo4j will provide you with a performance boost anytime you are doing traversals within a database instance, while hose parts of the traversal that run between

instances will run at roughly the same speed as a MongoDB join. Overall performance should be markedly faster however.

The holy grail of graph scalability

The eventual aim of most graph databases is to be able to partition a graph across many machines without application-level intervention so that access to the graph can be scaled horizontally. In the general case this is known to be an NP Hard problem, and thus is impractical to solve. This manifests itself in graph databases as unpredictable query times as traversals unexpectedly jump between machines over a (relatively slow) network. However there is innovation happening in this space with triple stores able to scale horizontally very well (at the expense of latency), some non-native graph stores piggy-backing on other distributed databases like Cassandra or HBase, or Neo4j's own horizontally scalable solution. This is certainly an exciting time to be in the data space.

Graph Compute Platforms

One key characteristic of a native graph database is its OLTP capabilities, with clients able to modify and query data in end-user in near real time. While this behavior fits a majority of user-facing applications, sometimes we're prepared to sacrifice latency to permit more sophisticated processing. In the broader NOSQL world, this is most clearly epitomized by the affinity between aggregate data stores and map-reduce, whereby data is pumped through external processing infrastructure (e.g. Apache Hadoop) to distil useful information out of disconnected records.

In the graph space there is a related technology category, the Graph Compute Engine, which is used to conduct specialized analytics involving graph-global operations in contrast to the graph-local operations for which graph databases are optimized [21]. A prominent example of this technology in the literature is Google's Pregel [22] which has

21. A graph global operation is one in which elements from the graph are accessed at random to calculate some property of the graph (e.g. PageRank), by contrast a graph-local operation is where we find a specific starting point and traverse paths from there.

22. See: *http://dl.acm.org/citation.cfm?id=1807184*

an open-source copycat implementation in Apache Giraphe [23] which can both helpfully be thought of as a kind of "Hadoop for graphs."

Graph theory

For analytics, it's truly hard to beat graphs. The properties of graph data are well understood and the fundamental toolkit that is graph theory has broad applicability across domains. From its Eulerian origins in the 18th century, our understanding of networks has continued to provide amazing insight and predictive analysis into data. Graph analysis will be a tremendously important aspect of data science in the coming years.

While some may choose to analyze networks the hard way by reifying graphs out of disconnected aggregates and pumping that data through a plain map-reduce implementation, the combination of a native graph database and native graph processing is far more powerful.

Pregel and equivalent open source implementations like Apache Giraph [24] are examples of just such a graph processing platform (as distinct from, but complimentary to graph databases). Pregel is an evolution of the map-reduce pattern to take advantage of the implicit locality provided by a graph structure. It's no surprise then, that at a high level the architecture resembles the familiar picture of any map-reduce architecture as we see in Figure 5-11.

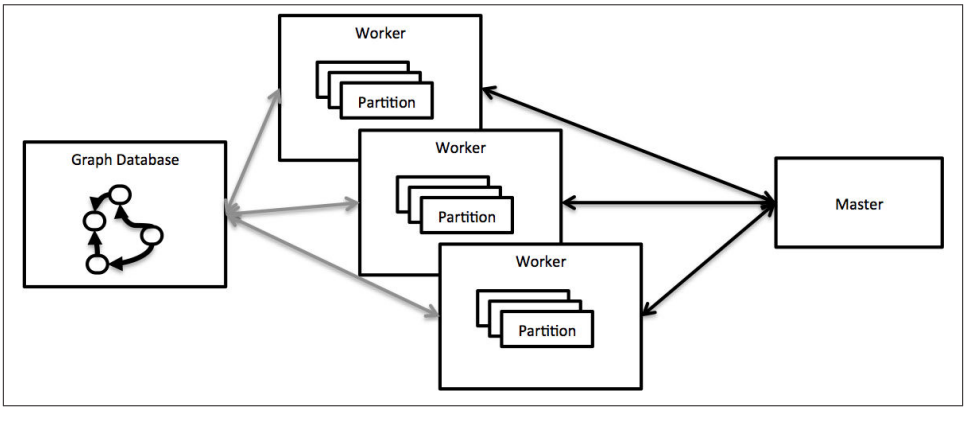


Figure 5-11. Pregel high-level architecture

23. See: *http://incubator.apache.org/giraph/*

24. See: *http://incubator.apache.org/giraph/*

The input to a Pregel computation is a directed graph. The source of the graph can come from any source (e.g. flat files), but it is reasonable to use a graph database as the source since it allows subgraphs to be easily extracted for graph-global processing. The graph is then partitioned so that it can be distributed across machines. While default strategies are available for this, it's typically better for developers to provide a domain-aware strategy to preserve subgraph locality. Once partitions are distributed to machines any non-local accesses are punished by network latency. In turn this implies that each partition tends to contain many nodes to exploit coarse-grained computation.

Pregel has a very node-centric approach to processing. The graph algorithm to be computed is implemented from the point of view of a single node and the transformations that happen to its messages, data and local topology. A node consists solely of its algorithm and an incoming and outgoing communication port. Scaling that algorithm to all of the nodes in the graph is how the overall behavior the system is expressed, while the scheduling of execution and communication for each node is the responsibility of the platform.

Each node uses its incoming and outgoing message queues to transmit the (partial) results of processing between nodes. Communication paths are (usually) constrained by the structure of the graph such that nodes can only address messages to other nodes to which they have an outgoing relationship. However messages can also be sent to any node whose identifier is known (thus avoiding intermediate nodes having to act as latent relays). The logical architecture for partitions and local graph topology is shown in Figure 5-12.

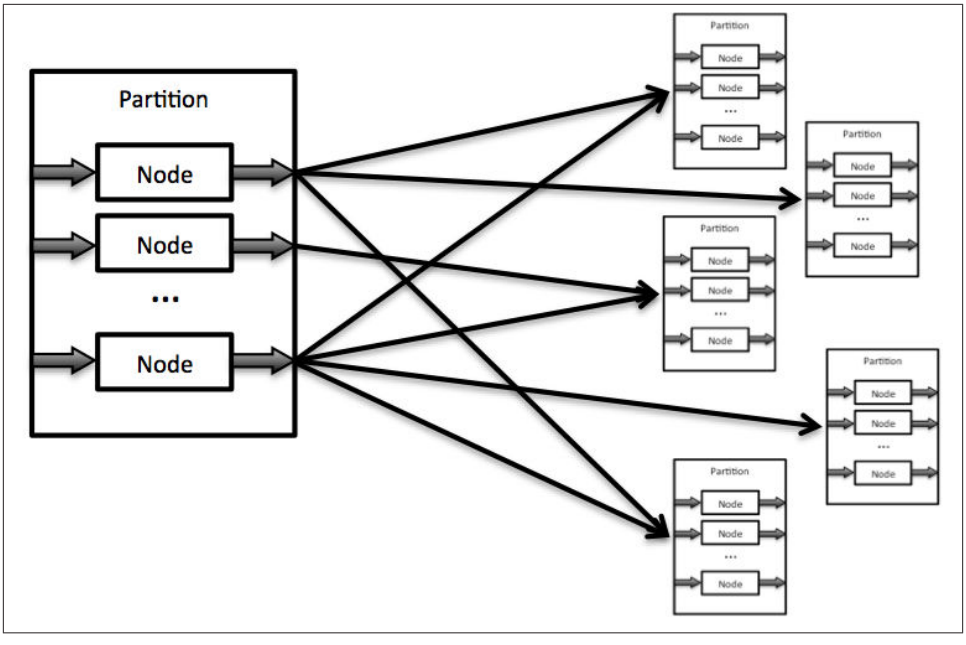


Figure 5-12. Partitions and local graph topology in Pregel

Processing is performed as a set of steps separated by synchronous barriers as per the *Bulk Synchronous Parallel* (or BSP) model. In BSP each *superstep* contains arbitrary parallel computation ending in a barrier where all concurrent processes are synchronized. Within each superstep the code embedded within each node executes logically in parallel, consuming input messages, changing node state (and possibly local graph topology) and producing output messages until the algorithm halts.

When each node's program has ended the superstep ends and messages are transmitted between nodes such that at the start of the next superstep all previous computation is guaranteed to have completed and all message exchanges and topology changes (a by-product of nodes' computation) will be visible.

Since OLAP queries are typically long-running (they process a lot of data), graph compute platforms need to be fault-tolerant. In Pregel, this is achieved via checkpointing where each machine saves its current state into durable storage (such as a graph database) at the beginning of each superstep. In the case of failures the affected partitions can be restarted from checkpoints and recomputed.

Pregel optimizations

Since Pregel assumes that computation is distributed over machines to harness greater processing power, its designers have sensibly baked in a number of optimizations to

help reduce the overheads of running a distributed computation, particularly for avoiding expensive network transfers.

Pregel's *Combiners* allow for multiple messages between nodes on one machine to a node on another to be combined into a single value before transmission. A motivating example is to send a single value for a weighted path over the network rather than numerous individual weights. However combining makes no guarantees about how values are combined, and so it's unlikely they can be uncombined. Hence it's safe to use only functions which are both commutative *and* associative in a combiner.

Aggregators allow (computed) values to be shared across all nodes the graph. Any node can provide a value to an aggregator in a superstep, which is combined with values from any other nodes. On the next superstep, the aggregate value is made available to every node in the graph. The typical use-case for aggregators is for global coordination, for example to terminate computation upon reaching some graph-wide condition. Another motivating case is to maintain graph statistics like the number of relationships or nodes in the graph.

In Pregel, processing completes when all nodes in the graph vote to halt — that each node believes it has no more work to do and when no messages are incident on a node. When processing ends a final result (from an entire graph to as little as a single value) is produced.

Compute or query?

Given that graph compute platforms are powerful machines for processing connected data, the question naturally arises why we wouldn't use them habitually in our information systems? The key to answering this question is understanding our appetite for latency.

Graph compute platforms can compute over large graphs but they do so in a latent manner by the very nature of the work they undertake. Even if we choose to process small graphs in this way however, we still have to deal with the latency of extracting and loading data into the platform which may still take several seconds.

Ultimately today's graph processing platforms are far more suited to OLAP analysis than in-the-clickstream OLTP applications. And in OLTP applications most of the time we don't want to compute enormous volumes of data anyway since most requests are highly contextualized by user, product, etc and are therefore generally graph local.

For a point in case, consider a typical e-commerce solution. A local graph query from the user will be able to efficiently recommend friends' and friend-of-friends' purchases taking into account both the user's recently viewed and purchased items. This is the right choice for an interactive Web store.

However to analyse all products, to rank them against one-another is a reasonable candidate for a graph compute solution. The synergy between the two is when the ranking algorithm enriches the graph such that the recommendations query provides even more accurate recommendations to the user.

Though a distinct technology category in their own right, it's clear that graph databases are a sensible underlay — they have the right APIs and data model affordances to support seamless integration — for graph compute platforms (like Pregel). The virtuous cycle of enriching the (authoritative) graph with the results from graph-global processing is a compelling strategy. Combining the ability to perform graph-local queries for OLTP scenarios on authoritative data in the graph database with the ability to perform arbitrary (graph) computation in an OLAP scenario that enriches the authoritative data provides covers all reasonable expectations of data platform.

Summary

In this chapter we've discussed the common types of graph and shown how property graphs are an excellent choice for pragmatic data modelling. We've also explored the architecture of a popular graph database (Neo4j) and discussed the non-functional characteristics of graph database implementations and what it means for them to be dependable. Finally we discussed the different between a graph database (OLTP, for graph local queries) and a graph processing engine (OLAP, for graph global processing) and explained how graph databases and graph processing are natural bedfellows.

Yet the discussion to this point has asked the reader to take on faith and implied reasoning that the technology is a sound choice (and not just an exciting one). In the next chapter we'll set that straight by presenting real-world deployment planning use cases — harvested from production systems — to bring home the utility of graphs and graph databases for a range of sophisticated scenarios.

CHAPTER 6

Working with a Graph Database

In this chapter we discuss some of the practical issues of working with a graph database. In previous chapters we've looked graph data, and the architecture and capabilities of graph databases; in this chapter, we'll apply that knowledge in the context of developing a graph database application. We'll look at some of the data modeling questions that may arise, and at some of the application architecture choices available to you.

In our experience, graph database applications are highly amenable to being developed using the evolutionary, incremental and iterative software development practices that have risen to prominence in the last decade. A key feature of these practices is the prevalence of testing throughout the software development lifecycle. Here we'll show how to develop your data model and your application in a test-driven fashion.

At the end of the chapter, we'll look at some of the issues you'll need to consider when planning for production.

Data Modeling

We covered modeling and working with graph data in detail in Chapter 4. Here we summarize some of the more important modeling guidelines, and discuss how implementing a graph data model fits with iterative and incremental software development techniques.

Describe the Model in Terms of Your Application's Needs

The *questions* you need to ask of the data help identify entities and relationships. Agile user stories provide a concise means for expressing an outside-in, user-centred view of

an application's needs, and the questions that arise in the course of satisfying this need. [1] Here's an example of a user story for a book reviews Web application:

- `AS A` reader who likes a book
- `I WANT` to know which books other readers who like the same book have liked
- `SO THAT` I can find other books to read

This story expresses a user need, which motivates our data model. From a data modeling point of view, the `AS A` clause establishes a context comprising two entities—a reader and a book—and the `LIKES` relationship that connects them. The `I WANT` clause then poses a question: which books have other readers who like the book I'm reading liked? This question exposes more `LIKES` relationships, and more entities: other readers and other books.

The entities and relationships that we've surfaced in analyzing the user story quickly translate into a simple data model, as shown in Figure 6-1:

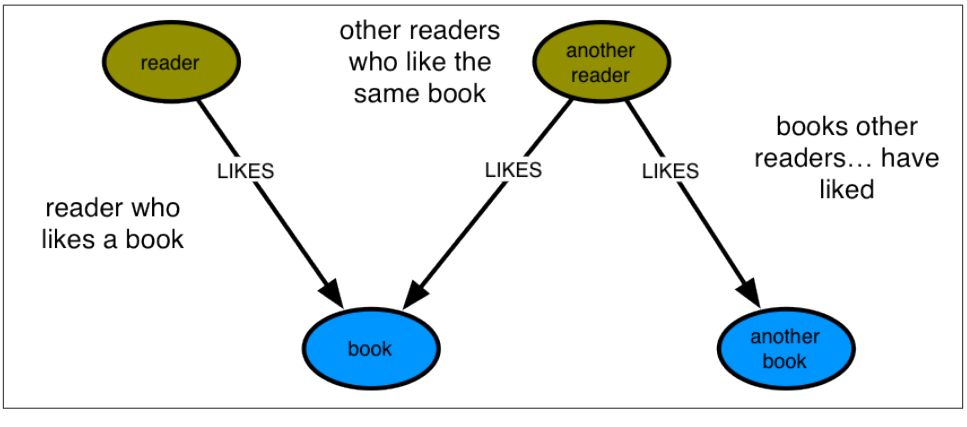


Figure 6-1. Data model for the book reviews user story

Because this data model directly encodes the question presented by the user story, it lends itself to being queried in a way that similarly reflects the structure of the question we want to ask of the data:

```
START reader=node:users(name={readerName})
      book=node:books(isbn={bookISBN})
MATCH reader-[:LIKES]->book<-[:LIKES]-other_readers-[:LIKES]->books
RETURN books.title
```

1. For agile user stories, see Mike Cohn, *User Stories Applied* (Addison-Wesley, 2004)

Nodes for Things, Relationships for Structure

While not applicable in every situation, these general guidelines will help you to choose when to use nodes, and when relationships:

- Use nodes to represent entities—that is, the *things* that are of interest to you in your domain.
- Use relationships to express the *connections* between entities and establish semantic context for each entity, thereby structuring the domain.
- Use node properties to represent entity attributes, plus any necessary entity metadata, such as timestamps, version numbers, etc.
- Use relationship properties to express the strength, weight or quality of a relationship, plus any necessary relationship metadata, such as timestamps, version numbers, etc.

If you're tempted to use a relationship to model an entity—an email, or a review, for example—make certain that this entity cannot be related to more than two other entities. Remember, a relationship must have a start node and an end node—nothing more, nothing less. If you find later that you need to connect something you've modeled as a relationship to more than two other entities, you'll have to refactor the entity inside the relationship out into a separate node. This is a breaking change to the data model, and will likely require

Model Facts as Nodes

When two or more domain entities interact for a period of time, a fact emerges. Represent these facts as separate nodes, with connections to each of the entities engaged in that fact. Modeling an action in terms of its product—that is, in terms of the *thing* that results from the action—produces a similar structure: an intermediate node that represents the outcome of an interaction between two or more entities. You can use timestamp properties on this intermediate node to represent start and end times.

The following examples show how we might model facts and actions using intermediate nodes.

Employment

Figure 6-2 shows how the fact of Ian being employed by Neo Technology in the role of engineer can be represented in the graph.

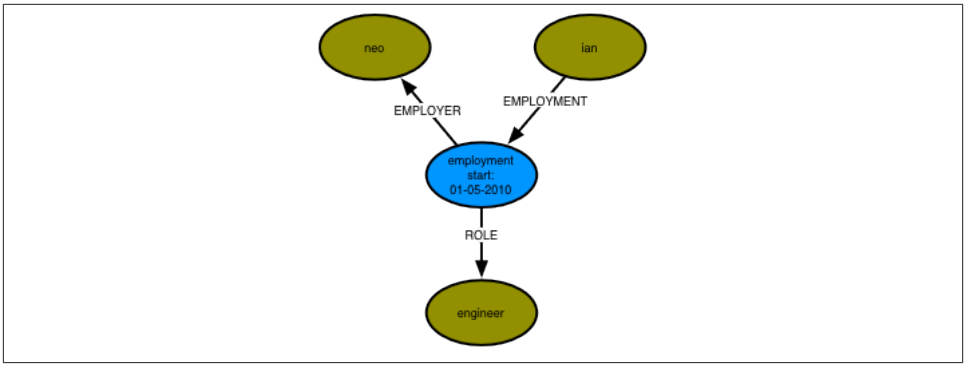


Figure 6-2. Ian was employed as an engineer at Neo Technology

In Cypher, this can be expressed as:

```
ian-[:EMPLOYMENT]->employment-[:EMPLOYER]->neo,
employment-[:ROLE]->engineer
```

Performance

Figure 6-3 shows how the fact of William Hartnell having played the Doctor in the story *The Sensorites* can be represented in the graph.

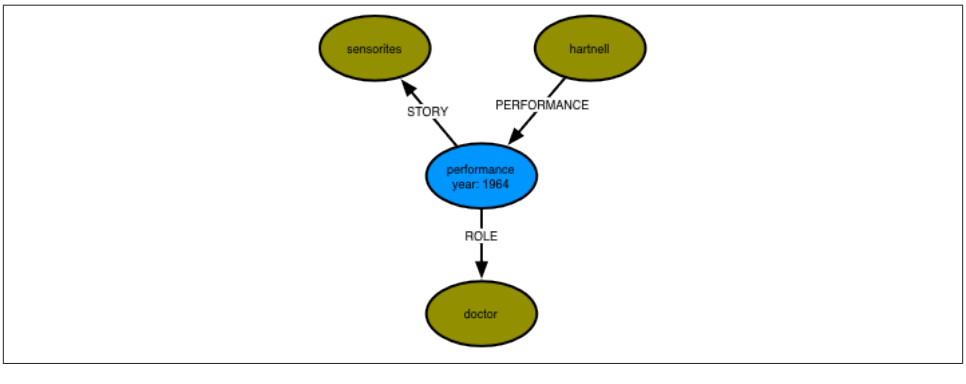


Figure 6-3. William Hartnell played the Doctor in the story The Sensorites

In Cypher:

```
hartnell-[:PERFORMANCE]->performance-[:ROLE]->doctor,
performance-[:STORY]->sensorites
```

Emailing

Figure 6-4 shows the act of Ian emailing Jim and copying in Alistair.

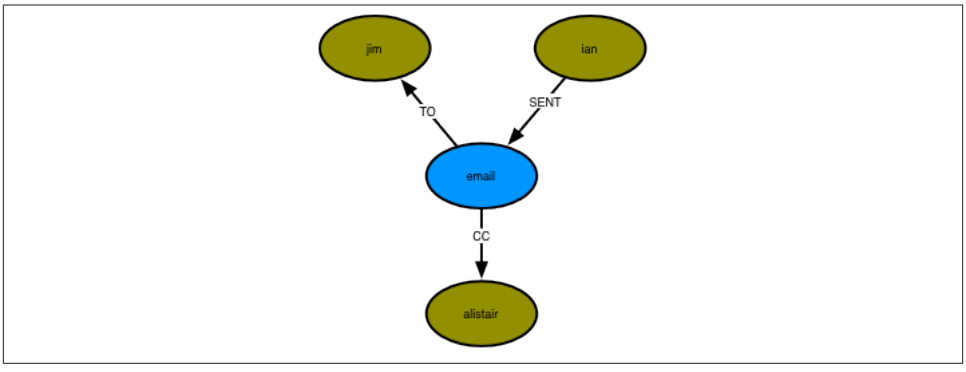


Figure 6-4. Ian emailed Jim, and copied in Alistair

In Cypher, this can be expressed as:

```
ian-[:SENT]->email-[:TO]->jim,
email-[:CC]->alistair
```

Reviewing

Figure 6-5 shows how the act of Alistair reviewing a film can be represented in the graph.

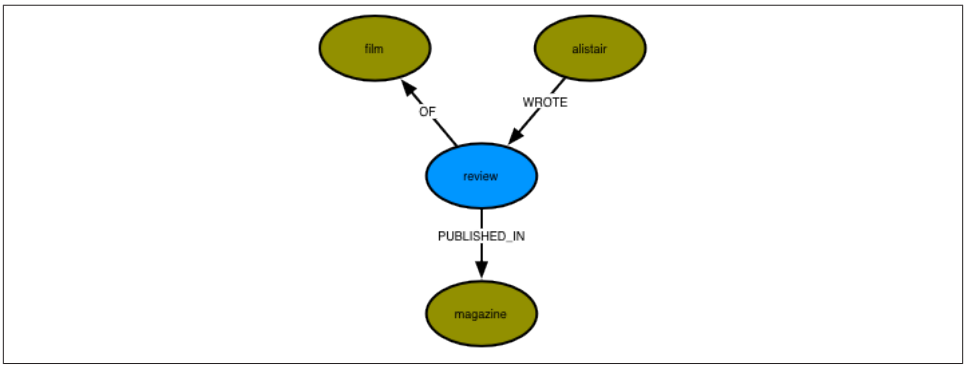


Figure 6-5. Alistair wrote a review of a film, which was published in a magazine

In Cypher:

```
alistair-[:WROTE]->review-[:OF]->film,
review-[:PUBLISHED_IN]->magazine
```

Represent Complex Value Types as Nodes

Value types are things that do not have an identity, and whose equivalence is based solely on their values. Examples include *money*, *address*, and *SKU*. Complex value types are

value types with more than one field or property. *Address*, for example, is a complex value type. Such multi-property value types are best represented as separate nodes:

```
START ord=node:orders(orderid={orderId})
MATCH ord-[:DELIVERY_ADDRESS]->address
RETURN address.first_line, address.zipcode
```

Time

Time can be modelled in several different ways in the graph. Here we describe two techniques: timeline trees and linked lists.

Timeline trees

If you need to find all the events that have occurred over a specific period, you can build a timeline tree, as shown in Figure 6-6:

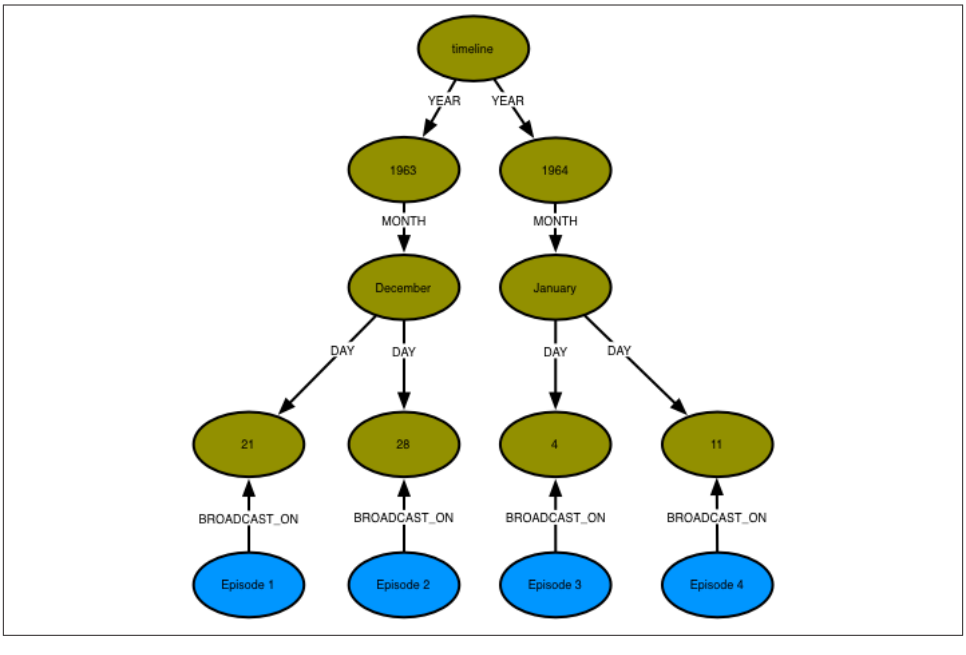


Figure 6-6. A timeline tree showing the broadcast dates for four episodes of a TV programme

You need only insert nodes into the timeline tree as and when they are needed. Assuming the root `timeline` node has been indexed, or is in some other way discoverable, the following Cypher statement will insert all necessary nodes—year, month, day, plus the node to be attached to the timeline:

```
START timeline=node:timeline(name={timelineName})
CREATE UNIQUE timeline-[:YEAR]->(year{value:{year}, name:{yearName}})
              -[:MONTH]->(month{value:{month}, name:{monthName}})
              -[:DAY]->(day{value:{day}, name:{dayName}})
              <-[:BROADCAST_ON]-(n {newNode})
```

Querying the calendar for all events between a start date (inclusive) and an end date (exclusive) can be done with the following Cypher:

```
START timeline=node:timeline(name={timelineName})
MATCH timeline-[:YEAR]->year-[:MONTH]->month-[:DAY]->day<-[:BROADCAST_ON]-n
WHERE ((year.value > {startYear} AND year.value < {endYear})
      OR ({startYear} = {endYear} AND {startMonth} = {endMonth}
          AND year.value = {startYear} AND month.value = {startMonth}
          AND day.value >= {startDay} AND day.value < {endDay})
      OR ({startYear} = {endYear} AND {startMonth} < {endMonth}
          AND year.value = {startYear}
          AND ((month.value = {startMonth} AND day.value >= {startDay})
              OR (month.value > {startMonth} AND month.value < {endMonth})
              OR (month.value = {endMonth} AND day.value < {endDay})))
      OR ({startYear} < {endYear}
          AND year.value = {startYear}
          AND ((month.value > {startMonth})
              OR (month.value = {startMonth} AND day.value >= {startDay})))
      OR ({startYear} < {endYear}
          AND year.value = {endYear}
          AND ((month.value < {endMonth})
              OR (month.value = {endMonth} AND day.value < {endDay}))))
RETURN n
```

The `WHERE` clause here, though somewhat verbose, simply filters each match based on the start and end dates supplied to the query.

Linked lists

Many events have temporal relationships to the events that precede and follow them. You can use `NEXT` and `PREVIOUS` relationships (or similar) to create linked lists that capture this natural ordering, as shown in Figure 6-7. Linked lists allow for very rapid traversal of time-ordered events.

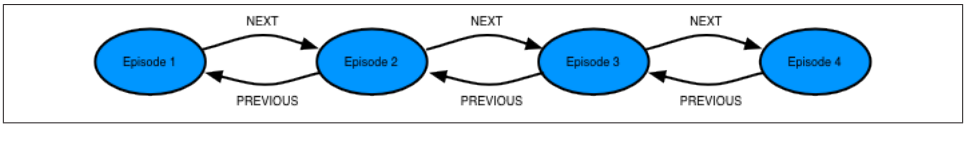


Figure 6-7. A doubly linked list representing a time-ordered series of events

Versioning

A versioned graph allows you to recover the state of the graph at a particular point in time. Neo4j doesn't support versioning as a first-class concept: it is possible, however, to create a versioning scheme inside the graph model whereby nodes and relationships are timestamped and archived whenever they are modified.[2] The downside of such versioning schemes is they leak into any queries written against the graph, adding a layer of complexity to even the simplest query.

Iterative and Incremental Development

Develop the data model feature by feature, user story by user story. This will ensure you identify the relationships your application will use to query the graph. A data model that is developed in line with the iterative and incremental delivery of application features will look quite different from one drawn up using data model-first approach, but it will be the correct model, motivated throughout by specific needs, and the questions that arise in conjunction with those needs.

Graph databases provide for the smooth evolution of your data model. Migrations and denormalization are rarely an issue. New facts and new compositions become new nodes and relationships, while optimizing for performance-critical access patterns typically involves introducing a direct relationship between two nodes that would otherwise be connected only by way of intermediaries. Unlike the relational model, this is not an either/or issue: either the high-fidelity structure, or the high performance compromise. With the graph we retain the original high-fidelity graph structure while at the same time enriching it with new elements that cater to new needs.

You'll quickly see how different relationships can sit side-by-side with one another, catering to different needs without distorting the model in favour of any one particular need. Addresses help illustrate the point here. Imagine, for example, that we are developing a retail application. While developing a fulfilment story, we add the ability to dispatch a parcel to a customer's delivery address, which we find using the following query:

```
START user=node:users(id={userId})
MATCH user-[:DELIVERY_ADDRESS]->address
RETURN address
```

Later on, when adding some billing functionality, we introduce a `BILLING_ADDRESS` relationship. Later still, we add the ability for customers to manage all their addresses. This last feature requires us to find all addresses—whether delivery, or billing, or some other address. To facilitate this, we introduce a general `ADDRESS` relationship:

2. See, for example, *https://github.com/dmontag/neo4j-versioning*, which hooks into Neo4j's transaction lifecycle to created versioned copies of nodes and relationships.

```
START user=node:users(id={userId})
MATCH user-[:ADDRESS]->address
RETURN address
```

By this time, our data model looks something like the one shown in Figure 6-8. `DELIVERY_ADDRESS` specializes the data on behalf of the application's fulfilment needs; `BILLING_ADDRESS` specializes the data on behalf of the application's billing needs; and `ADDRESS` specializes the data on behalf of the application's customer management needs.

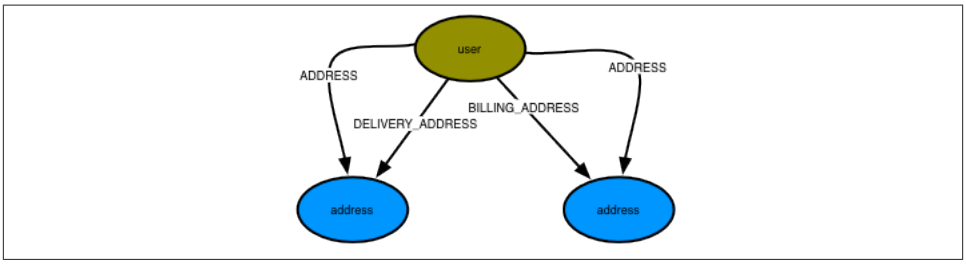


Figure 6-8. Different relationships for different application needs

Being able to add new relationships to meet new application needs doesn't mean you should always do this. You'll invariably identify opportunities for refactoring the model as you go: there'll be plenty of times, for example, where renaming an existing relationship will allow it to be used for two different needs. When these opportunities arise, you should take them. If you're developing your solution in a test-driven manner—described in more detail later in this chapter—you'll have a sound suite of regression tests in place, allowing you to make substantial changes to the model with confidence.

Application Architecture

In planning a graph database-based solution, there are several architectural decisions to be made. These decisions will vary slightly depending on the database product you've chosen; in this section, we'll describe some of the architectural choices, and the corresponding application architectures, available to you when using Neo4j.

Embedded Versus Server

Most databases today run as a server that is accessed through a client library. Neo4j is somewhat unusual in that it can be run in embedded as well as server mode—in fact, going back nearly ten years, its origins are as an embedded graph database.

An embedded database is not the same as an in-memory database. An embedded instance of Neo4j still makes all data durable on disk. Later, in the "Testing" section, we'll discuss `ImpermanentGraphDatabase`, which is an in-memory version of Neo4j designed for testing purposes.

Embedded Neo4j

In embedded mode, Neo4j runs in the same process as your application. Embedded Neo4j is ideal for hardware devices, desktop applications, and for incorporating in your own application servers. Some of the advantages of embedded mode include:

- Low latency. Because your application speaks directly to the database, there's no network overhead.
- Choice of APIs. You have access to the full range of APIs for creating and querying data: the core API, traversal framework, and the Cypher query language.
- Explicit transactions. Using the core API, you can control the transactional lifecycle, executing an arbitrarily complex sequence of commands against the database in the context of a single transaction. The Java APIs also expose the transaction lifecycle, allowing you to plug in custom transaction event handlers that execute additional logic with each transaction.[3]
- Named indexes. Embedded mode gives you full control over the creation and management of named indexes. This functionality is also available through the REST interface; it is not, however, available in Cypher.

When running in embedded mode, however, you should bear in mind the following:

- JVM only. Neo4j is a JVM-based database. Many of its APIs are, therefore, only accessible from a JVM-based language.
- GC behaviours. When running in embedded mode, Neo4j is subject to the garbage collection behaviours of the host application. Long GC pauses can effect on query times. Further, when running an embedded instance as part of an HA cluster, long GC pauses can cause the cluster protocol to trigger a master reelection.
- Database lifecycle. The application is responsible for controlling the database lifecycle, which includes starting it, and closing it safely.

Embedded Neo4j can be clustered for high availability and horizontal read scaling just as the server version. In fact, you can run a mixed cluster of embedded and server instances (clustering is performed at the database level, rather than the server level). This is common in enterprise integration scenarios, where regular updates from other

3. See *http://docs.neo4j.org/chunked/stable/transactions-events.html*

systems are executed against an embedded instance, and then replicated out to server instances.

Server mode

Running Neo4j in server mode is the most common means of deploying the database today. At the heart of each server is an embedded instance of Neo4j. Some of the benefits of server mode include:

- REST API. The server exposes a rich REST API that allows clients to send JSON-formatted requests over HTTP. Responses comprise JSON-formatted documents enriched with hypermedia links that advertise additional features of the dataset.
- Platform independence. Because access is by way of JSON-formatted documents sent over HTPP, a Neo4j server can be accessed by a client running on practically any platform. All that's needed is an HTTP client library.[4]
- Scaling independence. With Neo4j running in server mode, you can scale your database cluster independently of your application server cluster.
- Isolation from application GC behaviours. In server mode, Neo4j is protected from any untoward GC behaviours triggered by the rest of the application. Of course, Neo4j still produces some garbage, but its impact on the garbage collector has been carefully monitored and tuned during development to mitigate any significant side effects. Note, however, that because server extensions allow you to run arbitrary Java code inside the server (see "Server extensions" later in this section), the use of server extensions may impact the server's GC behaviour.

When using Neo4j in server mode, bear in mind the following:

- Network overhead. There is some communication overhead to each HTTP request, though it's fairly minimal. After the first client request, the TCP connection remains open until closed by the client.
- Per request transactions. Each client request is executed in the context of a separate transaction, though there is some support in the REST API for batch operations. For more complex, multistep operations requiring a single transactional context, consider using a server extension (see "Server extensions" later in this section).

Access to Neo4j server is typically by way of its REST API, as discussed above. The REST API comprises JSON-formatted documents over HTTP. Using the REST API you can submit Cypher queries, configure named indexes, and execute several of the built-in graph algorithms. You can also submit JSON-formatted traversal descriptions, and per-

4. A list of Neo4j remote client libraries, as developed by the community, is maintained at *http://docs.neo4j.org/chunked/stable/tutorials-rest.html*

form batch operations. [5] For the majority of use cases the REST API is sufficient; however, if you need to do something you cannot currently accomplish using the REST API, you may wish to consider developing a server extension.

Server extensions

Server extensions allow you to run Java code inside the server. Using server extensions, you can extend the REST API, or replace it entirely.

Extensions take the form of JAX-RS annotated classes. JAX-RS is a Java API for building RESTful resources.[6] Using JAX-RS annotations, you decorate each extension class to indicate to the server which HTTP requests it handles. Additional annotations control request and response formats, HTTP headers, and the formatting of URI templates.

Here's an implementation of a simple server extension that allows a client to request the distance between two members of a social network:

```
@Path("/distance")
public class SocialNetworkExtension
{
    private final ExecutionEngine executionEngine;

    public SocialNetworkExtension( @Context GraphDatabaseService db )
    {
        this.executionEngine = new ExecutionEngine( db );
    }

    @GET
    @Produces("text/plain")
    @Path("/{name1}/{name2}")
    public String getDistance  ( @PathParam("name1") String name1,
                                 @PathParam("name2") String name2 )
    {
        String query = "START first=node:user(name={name1}),\n" +
                "  second=node:user(name={name2})\n" +
                "MATCH p=shortestPath(first-[*..4]-second)\n" +
                "RETURN length(p) AS depth";

        Map<String, Object> params = new HashMap<String, Object>();
        params.put( "name1", name1 );
        params.put( "name2", name2 );

        ExecutionResult result = executionEngine.execute( query, params );

        return String.valueOf( result.columnAs( "depth" ).next() );
```

5. See *http://docs.neo4j.org/chunked/stable/rest-api.html*

6. See *http://jax-rs-spec.java.net/*

```
        }
    }
```

Of particular interest here are the various annotations:

- `@Path("/distance")` specifies that this extension will respond to requests directed to relative URIs beginning */distance*.
- The `@Path("/{name1}/{name2}")` annotation on `getDistance()` further qualifies the URI template associated with this extension. The fragment here is concatenated with */distance* to produce */distance/{name1}/{name2}*, where {name1} and {name2} are placeholders for any characters occurring between the forward slashes. Later on, in the section "Testing server extensions", we'll register this extension under the */socnet* relative URI. At that point, these several different parts of the path ensure that HTTP requests directed to a relative URI beginning */socnet/distance/{name1}/{name2}* (for example, *http://<server>/socnet/distance/Ben/Mike*) will be dispatched to an instance of this extension.
- `@GET` specifies that `getDistance()` should be invoked only if the request is an HTTP GET. `@Produces` indicates that the response entity body will be formatted as *text/plain*.
- The two `@PathParam` annotations prefacing the parameters to `getDistance()` serve to map the contents of the *{name1}* and *{name2}* path placeholders to the method's `name1` and `name2` parameters. Given the URI *http://<server>/socnet/distance/Ben/Mike*, `getDistance()` will be invoked with *Ben* for `name1` and *Mike* for `name2`.
- The `@Context` annotation in the constructor causes this extension to be handed a reference to the embedded graph database inside the server. The server infrastructure takes care of creating an extension and injecting it with a graph database instance, but the very presence of the `GraphDatabaseService` parameter here makes this extension exceedingly testable. As we'll see later, in "Testing server extensions", we can unit test extensions without having to run them inside a server.

Server extensions can be powerful elements in your application architecture. Their chief benefits include:

- Complex transactions. Extensions allow you to execute an arbitrarily complex sequence of operations in the context of a single transaction.
- Choice of APIs. Each extension is injected with a reference to the embedded graph database at the heart of the server. This gives you access to the full range of APIs—core API, traversal framework, graph algorithm package, and Cypher—for developing your extension's behaviour.
- Encapsulation. Because each extension is hidden behind a RESTful interface, you can improve and modify its implementation over time.

- Response formats. You control the response: both the representation format and the HTTP headers. This allows you to create response messages whose contents employ terminology from your domain, rather than the graph-based terminology of the standard REST API (*users*, *products* and *orders* for example, rather than nodes, relationships and properties). Further, in controlling the HTTP headers attached to the response, you can leverage the HTTP protocol for things such as caching and conditional requests.

If you're considering using server extensions, bear in mind the following points:

- JVM only. As with developing against embedded Neo4j, you'll have to use a JVM-based language.
- GC behaviours. You can do arbitrarily complex (and dangerous) things inside a server extension. Monitor garbage collection behaviours to ensure you don't introduce any untoward side effects.

Clustering

As discussed in "Availability" in Chapter 5, Neo4j clusters for high availability and horizontal read scaling using master-slave replication. In this section we discuss some of the strategies you should consider when using clustered Neo4j.

Replication

While all writes to a cluster are coordinated through the master, Neo4j does allow writing through slaves, but even then, the slave syncs with the master before returning to the client. Because of the additional network traffic and coordination protocol, writing through slaves can be an order of magnitude slower than writing direct to the master. The only reasons for writing through slaves are to increase the durability guarantees of each write (the write is made durable on two instances, rather than one) and to ensure you can read your own writes when employing cache sharding (see "Cache sharding" and "Read your won writes" later in this chapter). Since newer versions of Neo4j allow you to specify that writes to the master must be replicated out to one or more slaves, thereby increasing the durability guarantees of writes to the master, the case for writing through slaves is now less compelling. Today it is recommended that all writes be directed to the master, and then replicated to slaves using the `ha.tx_push_factor` and `ha.tx_push_strategy` configuration settings.[7]

7. See *http://docs.neo4j.org/chunked/milestone/ha-configuration.html*

Buffer writes using queues

In high write load scenarios, use queues to buffer writes and regulate load. With this strategy, writes to the cluster are buffered in a queue; a worker then polls the queue and executes batches of writes against the database. Not only does this regulate write traffic, it reduces contention, and allows you to pause write operations without refusing client requests during maintenance periods.

Global clusters

For applications catering to a global audience, it is possible to install a multi-region cluster in multiple datacenters and on cloud platforms such as Amazon Web Services (AWS). A multi-region cluster allows you to service reads from the portion of the cluster geographically closest to the client. In these situations, however, the latency introduced by the physical separation of the regions can sometimes disrupt the coordination protocol; it is, therefore, often desirable to restrict master reelection to a single region. To achieve this, create slave-only databases for the instances you don't want to participate in master reelection; you do this by including the `ha.slave_coordinator_update_mode=none` configuration parameter in an instance's configuration.

Load Balancing

If you're using a clustered graph database, consider load balancing traffic across the cluster to help maximize throughput and reduce latency. Neo4j doesn't include a native load balancer, relying instead on the load balancing capabilities of the network infrastructure.

Separate read traffic from write traffic

Given the recommendation to direct the majority of write traffic to the master, you should consider clearly separating read requests from write requests. Configure your load balancer to direct write traffic to the master, while balancing the read traffic across the entire cluster.

In a Web-based application, the HTTP method is often sufficient to distinguish a request with a significant side-effect—a write—from one that has no significant side-effect on the server: POST, PUT and DELETE can modify server-side resources, whereas GET is side-effect free.

If you're using server extensions, make sure to distinguish read and write operations using `@GET` and `@POST` annotations. If your application depends solely on server extensions, this will suffice to separate the two. If you're using the REST API to submit Cypher queries to the database, however, the situation is not so straightforward. The REST API uses POST as a general "process this" semantic for requests whose contents can include Cypher statements that modify the database. To separate read and write requests in this scenario, introduce a pair of load balancers: a write load balancer that always directs

requests to the master, and a read load balancer that balances requests across the entire cluster. In your application logic, where you know whether the operation is a read or a write, you will then have to decide which of the two addresses you should use for any particular request, as illustrated in Figure 6-9.

When running in server mode, Neo4j exposes a URI that indicates whether that instance is currently the master, and if it isn't, which is. Load balancers can poll this URI at intervals to determine where to route traffic.

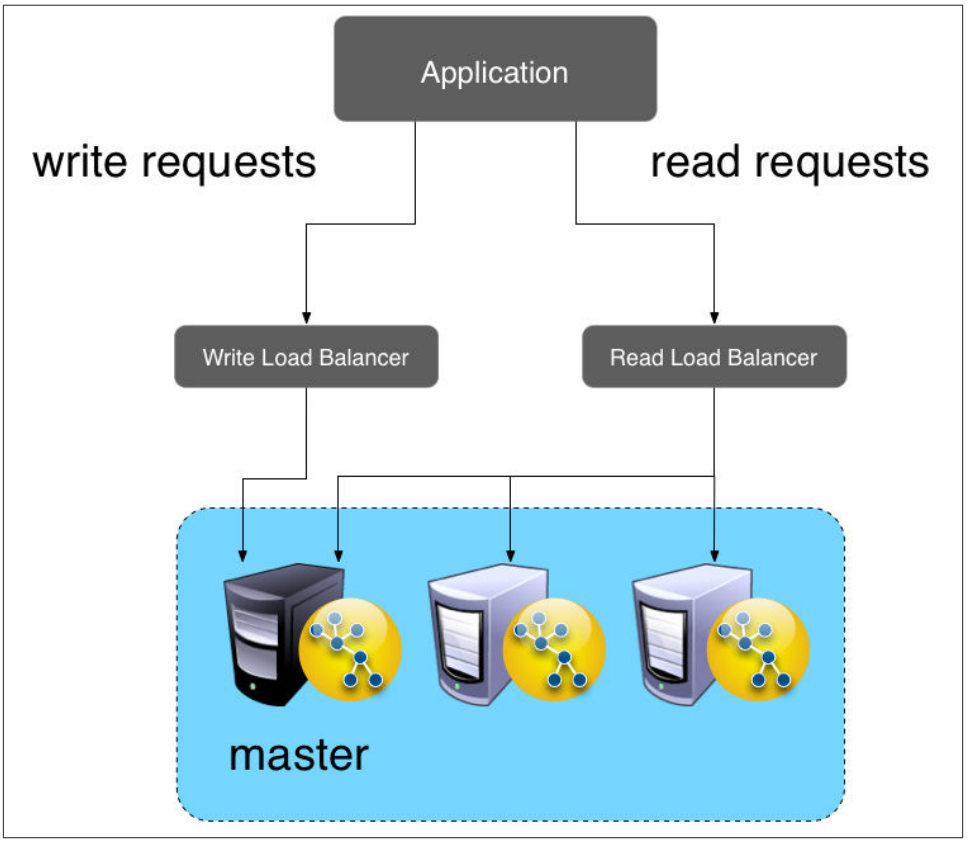


Figure 6-9. Using read/write load balancers to direct requests to a cluster

Cache sharding

Queries run fastest when the portions of the graph needed to satisfy them reside in main memory (that is, in the filesystem cache and the object cache). A single graph database instance today can hold many billions of nodes, relationships and properties; some graphs, therefore, will be just too big to fit into main memory. Partitioning or sharding

a graph is a difficult problem to solve (see "The holy grail of graph scalability" in Chapter 5). How, then, can we provide for high-performance queries over a very large graph?

One solution is to use a technique called cache sharding (Figure 6-10), which consists in routing each request to a database instance in an HA cluster where the portion of the graph necessary to satisfy that request is *likely* already in main memory (remember: every instance in the cluster will contain a full copy of the data). If the majority of an application's queries are graph-local queries, meaning they start from one or more specific points in the graph, and traverse the surrounding subgraphs, then a mechanism that consistently routes queries beginning from the same set of start points to the same database instance will increase the likelihood of each query hitting a warm cache.

The strategy used to implement consistent routing will vary by domain. Sometimes it's good enough to have sticky sessions; other times you'll want to route based on the characteristics of the data set. The simplest strategy is to have the instance which first serves requests for a particular user thereafter serve subsequent requests for that user. Other domain-specific approaches will also work. For example, in a geographical data system we can route requests about particular locations to specific database instances that have been warmed for that location. Both strategies increase the likelihood of the required nodes and relationships already being cached in main memory, where they can be quickly accessed and processed.

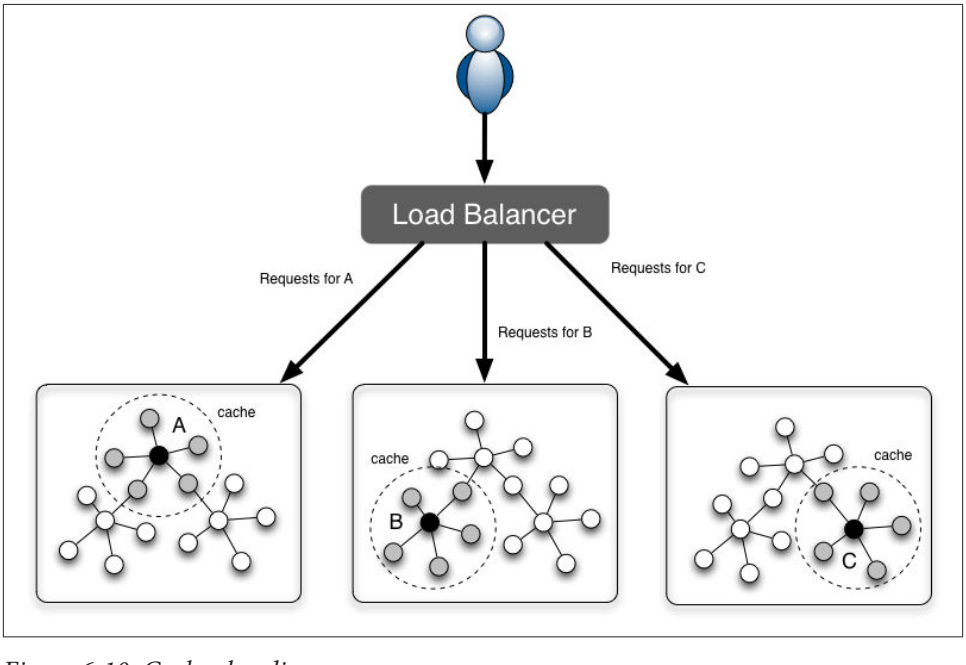


Figure 6-10. Cache sharding

Read your own writes

Occasionally you may need to read your own writes—typically when the application applies an end-user change, and needs on the next request to reflect the effect of this change back to the user. While writes to the master are immediately consistent, the cluster as a whole is eventually consistent. How can we ensure that a write directed to the master is reflected in the next load-balanced read request? One solution is to use the same consistent routing technique used in cache sharding to direct the write to the slave that will be used to service the subsequent read. This assumes that the write and the read can be consistently routed based on some domain criteria in each request.

This is one of the few occasions where it makes sense to write through a slave. But remember: writing through a slave can be an order of magnitude slower than writing direct to the master. Use this technique sparingly. If a high proportion of your writes require you to read your own write, this technique will significantly impact throughput and latency.

Testing

Testing is a fundamental part of the application development process—not only as a means of verifying that a query or application feature behaves correctly, but also as a way of designing and documenting your application and its data model. Throughout this section we emphasize that testing is an everyday activity; by developing your graph database solution in a test-driven manner, you provide for the rapid evolution of your system, and its continued responsiveness to new business needs.

Test-Driven Data Model Development

In discussing data modeling, we've stressed that your graph model should reflect the kinds of queries you want to run against it. By developing your data model in a test-driven fashion you document your understanding of your domain, and validate that your queries behave correctly.

With test-driven data modeling, you write unit tests based on small, representative example graphs drawn from your domain. These example graphs contain just enough data to communicate a particular feature of the domain; in many cases, they might only comprise ten or so nodes, plus the relationships that connect them. Use these examples to describe what is normal for the domain, and also what is exceptional. As you discover anomalies and corner cases in your real data, write a test that reproduces what you've discovered.

The example graphs you create for each test comprise the setup or context for that test. Within this context you exercise a query, and assert that the query behaves as expected. Because you control the context, you, as the author of the test, know what results to expect.

Tests can act like documentation. By reading the tests, developers gain an understanding of the problems and needs the application is intended to address, and the ways in which the authors have gone about addressing them. With this is mind, it's best to use each test to test just one aspect of your domain. It's far easier to read lots of small tests, each of which communicates a discrete feature of your data in a clear, simple and concise fashion, than it is to reverse engineer a complex domain from a single large and unwieldy test. In many cases, you'll find a particular query being exercised by several tests, some of which demonstrate the happy path through your domain, others of which exercise it in the context of some exceptional structure or set of values.[8]

Over time, you'll build up a suite of tests that act as a power regression test mechanism. As your application evolves, and you add new sources of data, or change the model to meet new needs, your regression test suite will continue to assert that existing features continue to behave as they ought. Evolutionary architectures, and the incremental and iterative software development techniques that support them, depend upon a bedrock of asserted behaviour. The unit testing approach to data model development described here allows developers to respond to new business needs with very little risk of undermining or breaking what has come before, confident in the continued quality of the solution.

Example: A test-driven social network data model

In this example we're going to demonstrate developing a very simple Cypher query for a social network. Given the names of a couple of members of the network, our query determines how far apart: direct friends are at distance 1, friends-of-a-friend at distance 2, and so on.

The first thing we do is create a small graph that is representative of our domain. Using Cypher, we create a network comprising ten nodes and eight relationships:

```
public GraphDatabaseService createDatabase()
{
    // Create nodes
    String cypher = "CREATE\n" +
            "(ben {name:'Ben', _label:'user'}),\n" +
            "(arnold {name:'Arnold', _label:'user'}),\n" +
            "(charlie {name:'Charlie', _label:'user'}),\n" +
            "(gordon {name:'Gordon', _label:'user'}),\n" +
            "(lucy {name:'Lucy', _label:'user'}),\n" +
            "(emily {name:'Emily', _label:'user'}),\n" +
            "(sarah {name:'Sarah', _label:'user'}),\n" +
            "(kate {name:'Kate', _label:'user'}),\n" +
            "(mike {name:'Mike', _label:'user'}),\n" +
```

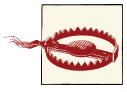
8. Tests not only act as documentation, they can also be used to generate documentation. All of the Cypher documentation in the Neo4j manual (see *http://docs.neo4j.org/chunked/stable/cypher-query-lang.html*) is generated automatically from the unit tests used to develop Cypher.

```
                "(paula {name:'Paula', _label:'user'}),\n" +
                "ben-[:FRIEND]->charlie,\n" +
                "charlie-[:FRIEND]->lucy,\n" +
                "lucy-[:FRIEND]->sarah,\n" +
                "sarah-[:FRIEND]->mike,\n" +
                "arnold-[:FRIEND]->gordon,\n" +
                "gordon-[:FRIEND]->emily,\n" +
                "emily-[:FRIEND]->kate,\n" +
                "kate-[:FRIEND]->paula";

        GraphDatabaseService db = new ImpermanentGraphDatabase();

        ExecutionEngine engine = new ExecutionEngine( db );
        engine.execute( cypher );

        // Index all nodes in "user" index
        Transaction tx = db.beginTx();
        try
        {
            Iterable<Node> allNodes =
                    GlobalGraphOperations.at( db ).getAllNodes();
            for ( Node node : allNodes )
            {
                if ( node.hasProperty( "name" ) )
                {
                    db.index().forNodes( "user" )
                            .add( node, "name", node.getProperty( "name" ) );
                }

            }
            tx.success();
        }
        finally
        {
            tx.finish();
        }

        return db;
    }
```

There are two things of interest in `createDatabase()`. The first is the use of `ImpermanentGraphDatabase`, which is a lightweight, in-memory version of Neo4j designed specifically for unit testing. By using `ImpermanentGraphDatabase`, you avoid having to clear up store files on disk after each test. The class can be found in the Neo4j kernel test jar, which can be obtained with the following dependency reference:

```
<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j-kernel</artifactId>
    <version>${project.version}</version>
    <type>test-jar</type>
```

```
    <scope>test</scope>
</dependency>
```



`ImpermanentGraphDatabase` should only be used in unit tests. It is not a production-ready, in-memory version of Neo4j.

The second thing of interest in `createDatabase()` is the code for adding nodes to a named index. Cypher doesn't add the newly created nodes to an index, so for now, we have to do this manually by iterating all the nodes in the sample graph, adding any with a `name` property to the `users` named index.

Having created a sample graph, we can now write our first test. Here's the test fixture for testing our social network data model and its queries:

```
public class SocialNetworkTest
{
    private static GraphDatabaseService db;
    private static SocialNetworkQueries queries;

    @BeforeClass
    public static void init()
    {
        db = createDatabase();
        queries = new SocialNetworkQueries( db );
    }

    @AfterClass
    public static void shutdown()
    {
        db.shutdown();
    }

    @Test
    public void shouldReturnShortestPathBetweenTwoFriends() throws Exception
    {
        // when
        ExecutionResult results = queries.distance( "Ben", "Mike" );

        // then
        assertTrue( results.iterator().hasNext() );
        assertEquals( 4, results.iterator().next().get( "distance" ) );
    }

    // more tests
}
```

This test fixture includes an initialization method, annotated with `@BeforeClass`, which executes before any tests start. Here we call `createDatabase()` to create an instance of

the sample graph, and an instance of `SocialNetworkQueries`, which houses the queries under development.

Our first test, `shouldReturnShortestPathBetweenTwoFriends()`, tests that the query under development can find a path between any two members of the network—in this case, `Ben` and `Mike`. Given the contents of the sample graph, we know that `Ben` and `Mike` are connected, but only remotely, at a distance of 4. The test, therefore, asserts that the query returns a non-empty result containing a `distance` value of 4.

Having written the test, we now start developing our first query. Here's the implementation of `SocialNetworkQueries`:

```
public class SocialNetworkQueries
{
    private final ExecutionEngine executionEngine;

    public SocialNetworkQueries( GraphDatabaseService db )
    {
        this.executionEngine = new ExecutionEngine( db );
    }

    public ExecutionResult distance( String firstUser, String secondUser )
    {
        String query = "START first=node:user({firstUserQuery}),\n" +
                " second=node:user({secondUserQuery})\n" +
                "MATCH p=shortestPath(first-[*..4]-second)\n" +
                "RETURN length(p) AS distance";

        Map<String, Object> params = new HashMap<String, Object>();
        params.put( "firstUserQuery", "name:" + firstUser );
        params.put( "secondUserQuery", "name:" + secondUser );

        return executionEngine.execute( query, params );
    }

    // More queries
}
```

In the constructor for `SocialNetworkQueries` we create a Cypher `ExecutionEngine`, passing in the supplied database instance. We store the execution engine in a member variable, which allows it to be reused over and again throughout the lifetime of the queries instance. The query itself we implement in the `distance()` method. Here we create a Cypher statement, initialize a map containing the query parameters, and execute the statement using the execution engine.

If `shouldReturnShortestPathBetweenTwoFriends()` passes (it does), we then go on to test additional scenarios. What happens, for example, if two members of the network are separated by more than 4 connections? We write up the scenario and what we expect the query to do in another test:

```
@Test
public void shouldReturnNoResultsWhenNoPathUnderAtDistance4OrLess() throws Exception
{
    // when
    ExecutionResult results = queries.distance( "Ben", "Arnold" );

    // then
    assertFalse( results.iterator().hasNext() );
}
```

In this instance, this second test passes without us having to modify the underlying Cypher query. In many cases, however, a new test will force you to modify a query's implementation. When that happens, modify the query to make the new test pass, and then run all the tests in the fixture. A failing test anywhere in the fixture indicates you've broken some existing functionality. Continue to modfy the query until all tests are green once again.

Testing server extensions

Server extensions can be developed in a test-driven manner just as easily as embedded Neo4j. Using the simple server extension described earlier, here's how we test it:

```
@Test
public void extensionShouldReturnDistance() throws Exception
{
    // given
    SocialNetworkExtension extension = new SocialNetworkExtension( db );

    // when
    String distance = extension.getDistance( "Ben", "Mike" );

    // then
    assertEquals( "4", distance );
}
```

Because the extension's constructor accepts a `GraphDatabaseService` instance, we can inject a test instance (an `ImpermanentGraphDatabase` instance), and then call its methods as per any other object.

If, however, we wanted to test the extension running inside a server, we have a little more setup to do:

```
public class SocialNetworkExtensionTest
{
    private static CommunityNeoServer server;

    @BeforeClass
    public static void init() throws IOException
    {
        server = ServerBuilder.server()
                .withThirdPartyJaxRsPackage(
```

```
                        "org.neo4j.graphdatabases.queries.server",
                        "/socnet" )
                .build();
        server.start();
        populateDatabase( server.getDatabase().getGraph() );
    }

    @AfterClass
    public static void teardown()
    {
        server.stop();
    }

    @Test
    public void serverShouldReturnDistance() throws Exception
    {
        ClientConfig config = new DefaultClientConfig();
        Client client = Client.create( config );

        WebResource resource = client
                .resource( "http://localhost:7474/socnet/distance/Ben/Mike" );
        ClientResponse response = resource
                .accept( MediaType.TEXT_PLAIN )
                .get( ClientResponse.class );

        assertEquals( 200, response.getStatus() );
        assertEquals( "text/plain",
                response.getHeaders().get( "Content-Type" ).get( 0 ) );
        assertEquals( "4", response.getEntity( String.class ) );
    }

    // Populate graph

}
```

Here we're using an instance of `CommunityNeoServer` to host the extension. We create the server and populate its database in the test fixture's `init()` method using a `ServerBuilder`, which is a helper class from Neo4j's server test jar. This builder allows us to register the extension package, and associate it with a relative URI space (in this case, everything below */socnet*). Once `init()` has completed, we have a server instance up and running and listening on port 7474.

In the test itself, `serverShouldReturnDistance()`, we access this server using an HTTP client from the Jersey client library.[9] The client issues an HTTP GET request for the resource at *http://localhost:7474/socnet/distance/Ben/Mike*. (At the server end, this request is dispatched to an instance of `SocialNetworkExtension`.) When the client re-

9. See *http://jersey.java.net/*

ceives a response, the test asserts that the HTTP status code, content-type, and content of the response body are correct.

Performance Testing

The test-driven approach that we've described so far communicates context and domain understanding, and tests for correctness. It does not, however, test for performance. What works fast against a small, twenty node sample graph may not work so well when confronted with a much larger graph. Therefore, to accompany your unit tests, you should consider writing a suite of query performance tests. On top of that, you should also invest in some thorough application performance testing early in your application's development lifecycle.

Query performance tests

Query performance tests are not the same as full-blown application performance tests. All we're interested in at this stage is whether a particular query performs well when run against a graph that is proportionate to the kind of graph you expect to encounter in production. Ideally, these tests are developed side-by-side with your unit tests. There's nothing worse than investing a lot of time in perfecting a query, only to discover it is not fit for production-sized data.

When creating query performance tests, bear in mind the following guidelines:

- Create a suite of performance tests that exercise the queries developed through your unit testing. Record the performance figures so that you can see the relative effects of tweaking a query, modifying the heap size, or upgrading from one version of Neo4j to another.
- Run these tests often, so that you quickly become aware of any deterioration in performance. You might consider incorporating these tests into a continuous delivery build pipeline, failing the build if the test results exceed a certain value.
- You can run these tests in-process on a single thread. There's no need to simulate multiple clients at this stage: if the performance is poor for a single client, it's unlikely to improve for multiple clients. Even though they are not strictly speaking unit tests, you can drive them using the same unit testing framework you use to develop your unit tests.
- Run each query many times, picking starting nodes at random each time, so that you can see the effect of starting from a cold cache, which is then gradually warmed as multiple queries execute.

Application performance tests

Application performance tests, as distinct from query performance tests, test the performance of the entire application under representative production usage scenarios.

As with query performance tests, we recommend that this kind of performance testing is done as part of everyday development, side-by-side with the development of application features, rather than as a distinct project phase.[10] To facilitate application performance testing early in the project lifecycle, it is often necessary to develop a "walking skeleton", an end-to-end slice through the entire system, which can be accessed and exercised by performance test clients. By developing a walking skeleton, you not only provide for performance testing, you also establish the architectural context for the graph database part of your solution. This allows you to verify your application architecture, and identify layers and abstractions that allow for discrete testing of individual components.

Performance tests serve two purposes: they demonstrate how the system will perform when used in production, and they drive out the operational affordances that make it easier to diagnose performance issues, incorrect behaviour, and bugs. What you learn in creating and maintaining a performance test environment will prove invaluable when it comes to deploying and operating the system for real.

As with your unit tests and query performance tests, application performance tests prove most valuable when employed in an automated delivery pipeline, where successive builds of the application are automatically deployed to a testing environment, the tests executed, and the results automatically analyzed. Log files and test results should be stored for later retrieval, analysis and comparison. Regressions and failures should fail the build, prompting developers to address the issues in a timely manner. One of the big advantages of conducting performance testing over the course of an application's development lifecycle, rather than at the end, is that failures and regressions can very often be tied back to a recent piece of development; this allows you to diagnose, pinpoint and remedy issues rapidly and succinctly.

For generating load, you'll need a load-generating agent. For Web applications, there are several open source stress and load testing tools available, including Grinder, JMeter and Gatling.[11] When testing load balanced Web applications, you should ensure your test clients are distributed across different IP addresses so that requests are balanced across the cluster.

10. A thorough discussion of agile performance testing can be found in Alistair Jones and Patrick Kua, "Extreme Performance Testing", *The ThoughtWorks Anthology, Volume 2* (Pragmatic Bookshelf, 2012)

11. See *http://grinder.sourceforge.net/*, *http://jmeter.apache.org/* and *http://gatling-tool.org/*. Max De Marzi describes using Gatling to test Neo4j here: *http://maxdemarzi.com/2013/02/14/neo4j-and-gatling-sitting-in-a-tree-performance-t-e-s-t-ing/*

Testing with representative data

For both query performance testing and application performance testing you will need a dataset that is representative of the data you will encounter in production. It will, therefore, be necessary to create or otherwise source such a dataset. In some cases you can obtain a dataset from a third party, or adapt an existing dataset that you own, but unless these datasets are already in the form of a graph, you will have to write some custom export-import code.

In many cases, however, you're starting from scratch. If this is the case, dedicate some time to creating a dataset builder. As with the rest of the software development lifecycle, this is best done in an iterative and incremental fashion. Whenever you introduce a new element into your domain's data model, as documented and tested in your unit tests, add the corresponding element to your performance dataset builder. That way, your performance tests will come as close to real-world usage as your current understanding of the domain allows.

When creating a representative dataset, try to reproduce any domain invariants you have identified: the minimum, maximum and average number of relationships per node, the spread of different relationship types, property value ranges, and so on. Of course, it's not always possible to know these things upfront, and often you'll find yourself working with rough estimates until such point as production data is available to verify your assumptions.

While ideally we would always test with a production-sized dataset, it is often not possible or desirable to reproduce extremely large volumes of data in a test environment. In such cases, you should at least ensure that you build a representative dataset whose size exceeds the capacity of the object cache. That way, you'll be able to observe the effect of cache evictions, and querying for portions of the graph not currently held in main memory.

Representative datasets also help with capacity planning. Whether you create a full-sized dataset, or a scaled-down sample of what you expect the production graph to be, your representative dataset will give you some useful figures for estimating the size of the production data on disk. These figures then help you plan how much memory to allocate to the filesystem cache and the JVM heap (see "Capacity Planning" for more details).

In the following example we're using a dataset builder called Neode to build a sample social network:[12]

```
private void createSampleDataset( GraphDatabaseService db )
{
    DatasetManager dsm = new DatasetManager( db, new Log()
```

12. See *https://github.com/iansrobinson/neode*. Max De Marzi describes an alternative graph generation technique here: *http://maxdemarzi.com/2012/07/03/graph-generator/*

```
        {
            @Override
            public void write( String value )
            {
                System.out.println(value);
            }
        } );

        // User node specification
        NodeSpecification userSpec =
                dsm.nodeSpecification( "user",
                                       indexableProperty( "name" ) );

        // FRIEND relationship specification
        RelationshipSpecification friend =
                dsm.relationshipSpecification( "FRIEND" );

        Dataset dataset =
                dsm.newDataset( "Social network example" );

        // Create user nodes
        NodeCollection users =
                userSpec.create( 1000000 )
                        .update( dataset );


        // Relate users to each other
        users.createRelationshipsTo(
                getExisting( users )
                        .numberOfTargetNodes( minMax( 50, 100 ) )
                        .relationship( friend )
                        .relationshipConstraints( BOTH_DIRECTIONS ) )
                .updateNoReturn( dataset );

        dataset.end();
    }
```

Neode uses node and relationship specifications to describe which nodes and relationships are to be found in the graph, together with their properties and permitted property values. Neode then provides a fluent interface for creating and relating nodes.

Capacity Planning

At some point in your application's development lifecycle you'll want to start planning for production deployment. In many cases, an organization's project management gating processes mean a project cannot get underway without some understanding of the production needs of the application. Capacity planning is essential both for budgeting purposes and for ensuring there is sufficient lead time for procuring hardware and reserving production resources.

In this section we describe some of the techniques you can use for hardware sizing and capacity planning. Your ability to estimate your production needs depends on a number of factors: the more data you have regarding representative graph sizes, query performance, and the number of expected users and their behaviors, the better your ability to estimate your hardware needs. You can gain much of this information by applying the techniques described in the "Testing" section early in your application development lifecycle. In addition, you should understand the cost/performance tradeoffs available to you in the context of your business needs.

Optimization Criteria

As you plan your production environment you will be faced with a number of optimization choices. Which you favour will depend upon your business needs:

- Cost. You can optimize for cost by installing the minimum hardware necessary to get the job done.
- Performance. You can optimize for performance by procuring the fastest solution (subject to budgetary constraints).
- Redundancy. You can optimize for redundancy and availability by ensuring the database cluster is big enough to survive a certain number of machine failures (i.e., to survive two machines failing, you will need a cluster comprising five instances).
- Load. With a replicated graph database solution, you can optimize for load by scaling horizontally (for read load), and vertically (for write load).

Performance

Redundancy and load can be costed in terms of the number of machines necessary to ensure availability (five machines to provide continued availability in the face of two machines failing, for example) and scalability (one machine per some number of concurrent requests, as per the calculations in the "Load" section later in this Chapter). But what about performance? How can we cost performance?

Calculating the cost of graph database performance

In order to understand the cost implications of optimizing for performance, we need to understand the performance characteristics of the database stack. As we describe in "Architecture" in Chapter 5, a graph database uses disk for durable storage, and main memory for caching portions of the graph. In Neo4j, the caching parts of main memory are further divided between the filesystem cache (which is typically managed by the operating system) and the object cache. The filesystem cache is a portion of off-heap RAM into which files on disk are read and cached before being served to the application.

The object cache is an on-heap cache that stores object instances of nodes, relationships and properties.

Spinning disk is by far the slowest part of the database stack. Queries that have to reach all the way down to spinning disk will be orders of magnitude slower than queries which touch only the object cache. Disk access can be improved by using SSDs in place of spinning disks, providing an approximate 20 times increase in performance, or enterprise flash hardware, which can reduce latencies yet further.

Spinning disks and SDDs are cheap, but not very fast. Far more significant performance benefits accrue when the database has to deal only with the caching layers. The filesystem cache offers up to 500 times the performance of spinning disk, while the object cache can be up to 5000 times faster.

For comparative purposes, graph database performance can be expressed as a function of the percentage of data available at each level of the object cache-filesystem cache-disk hierarchy:

*(% graph in object cache x 5000) * (% graph in filesystem cache * 500) * 20 (if using SSDs)*

An application in which 100 percent of the graph is available in the object cache (as well as in the filesystem cache, and on disk) will be more performant than one in which 100 percent is available on disk, but only 80 percent in the filesystem cache and 20 percent in the object cache.

Performance optimization options

There are, then, three areas in which we can optimize for performance:

- Increase the object cache (from 2 GB, all the way up to 200 GB or more in exceptional circumstances)
- Increase the percentage of the store mapped into the filesystem cache
- Invest in faster disks: SSDs or enterprise flash hardware

The first two options here require adding more RAM. In costing the allocation of RAM, however, there are a couple of things to bear in mind. First, while the size of the store files in the filesystem cache map one-to-one with the size on disk, graph objects in the object cache can be up to 10 times bigger than their on-disk representations. Allocating RAM to the object cache is, therefore, far more expensive per graph element than allocating it to the filesystem cache. The second point to bear in mind relates to the location of the object cache. If your graph database uses an on-heap cache, as does Neo4j, then increasing the size of the cache requires allocating more heap. Most modern JVMs do

not cope well with heaps much larger than 8 GB. Once you start growing the heap beyond this size, garbage collection can impact the performance of your application.[13]

As Figure 6-11 shows, the sweet spot for any cost versus performance tradeoff lies around the point where you can map your store files in their entirety into RAM, while allowing for a healthy, but modestly-sized object cache. Heaps of between 4 and 8 GB are not uncommon, though in many cases, a smaller heap can actually improve performance (by mitigating expensive GC behaviors).

Calculating how much RAM to allocate to the heap and the filesystem cache depends on your knowing the projected size of your graph. Building a representative dataset early in your application's development lifecycle will furnish you with some of the data you need to make your calculations. If you cannot fit the entire graph into main memory (that is, at a minimum, into the filesystem cache), you should consider cache sharding (see the section on "Cache sharding" earlier in this chapter).

The Neo4j documentation includes details of the size of records and objects that you can use in your calculations. See *http://docs.neo4j.org/chunked/stable/configuration-caches.html*.

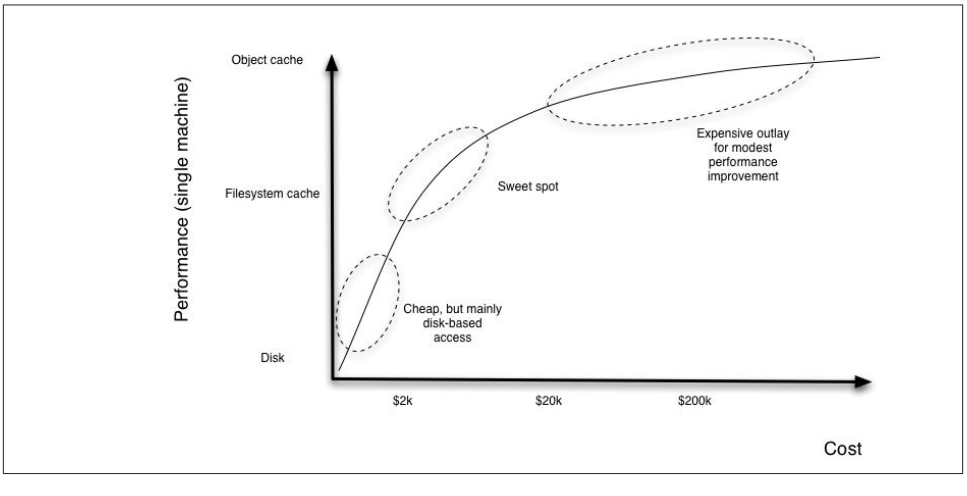


Figure 6-11. Cost versus performance tradeoffs

In optimizing a graph database solution for performance, bear in mind the following guidelines:

13. Neo4j Enterprise Edition includes a cache implementation that mitigates the problems encountered with large heaps, and is being successfully used with heaps in the order of 200 GB.

- Utilize the filesystem cache as much as possible; if possible map your store files in their entirety into this cache
- Tune the JVM heap and keep an eye on GC behaviors
- Consider using fast disks—SSDs or enterprise flash hardware—to bring up the bottom line when disk access becomes inevitable

Redundancy

Planning for redundancy requires you to determine how many instances in a cluster you can afford to lose while keeping the application up and running. For non-business critical applications, this figure might be as low as one (or even zero); once a first instance has failed, another failure will render the application unavailable. Business-critical applications will likely require redundancy of at least two; that is, even after two machines have failed, the application continues serving requests.

For a graph database whose cluster management protocol requires a majority of coordinators to be available to work properly, redundancy of one can be achieved with three or four instances, redundancy of two with five instances. Four is no better than three in this respect, since if two instances from a four-instance cluster become unavailable, the remaining coordinators will no longer be able to achieve majority.

Neo4j permits coordinators to be located on separate machines from the database instances in the cluster. This allows you to scale the coordinators independently of the databases. With three database instances and five coordinators located on different machines, you could lose two databases and two coordinators, and still achieve majority, albeit with a lone database.

Load

Optimizing for load is perhaps the trickiest part of capacity planning. As a rule of thumb:

*number of concurrent requests = (1000 / average request time (in milliseconds)) * number of cores per machine * number of machines*

Actually determining what some of these figures are, or are projected to be, can sometimes be very difficult:

- Average request time. This covers the period from when a server receives a request, to when it sends a response. Performance tests can help determine average request time, assuming the tests are are running on representative hardware against a representative dataset (you'll have to hedge accordingly if not). In many cases, the

"representative dataset" itself is based on a rough estimate; you should modify your figures whenever this estimate changes.

- Number of concurrent requests. You should distinguish here between average load and peak load. Determining the number of concurrent requests a new application must support is a difficult thing to do. If you're replacing or upgrading an existing application you may have access to some recent production statistics you can use to refine your estimates. Some organizations are able to extrapolate from existing application data the likely requirements for a new application. Other than that, it's down to your stakeholders to estimate the projected load on the system, but beware inflated expectations.

CHAPTER 7

Graph Data in the Real World

In this chapter we look at some of the common real-world use cases for graph databases and identify the reasons why organisations choose to use a graph database rather than a relational or other NOSQL store. The bulk of the chapter comprises three in-depth use cases, with details of the relevant data models and queries. Each of these examples has been drawn from a real-world production system; the names, however, have been changed, and the technical details simplified where necessary to hide any accidental complexity, and thereby highlight key design points.

Reasons for Choosing a Graph Database

Throughout this book, we've sung the praises of the graph data model, its power and flexibility, and its innate expressiveness. When it comes to applying a graph database to a real-world problem, with real-world technical and business constraints, organisations choose graph databases for the following reasons:

1. "Minutes to milliseconds" performance. Query performance and responsiveness are top of many organisations' concerns with regard to their data platforms. Online transactional systems, large Web applications in particular, must respond to end-users in milliseconds if they are to be successful. In the relational world, as an application's dataset size grows, join pains begin to manifest themselves, and performance deteriorates. Using index-free adjacency, a graph database turns complex joins into fast graph traversals, thereby maintaining millisecond performance irrespective of the overall size of the dataset.
2. Drastically-accelerated development cycles. The graph data model reduces the impedance mismatch that has plagued software development for decades, thereby reducing the development overhead of translating back and forth between an object model and a tabular relational model. More importantly, the graph model reduces the impedance mismatch between the technical and business domains: subject

matter experts, architects and developers can talk about and picture the core domain using a shared model that is then incorporated into the application itself.

3. Extreme business responsiveness. Successful applications rarely stay still; changes in business conditions, user behaviours and technical and operational infrastructures drive new requirements. In the past, this has required organisations to undertake careful and lengthy data migrations that involve modifying schemas, transforming data, and maintaining redundant data to serve old and new features. The schema-free nature of a graph database coupled with the ability to simultaneously relate data elements in lots of different ways allows a graph database solution to evolve as the business evolves, reducing risk and time-to-market.
4. Enterprise ready. Data is important: when employed in a business-critical application, a data technology must be robust, scalable, and more often than not, transactional. A graph database like Neo4j provides all the enterprise *-ilities* --ACID transactionality, high-availability, horizontal read scalability, and storage of billions of entities—on top of the performance and flexibility characteristics discussed above.

Common Use Cases

In this section we describe some of the most common graph database use cases, identifying how the graph model and the specific characteristics of the graph database can be applied to generate competitive insight and significant business value.

Social Networks

Social networking allows organisations to gain competitive and operational advantage by leveraging social computing to identify the impact of social context on individual preferences and behaviour. As Facebook's use of the term *social graph* implies, the graph data model and by extension graph databases are a natural fit for this overtly relationship-centerd domain. Social networks help us identify the direct *and* indirect relationships between people, groups and the things with which they interact, allowing users to rate, review and discover each other and the things they care about. By understanding who interacts with whom, how people are connected, and what representatives within a group are likely to do or choose based on the aggregate behaviour of the group, we generate tremendous insight into the unseen forces that influence individual behaviours. We discuss predictive modeling and its role in social network analysis in more detail in the section "Graph Theory and Predictive Modeling", in Chapter 8.

Social relations may be either explicit or implicit. Explicit relations occur wherever social subjects volunteer a direct link—by liking someone on Facebook, for example, or indicating someone is a current or former colleague, as happens on LinkedIn. Implicit relations emerge out of other relationships that indirectly connect two or more subjects

by way of an intermediary. We can relate subjects based on their opinions, likes, purchases, even the products of their day-to-day work. Such indirect relationships lend themselves to being applied in multiple suggestive and inferential ways: we can say that A is *likely* to know, or like, or otherwise connect to B based on some common intermediaries. In so doing, we move from social network analysis into the realm of recommendation engines.

Recommendation Engines

Recommendation engines are a prime example of generating end-user value through the application of an inferential or suggestive capability. Whereas line of business applications typically apply deductive and precise algorithms—calculating payroll, applying tax, and so on—to generate end-user value, recommendation engines are inductive and suggestive, identifying people, products or services an individual or group is *likely* to have some interest in.

Recommendation engines establish relationships between people and things: other people, products, services, media content—whatever is relevant to the domain in which the engine is employed. Relationships are established based on users' behaviours, whether purchasing, producing, consuming, rating or reviewing the resources in question. The engine can then identify resources of interest to a particular individual or group, or individuals and groups likely to have some interest in a particular resource. With the first approach, identifying resources of interest to a specific user, the behaviour of the user in question—their purchasing behaviour, expressed preferences, and attitudes as expressed in ratings and reviews—are correlated with those of other users in order to identify similar users and thereafter the things with they are connected. The second approach, identifying users and groups for a particular resource, focusses on the characteristics of the resource in question; then engine then identifies similar resources, and the users associated with those resources.

As with social networks, recommendation engines depend on understanding the connections between things, as well as the quality and strength of those connections—all of which are best expressed as a property graph. Queries are primarily graph local, in that they start with one or more identifiable subjects, whether people or resources, and thereafter discover surrounding portions of the graph.

Taken together, social networks and recommendation engines provide key differentiating capabilities in the areas of retail, recruitment, sentiment analysis, search and knowledge management. Graphs are a good fit for the densely connected data structures germane to each of these areas; storing and querying this data using a graph database allows an application to surface end-user realtime results that reflect recent changes to the data, rather than pre-calculated, stale results.

Geospatial

Geospatial is the original graph use case: Euler solved the Seven Bridges of Königsberg problem by positing a mathematical theorem which later came to form the basis of graph theory. Geospatial applications of graph databases range from calculating routes between locations in an abstract network such as a road or rail network, airspace network, or logistical network (as illustrated by the logistics example later in this chapter) to spatial operations such as find all points of interest in a bounded area, find the center of a region, and calculate the intersection between two or more regions.

Geospatial operations depend upon specific data structures, ranging from simple weighted and directed relationships, through to spatial indexes, such as R-Trees, which represent multi-dimensional properties using tree data structures.[1] As indexes, these data structures naturally take the form of a graph, typically hierarchical in form, and as such they are a good fit for a graph database. Because of the schema-free nature of graph databases, geospatial data can reside in the database beside other kinds of data—social network data, for example—allowing for complex multidimensional querying across several domains.[2]

Geospatial applications of graph databases are particularly relevant in the areas of telecommunications, logistics, travel, timetabling and route planning.

Master Data Management

Master data is data that is critical to the operation of a business, but which itself is non-transactional. Master data includes data concerning users, customers, products, suppliers, departments, geographies, sites, cost centers and business units. In large organisations, this data is often held in many different places, with lots of overlap and redundancy, in many different formats, and with varying degrees of quality and means of access. Master Data Management (MDM) is the practice of identifying, cleaning, storing, and, most importantly, governing this data. Its key concerns include managing change over time as organisational structures change, businesses merge, and business rules change; incorporating new sources of data; supplementing exiting data with externally sourced data; addressing the needs of reporting, compliance and business intelligence consumers; and versioning data as its values and schemas change.

Graph databases don't provide a full MDM solution; they are however, ideally applied to the modeling, storing and querying of hierarchies, master data metadata, and master data models. Such models include type definitions, constraints, relationships between entities and the mappings between the model and the underlying source systems. A

1. See *http://en.wikipedia.org/wiki/R-tree*
2. Neo4j Spatial is an open source library of utilities that implement spatial indexes and expose Neo4j data to geotools. See *https://github.com/neo4j/spatial*

graph database's structured yet schema-free data model provides for ad hoc, variable and exceptional structures—schema anomalies that commonly arise when there are multiple redundant data sources—while at the same time allowing for the rapid evolution of the master data model in line with changing business needs.

Network Management

In Chapter 4 we looked at a simple data center domain model, showing how the physical and virtual assets inside a data center can be easily modeled with a graph. Communications networks are graph structures; graph databases are, therefore, a great fit for modeling, storing and querying this kind of domain data.

A graph representation of a network allows us to catalogue assets, visualize how they are deployed, and identify the dependencies between them. The graph's connected structure, together with a query language like Cypher, enable us to conduct sophisticated impact analyses, answering questions such as:

- Which parts of the network—which applications, services, virtual machines, physical machines, data centers, routers, switches and fibre—do important customers depend on? (top-down analysis)
- Conversely, which applications and services, and ultimately, customers, in the network will be affected if a particular network element—a router or switch, for example—fails? (bottom-up analysis)
- Is there redundancy throughout the network for the most important customers?

Graph database solutions complement existing network management and analysis tools. As with master data management, they can be used to bring together data from disparate inventory systems, providing a single view of the network and its consumers, from the smallest network element all the way to application and services and the customer who use them. A graph database representation of the network can also be used to enrich operational intelligence based on event correlations: whenever an event correlation engine (a Complex Event Processor, for example) infers a complex event from a stream of low-level network events, it can assess the impact of that event using the graph model, and thereafter trigger any necessary compensating or mitigating actions.[3]

Today, graph databases are being successfully employed in the areas of telecommunications, network management and analysis, cloud platform management, data center and IT asset management, and network impact analysis, where they are reducing impact analysis and problem resolution times from days and hours down to minutes and sec-

3. See *http://en.wikipedia.org/wiki/Complex_event_processing*

onds. Performance, flexibility in the face of changing network schemas, and fit for the domain are all important factors here.

Access Control and Authorization

Access control and authorization solutions store information about parties (e.g. administrators, organizational units, end-users) and resources (e.g. files, shares, network devices, products, services, agreements), together with the rules governing access to those resources; they then apply these rules to determine who can access or manipulate a resource. Access control has traditionally been implemented either using directory services or by building a custom solution inside an application's back-end. Hierarchical directory structures, however, cannot cope with the non-hierarchical organizational and resource dependency structures that characterize multi-party distributed supply chains. Hand-rolled solutions, particularly those developed on a relational database, suffer join pain as the dataset size grows, becoming slow and unresponsive, and ultimately delivering a poor end-user experience.

A graph database such as Neo4j can store complex, densely-connected access control structures spanning billions of parties and resources. Its structured, schema-free data model supports both hierarchical and non-hierarchical structures, while its extensible property model allows for capturing rich metadata regarding every element in the system. With a query engine that can traverse millions of relationships per second, access lookups over large, complex structures execute in milliseconds.

As with network management and analysis, a graph database access control solution allows for both top-down and bottom-up queries:

- Which resources—company structures, products, services, agreements and end-users—can a particular administrator manage? (Top-down)
- Which resource can an end-user access?
- Given a particular resource, who can modify its access settings? (Bottom-up)

Graph database access control and authorization solutions are particularly applicable in the areas of content management, federated authorization services, social networking preferences, and software as a service offerings, where they realizing minutes to milliseconds increases in performance over their hand-rolled, relational predecessors.

Detailed Use Cases

In this section we describe three use cases in detail: social networking and recommendations, access control, and logistics. Each use case is drawn from one or more production applications of Neo4j; however, company names, context, data models and

queries have been tweaked to eliminate accidental complexity and so highlight important design and implementation choices.

Social Networking and Recommendations

SocNet is a social networking and recommendations application that allows users to discover their own professional network, and identify other users with particular skill sets. Users work for companies, work on projects, and have one or more interests or skills. Based on this information, SocNet can describe a user's professional network by identifying other subscribers who share their interests. Searches can be restricted to the user's current company, or extended to encompass the entire subscriber base. SocNet can also identify individuals with specific skills who are directly or indirectly connected to the current user; such searches are useful when looking for a subject matter expert for a current engagement.

SocNet illustrates how a powerful inferential capability can be developed using a graph database. While many line-of-business applications are deductive and precise—calculating tax or salary, or balancing debits and credits, for example—a new seam of end-user value opens up when we apply inductive algorithms to our data. This is what SocNet does. Based on people's interests and skills, and their work history, the application can suggest likely candidates for including in one's professional network. These results are not precise in the way a payroll calculation must be precise, but they are extremely useful nonetheless.

The important point with SocNet is that the connections between people are inferred. Contrast this with LinkedIn, for example, where users explicitly declare that they know or have worked with someone. This is not to say that LinkedIn is solely a precise social networking capability, for it too applies inductive algorithms to generate further insight. But with SocNet even the primary tie, `A-[:KNOWS]->B`, is inferred, rather than volunteered.

Today, SocNet depends on users having supplied information regarding their interests, skills and work history before it can infer their professional social relations. But with the core inferential capabilities in place, the platform is set to generate even greater insight for less end-user effort. Skills and interests, for example, can be inferred from the processes and products surrounding people's day-to-day work activities. Writing code, writing documents, exchanging emails: activities such as these require interacting with systems that allow us to capture hints as to a person's skills. Other sources of data that help contextualize a user include group memberships and meetup lists. While the use case presented here does not cover these higher-order inferential features, their implementation requires mostly application integration and partnership agreements rather than any significant change to the graph or the algorithms used.

SocNet data model

To help describe the SocNet data model, we've created a small sample graph, as shown in Figure 7-1, which we'll use throughout this section to illustrate the Cypher queries behind the main SocNet use cases.

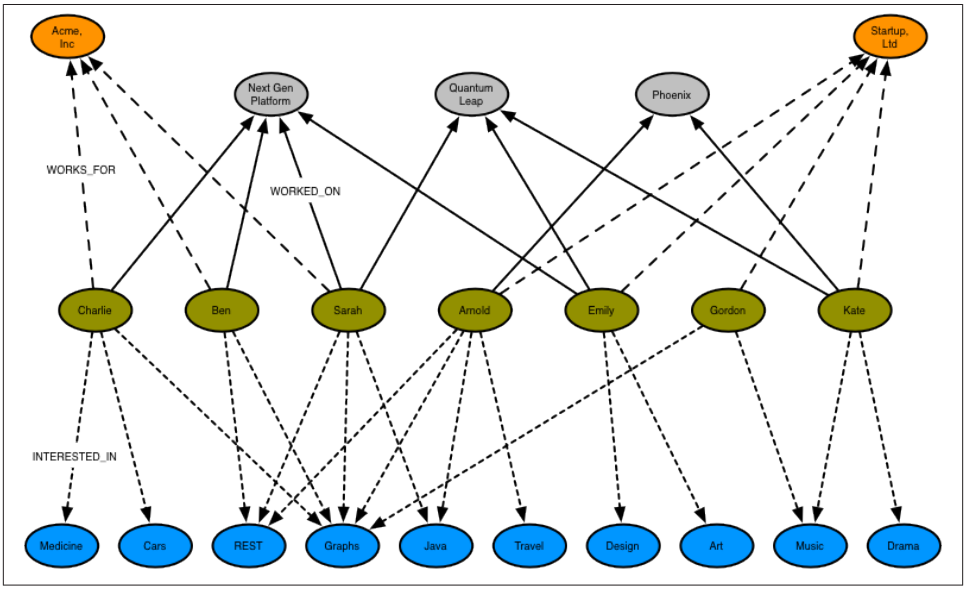


Figure 7-1. Sample of the SocNet social network

The sample graph shown here has just two companies, each with several employees. An employee is connected to their employer by a `WORKS_FOR` relationship. Each employee is `INTERESTED_IN` one or more topics, and has `WORKED_ON` one or more projects. Occasionally, employees from different companies work on the same project.

This structure addresses two important use cases:

- Given a user, inferring social relations—that is, identifying their professional social network—based on shared interests and skills.
- Given a user, recommend someone that they have worked with, or who has worked with people they have worked with, who has a particular skill.

The first use case helps build communities around shared interests; the second helps identify people to fill specific project roles.

Inferring social relations

SocNet's graph allows it to infer a user's professional social network by following the relationships that connect an individual's interests to other people. The strength of the recommendation depends on the number of shared interests. If Sarah is interested in Java, graphs, and REST, Ben in graphs and REST, and Charlie in graphs, cars and medicine, then there is a likely tie between Sarah and Ben based on their mutual interest in graphs and REST, and another tie between Sarah and Charlie based on their mutual interest in graphs, with the tie between Sarah and Ben stronger than the one between Sarah and Charlie (two shared interests versus one).

The following illustration shows the pattern representing colleagues who share a user's interests. The `subject` node refers to the subject of the query (in the example above, this will be Sarah). This node can be looked up in an index. The remaining nodes will be discovered once the pattern is anchored to the subject node and then flexed around the graph.

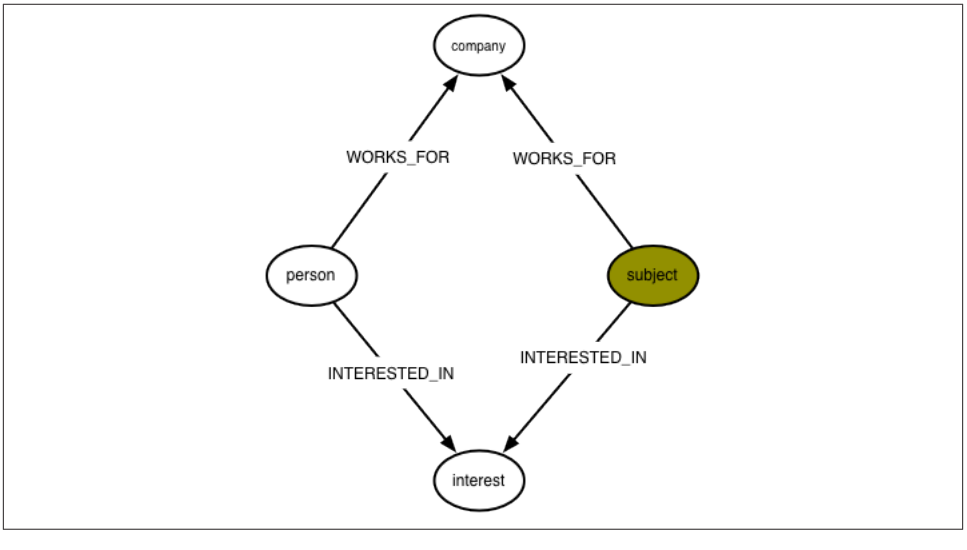


Figure 7-2. Pattern to find colleagues who share a user's interests

The Cypher to implement this query is shown below:

```
START  subject=node:user(name={name})
MATCH  subject-[:WORKS_FOR]->company<-[:WORKS_FOR]-person,
       subject-[:INTERESTED_IN]->interest<-[:INTERESTED_IN]-person
RETURN person.name AS name,
       COUNT(interest) AS score,
       COLLECT(interest.name) AS interests
ORDER BY score DESC
```

The query works as follows:

- The `START` clause looks up the subject of the query in the `user` index, and assigns the result to the `subject` identifier.
- The `MATCH` clause then matches this `person` with people who work for the same company, and who share one or more of their interests. If the subject of the query is Sarah, who works for Acme, then in the case of Ben, this `MATCH` clause will match twice: `('Acme')<-[:WORKS_FOR]-('Ben')-[:INTERESTED_IN]->('graphs')` and `('Acme')<-[:WORKS_FOR]-('Ben')-[:INTERESTED_IN]->('REST')`. In the case of Charlie, it will match once: `('Acme')<-[:WORKS_FOR]-('Charlie')-[:INTERESTED_IN]->('graphs')`.
- The `RETURN` clause creates a projection of the matched data: for each matched colleague, we extract their name, count the number of interests they have in common with the subject of the query (aliasing this result as `score`), and, using `COLLECT`, create a comma-separated list of these mutual interests. Where a person has multiple matches, as does Ben in our example, `COUNT` and `COLLECT` serve to aggregate their matches into a single row in the returned results (in fact, both `COUNT` and `COLLECT` can perform this aggregating function alone).
- Finally, we order the results based on each colleague's `score`, highest first.

Running this query against our sample graph, with Sarah as the subject, yields the following results:

```
+--------------------------------------+
| name      | score | interests        |
+--------------------------------------+
| "Ben"     | 2     | ["Graphs","REST"] |
| "Charlie" | 1     | ["Graphs"]       |
+--------------------------------------+
2 rows
0 ms
```

Here's the portion of the graph that was matched to generate these results:

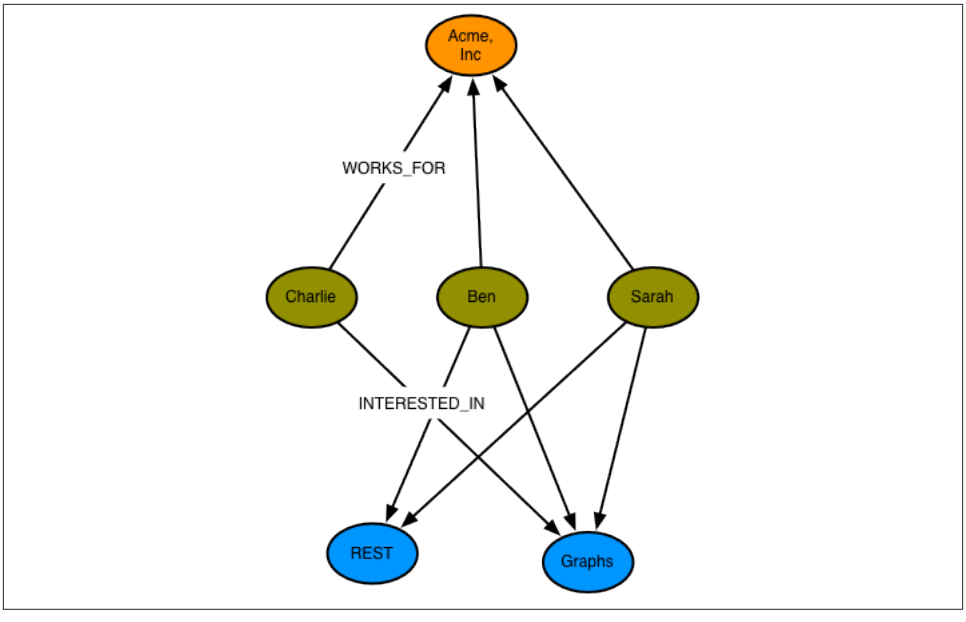


Figure 7-3. Colleagues who share Sarah's interests

You'll notice that this query only finds people who work for the same company as Sarah. If we want to extend the search to find people who work for other companies, we need to modify the query slightly:

```
START  subject=node:user(name={name})
MATCH  subject-[:INTERESTED_IN]->interest<-[:INTERESTED_IN]-person,
       person-[:WORKS_FOR]->company
RETURN person.name AS name,
       company.name AS company,
       COUNT(interest) AS score,
       COLLECT(interest.name) AS interests
ORDER BY score DESC
```

The changes are as follows:

- In the `MATCH` clause, we no longer require matched persons to work for the same company as the subject of the query. (We do, however, still capture the company for whom each matched person works, because we want to return this information in the results.)
- In the `RETURN` clause we now include the company details for each matched person.

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

```
+-------------------------------------------------------------+
| name      | company        | score | interests              |
```

```
+-------------------------------------------------------------+
| "Arnold"  | "Startup, Ltd" | 3     | ["Java","Graphs","REST"] |
| "Ben"     | "Acme, Inc"    | 2     | ["Graphs","REST"]        |
| "Gordon"  | "Startup, Ltd" | 1     | ["Graphs"]               |
| "Charlie" | "Acme, Inc"    | 1     | ["Graphs"]               |
+-------------------------------------------------------------+
4 rows
0 ms
```

And here's the portion of the graph that was matched to generate these results:

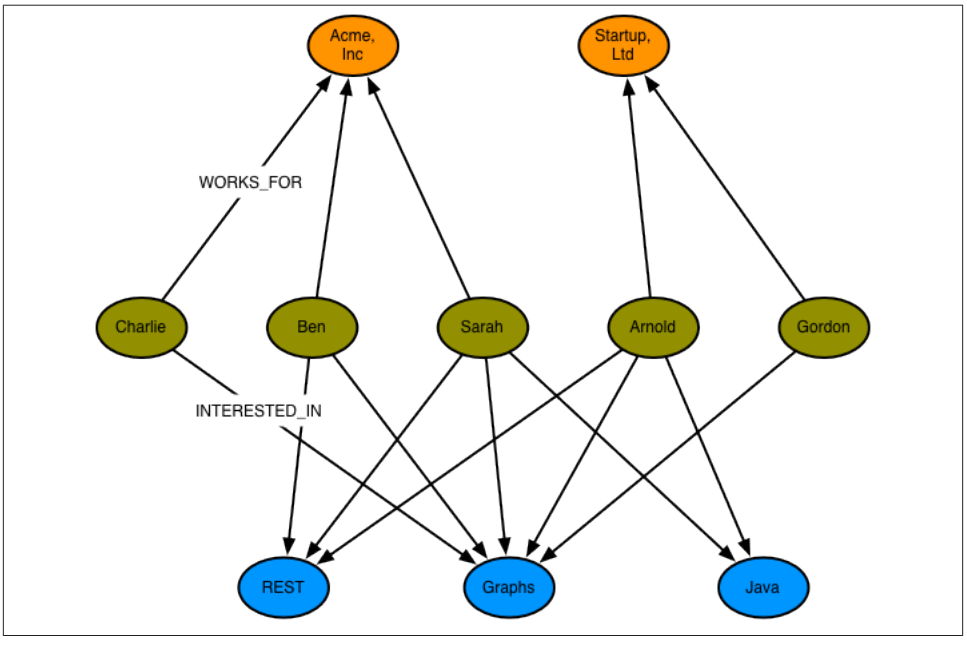


Figure 7-4. People who share Sarah's interests

While Ben and Charlie still figure in the results, it turns out that Arnold, who works for *Startup, Ltd*, has most in common with Sarah: three topics compared to Ben's two and Charlie's one.

Finding colleagues with particular interests

In the second SocNet use case, we turn from inferring social relations based on shared interests to finding individuals who have a particular skill-set, and who have either worked with the person who is the subject of the query, or worked with people who have worked with the subject. By applying the graph in this manner, we can source individuals to staff project roles based on their social ties to people we trust—or at least have worked with.

The social ties in question arise from individuals having worked on the same project. Contrast this with the previous use case, where the social ties were inferred based on shared interests. If people have worked on the same project, we infer a social tie. The projects, then, form intermediate nodes that bind two or more people together: they are instances of collaboration that have brought people into contact with one another. Anyone we discover in this fashion is a candidate for including in our results—as long as they possess the interests or skills we are looking for.

Here's a Cypher query that finds colleagues, and colleagues-of-colleagues, who have one or more particular interests:

```
START subject=node:user(name={name})
MATCH p=subject-[:WORKED_ON]->()-[:WORKED_ON*0..2]-()
        <-[:WORKED_ON]-person-[:INTERESTED_IN]->interest
WHERE person<>subject AND interest.name IN {interests}
WITH person, interest, MIN(LENGTH(p)) as pathLength
RETURN person.name AS name,
       COUNT(interest) AS score,
       COLLECT(interest.name) AS interests,
       ((pathLength - 1)/2) AS distance
ORDER BY score DESC
LIMIT {resultLimit}
```

This is quite a complex query. Let's break it down little, and look at each part in more detail:

- The `START` clause looks up the subject of the query in the `user` index, and assigns the result to the `subject` identifier.
- The `MATCH` clause finds people who are connected to the `subject` by way of having worked on the same project, or having worked on the same project as people who have worked with the `subject`. For each person we match, we capture their interests. This match is then further refined by the `WHERE` clause, which exclude nodes that match the subject of the query, and ensures we only match people who are interested in the things we care about. For each successful match, we assign the entire path of the match—that is, the path that extends from the subject of the query all the way through the matched person to their interest—to the identifier `p`. We'll look at this `MATCH` clause in more detail shortly.
- The `WITH` clause pipes the results to the `RETURN` clause, filtering out redundant paths as it does so. Redundant paths are present in the results at this point because colleagues and colleagues-of-colleagues are often reachable through different paths, some longer than others. We want to filter these longer paths out. That's exactly what the `WITH` clause does. The `WITH` clause emits triples comprising a person, an interest, and the length of the path from the subject of the query through the person to their interest. Given that any particular person/interest combination may appear more than once in the results, but with different path lengths, we want to aggregate

these multiple lines by collapsing them to a triple containing only the shortest path, which we do using `MIN(LENGTH(p)) as pathLength`.

- The `RETURN` clause creates a projection of the data, performing more aggregation as it does so. The data piped by the `WITH` clause to the `RETURN` clause contains one entry per person per interest: if a person matches two of the supplied interests, there will be two separate data entries. We aggregate these entries using `COUNT` and `COLLECT`: `COUNT` to create an overall score for a person, `COLLECT` to create a comma-separated list of matched interests for that person. As part of the results, we also calculate how far the matched person is from the subject of the query: we take the `pathLength` for that person, subtract 1 (for the `INTERESTED_IN` relationship at the end of the path), and then divide by 2 (because the person is separated from the subject by pairs of `WORKED_ON` relationships). Finally, we order the results based on `score`, highest `score` first, and limit them according to a `resultLimit` parameter supplied by the query's client.

The `MATCH` clause in the preceding query uses a variable length path, `[:WORKED_ON*0..2]`, as part of a larger pattern to match people who have worked directly with the subject of the query, as well as people who have worked on the same project as people who have worked with the subject. Because each person is separated from the subject of the query by one or two pairs of `WORKED_ON` relationships, SocNet could have written this portion of the query as `MATCH p=subject-[:WORKED_ON*2..4]-person-[:INTERESTED_IN]->interest`, with a variable length path of between two and four `WORKED_ON` relationships. However, long variable length paths can be relatively inefficient. When writing such queries, it is advisable to restrict variable length paths to as narrow a scope as possible. To increase the performance of the query, SocNet use a fixed length outgoing `WORKED_ON` relationship that extends from the subject to their first project, and another fixed length `WORKED_ON` relationship that connects the matched person to a project, with a smaller variable length path in between.

Running this query against our sample graph, and again taking Sarah as the subject of the query, if we look for colleagues and colleagues-of-colleagues who have interests in Java, travel or medicine, we get the following results:

```
+-------------------------------------------------+
| name      | score | interests         | distance |
+-------------------------------------------------+
| "Arnold"  | 2     | ["Travel","Java"] | 2        |
| "Charlie" | 1     | ["Medicine"]      | 1        |
+-------------------------------------------------+
2 rows
0 ms
```

Note that the results are ordered by `score`, not `distance`. Arnold has two out of the three interests, and therefore scores higher than Charlie, who only has one, even though he is at two removes from Sarah, whereas Charlie has worked directly with Sarah.

This is the portion of the graph that was traversed and matched to generate these results:

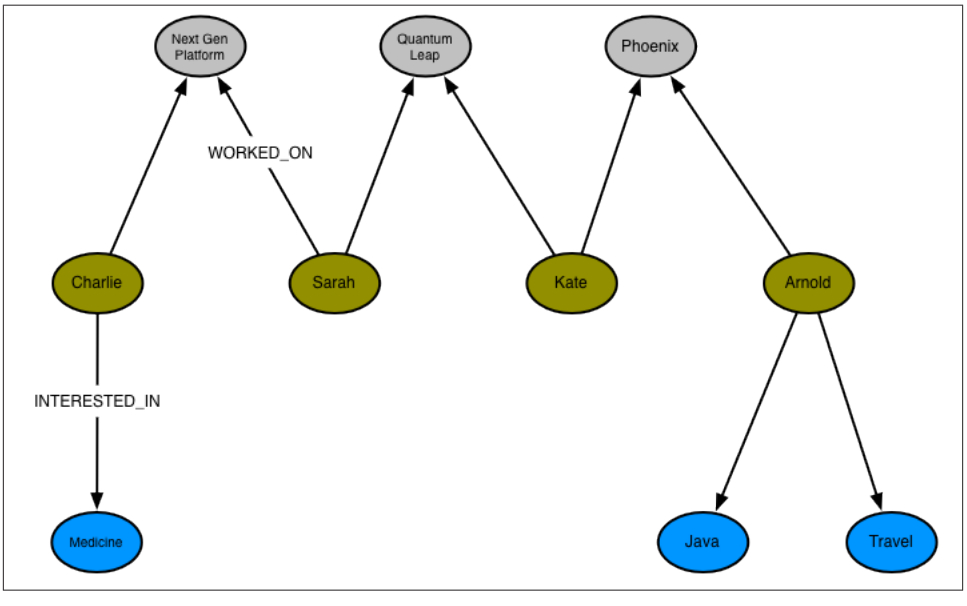


Figure 7-5. Finding people with particular interests

Let's take a moment to understand how this query executes in more detail. Figure 7-6 shows three stages in the execution of the query. The first stage shows each of the paths as they are matched by the `MATCH` and `WHERE` clauses. As you can see, there is one redundant path: `Charlie` is matched directly, through `Next Gen Platform`, and indirectly, by way of `Quantum Leap` and `Emily`. The second stage represents the filtering that takes place in the `WITH` clause. Here we emit triples comprising the matched person, the matched interest, and the length of the shortest path from the subject through the matched person to their interest. The third stage represents the `RETURN` clause, wherein we aggregate the results on behalf of each matched person, and calculate their score and distance from the subject.

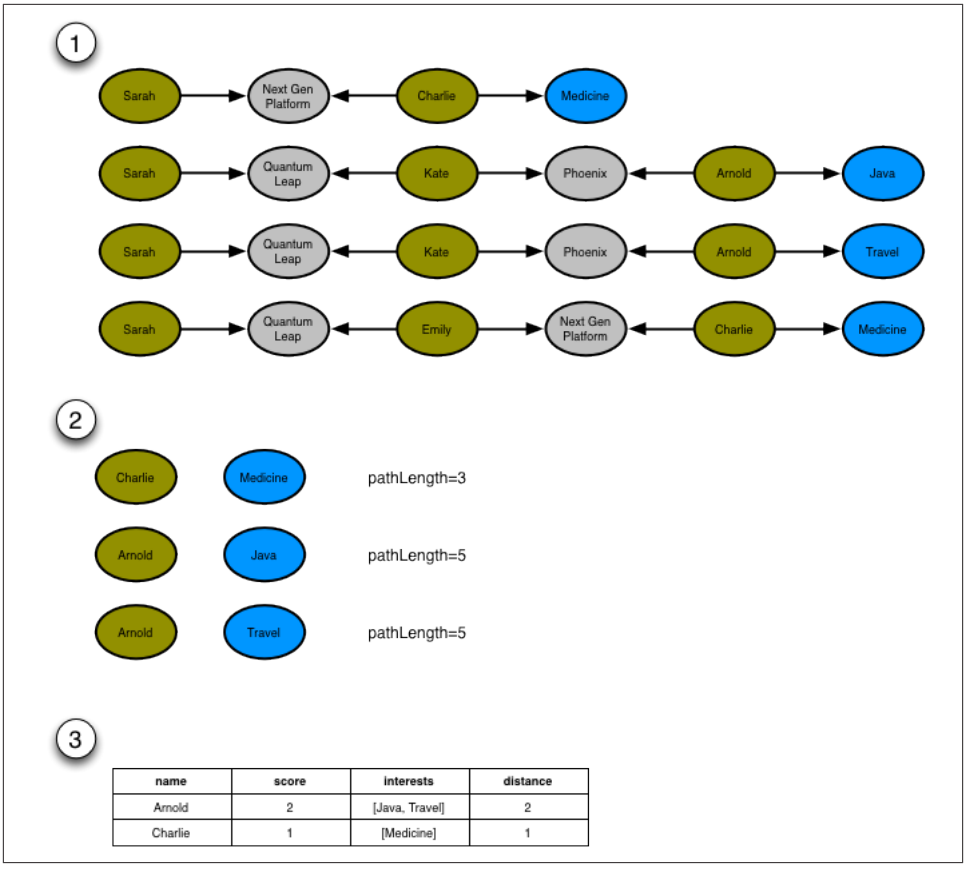


Figure 7-6. Query pipeline

Adding `WORKED_WITH` relationships

The query for finding colleagues and colleagues-of-colleagues with particular interests is the one most frequently executed on SocNet's site, and the success of the site depends in large part on its performance. The query uses pairs of `WORKED_ON` relationships (for example, `(Sarah)-[:WORKED_ON]->(Next Gen Platform)<-[:WORKED_ON]-(Charlie)`) to infer that users have worked with one another. While reasonably performant, this is nonetheless inefficient, in that it requires traversing two explicit relationships to infer the presence of a single implicit relationship.

To eliminate this inefficiency, SocNet now pre-compute `WORKED_WITH` relationships, thereby enriching the data and providing shorter paths for these performance-critical access patterns. This is a common optimisation strategy: introducing a direct relationship between two nodes that would otherwise be connected only by way of intermediaries. Unlike the optimisation strategies we employ in the relational world, however,

which typically involve denormalising and thereby compromising a high-fidelity model, this is not an either/or issue: either the high-fidelity structure, or the high performance compromise. With the graph we retain the original high-fidelity graph structure while at the same time enriching it with new elements catering to different access patterns.

In terms of the SocNet domain, `WORKED_WITH` is a bi-directional relationship. In the graph, however, it is implemented using a uni-directional relationship. While a relationship's direction can often add useful semantics to its definition, in this instance the direction is meaningless. This isn't a significant issue, so long as any queries that operate with `WORKED_WITH` relationships ignore the relationship direction.

Calculating a user's `WORKED_WITH` relationships and adding them to the graph isn't difficult, nor is it particularly expensive in terms of resource consumption. It can, however, add milliseconds to any end-user interactions that update a user's profile with new project information, so SocNet have decided to perform this operation asynchronously to end-user activities. Whenever a user changes their project history, SocNet add a job that recalculates that user's `WORKED_WITH` relationships to a queue. A single writer thread polls this queue and executes the jobs using the following Cypher statement:

```
START subject = node:user(name={name})
MATCH subject-[:WORKED_ON]->()<-[:WORKED_ON]-person
WHERE NOT(subject-[:WORKED_WITH]-person)
WITH DISTINCT subject, person
CREATE UNIQUE subject-[:WORKED_WITH]-person
RETURN subject.name AS startName, person.name AS endName
```

This is what our sample graph looks like once it has been enriched with `WORKED_WITH` relationships:

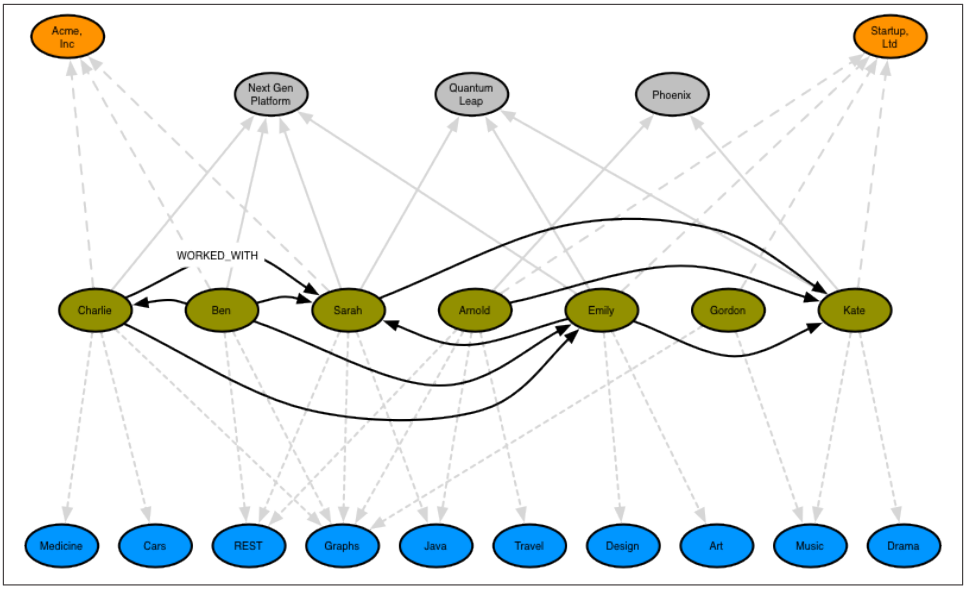


Figure 7-7. SocNet graph enriched with `WORKED_WITH` relationships

Using the enriched graph, SocNet now finds colleagues and colleagues-of-colleagues with particular interests using a slightly simpler version of the query we looked at above:

```
START subject=node:user(name={name})
MATCH p=subject-[:WORKED_WITH*0..1]-()-[:WORKED_WITH]-person-[:INTERESTED_IN]->interest
WHERE person<>subject AND interest.name IN {interests}
WITH person, interest, MIN(LENGTH(p)) as pathLength
RETURN person.name AS name,
       COUNT(interest) AS score,
       COLLECT(interest.name) AS interests,
       (pathLength - 1) AS distance
ORDER BY score DESC
LIMIT {resultLimit}
```

Access Control

SaasNet is an international telecommunications services company, with millions of domestic and business users subscribed to its products and services. For several years, it has offered its largest business customers the ability to self-service their accounts. Using a browser-based application, administrators within each of these customer organisations can add and remove services on behalf of their employees. To ensure that users and administrators see and change only those parts of the organisation and the products and services they are entitled to manage, the application employs a complex access control system, which assigns privileges to many millions of users across tens of millions of product and service instances.

SaasNet has decided to replace the existing access control system with a graph database solution. There are two drivers here: performance and business responsiveness.

Performance issues have dogged SaasNet's service-service application for several years. The original system is based on a relational database, which uses recursive joins to model complex organisational structures and product hierarchies, and stored procedures to implement the access control business logic. Because of the join-intensive nature of the data model, many of the most important queries are unacceptably slow: for an administrator in a large company, generating a view of the things that administrator can manage takes many minutes. This creates a very poor user experience, and hampers the revenue-generating capabilities of the self-service offering.

The performance issues that affect the original application suggest it is no longer fit for today's needs, never mind tomorrow's. SaasNet has ambitious plans to move into new regions and markets, effectively increasing its customer base by an order of magnitude. The existing solution clearly cannot accommodate the needs of this new strategy. A graph database solution, in contrast, offers the performance, scalability and adaptiveness necessary for dealing with a rapidly changing market.

SaasNet data model

Figure 7-8 shows a sample of the SaasNet data model. The model comprises two hierarchies. In the first hierarchy, administrators within each customer organization are assigned to groups; these groups are then accorded various permissions against that organization's organizational structure:

- `ALLOWED_INHERIT` connects an administrator group to an organizational unit, thereby allowing administrators within that group to manage the organizational unit. This permission is inherited by children of the parent organizational unit. We see na example of inhereited permissions in the SaasNet example data model in the relationships between `Group 1` and `Acme`, and the child of `Acme`, `Spinoff`. `Group 1` is connected to `Acme` using an `ALLOWED_INHERIT` relationship. `Ben`, as a member of `Group 1`, can manage employees both of `Acme` *and* `Spinoff` thanks to this `ALLOWED_INHERIT` relationship.
- `ALLOWED_DO_NOT_INHERIT` connects an administrator group to an organizational unit in a way that allows administrators within that group to manage the organizational unit, but not any of its children. `Sarah`, as a member of `Group 2`, can administer `Acme`, but not its child `Spinoff`, because `Group 2` is connected to `Acme` by an `ALLOWED_DO_NOT_INHERIT` relationship, not an `ALLOWED_INHERIT` relationship.
- `DENIED` forbids administrators from accessing an organizational unit. This permission is inherited by children of the parent organizational unit. In the SaasNet diagram, this is best illustrated by `Liz` and her permissions with respect to `Big Co`, `Acquired Ltd`, `Subsidiary` and `One-Map Shop`. As a result of her membership of

`Group 4` and its `ALLOWED_INHERIT` permission on `Big Co`, `Liz` can manage `Big Co`. But despite this being an inheritable relationship, `Liz` cannot manage `Acquired Ltd` or `Subsidiary`; the reason being, `Group 5`, of which `Liz` is a member, is `DENIED` access to `Acquired Ltd` and its children (which includes `Subsidiary`). `Liz` can, however, manage `One-Map Shop`, thanks to an `ALLOWED_DO_NOT_INHERIT` permission granted to `Group 6`, the last group to which `Liz` belongs.

`DENIED` takes precedence over `ALLOWED_INHERIT`, but is subordinate to `ALLOWED_DO_NOT_INHERIT`. Therefore, if an administrator is connected to a company by way of `ALLOWED_DO_NOT_INHERIT` and `DENIED`, `ALLOWED_DO_NOT_INHERIT` prevails.

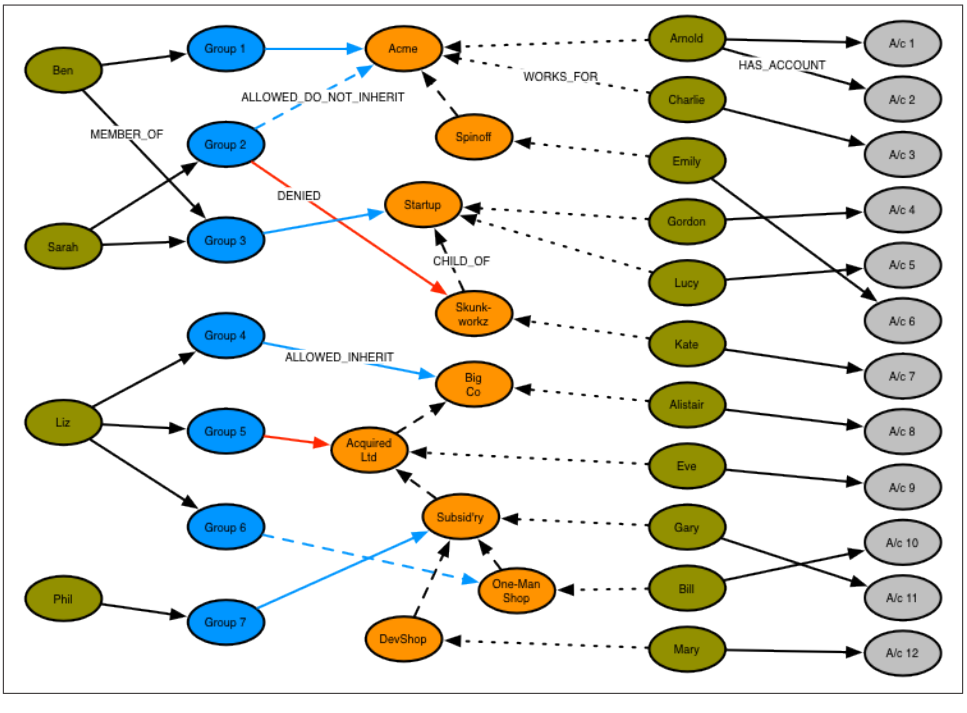


Figure 7-8. Access control graph

Fine-grained relationships, or relationships with properties?

You'll notice that the SaasNet access control data model uses fine-grained relationships (`ALLOWED_INHERIT`, `ALLOWED_DO_NOT_INHERIT` and `DENIED`) rather than a single relationship type qualified by properties—something like `PERMISSION` with `allowed` and `inherited` boolean properties. Why is this? And when would you choose between these two options?

SaasNet performance tested both approaches and determined that the fine-grained, property-free approach was nearly twice as fast as the one using properties. This should come as no surprise. Relationships are the royal road into the graph; differentiating by relationship type is the best way of eliminating large swathes of the graph from a traversal. Using one or more property values to decide whether or not to follow a relationship incurs extra IO—because the properties reside in a separate store file from the relationships—the first time those properties are accessed (after that, they're cached).

Consider using fine-grained relationships whenever you have a closed set of relationship types. Here, SaasNet has a closed set of permissions, ideal for representing as fine-grained relationships. In contrast, weightings—as required by a shortest weighted path algorithm—rarely comprise a closed set, and these are usually best represented as properties on relationships.

Sometimes, however, you have a closed set of relationships where in some traversals you want to follow specific kinds of relationship within that set, but in others you want to follow all of them, irrespective of type. Addresses are a good example. Following the closed-set principle, we might choose to create `HOME_ADDRESS`, `WORK_ADDRESS` and `DELIVERY_ADDRESS` relationships. This allows us to follow specific kinds of address relationship (`DELIVERY_ADDRESS`, for example) while ignoring all the rest. But what do we do if we want to find *all* addresses for a user? There are a couple of options here. First, we can encode knowledge of all the different relationship types in the relevant `MATCH` clauses: e.g. `MATCH user-[:HOME_ADDRESS|WORK_ADDRESS|DELIVERY_ADDRESS]->address`. This, however, quickly becomes unwieldy when there are lots of different kinds of relationship. Alternatively, we can add a more generic `ADDRESS` relationship to our model, in addition to the fine-grained relationships. Every node representing an address is then connected to a user using two relationships: a fined-grained relationship (e.g. `DELIVERY_ADDRESS`) *and* the more generic `ADDRESS` relationship.

As we discuss in "Describe the Model in Terms of Your Application's Needs" in Chapter 6, the key here is to let the questions you want to ask of your data guide the kinds of relationship you introduce into the model.

Finding all accessible resources for an administrator

The SaasNet application uses many different Cypher queries; we'll look at just a few of them here.

First up is the ability to find all resources accessible by a particular administrator. Whenever an onsite administrator logs in to the system, they are presented with a browser-based list of all the employees and employee accounts they can administer. This list is generated based on the results returned by the following query:

```
START admin=node:administrator(name={administratorName})
MATCH paths=admin-[:MEMBER_OF]->()-[:ALLOWED_INHERIT]->()<-[:CHILD_OF*0..3]-company
      <-[:WORKS_FOR]-employee-[:HAS_ACCOUNT]->account
```

```
WHERE NOT (admin-[:MEMBER_OF]->()-[:DENIED]->()<-[:CHILD_OF*0..3]-company)
RETURN employee.name AS employee, account.name AS account
UNION
START admin=node:administrator(name={administratorName})
MATCH paths=admin-[:MEMBER_OF]->()-[:ALLOWED_DO_NOT_INHERIT]->()
      <-[:WORKS_FOR]-employee-[:HAS_ACCOUNT]->account
RETURN employee.name AS employee, account.name AS account
```

This query sets the template for all the other queries we'll be looking at in this section, in that it comprises two separate queries joined by a `UNION` operator. The query before the `UNION` operator handles `ALLOWED_INHERIT` relationships qualified by any `DENIED` relationships; the query following the `UNION` operator handles any `ALLOWED_DO_NOT_INHERIT` permissions. This pattern, `ALLOWED_INHERIT` minus `DENIED`, followed by `ALLOWED_DO_NOT_INHERIT`, is repeated in all of the access control example queries that follow.

The first query here, the one before the `UNION` operator, can be broken down as follows:

- The `START` clause finds the logged-in administrator in the `administrator` index, and binds the result to the `admin` identifier.
- The `MATCH` clause matches all the groups to which this administrator belongs, and from these groups, all the parent companies connected by way of an `ALLOWED_INHERIT` relationship. The `MATCH` then uses a variable length path (`[:CHILD_OF0..3]`) to discover children of these parent companies, and thereafter the employees and accounts associated with all matched companies (whether parent company or child). At this point, the query has matched all companies, employees and accounts accessible by way of `ALLOWED_INHERIT` relationships.
- The `WHERE` clause eliminates matches where the `company` bound in the `MATCH` clause, or any of its parents, is connected by way of `DENIED` relationship to the administrator's groups. This `WHERE` clause is invoked for each match; if there is a `DENIED` relationship anywhere between the `admin` node and the `company` node bound by the match, that match is eliminated.
- The `RETURN` clause creates a projection of the matched data in the form of a list of employee names and accounts.

The second query here, following the `UNION` operator, is a little simpler:

- The `MATCH` clause simply matches companies (plus employees and accounts) which are directly connected to an administrator's groups by way of an `ALLOWED_DO_NOT_INHERIT` relationship.

The UNION operator joins the results of these two queries together, eliminating any duplicates. Note that the RETURN clause in each query must contain the same projection of the results; in other words, the column names in the results must match.

Cypher supports both UNION and UNION ALL operators. UNION eliminates duplicate results from the final result set, whereas UNION ALL includes any duplicates.

Figure 7-9 shows how this query matches all accessible resources for Sarah in the sample SaasNet graph. Note how, because of the DENIED relationship from Group 2 to Skunk workz, Sarah cannot administer Kate and Account 7.

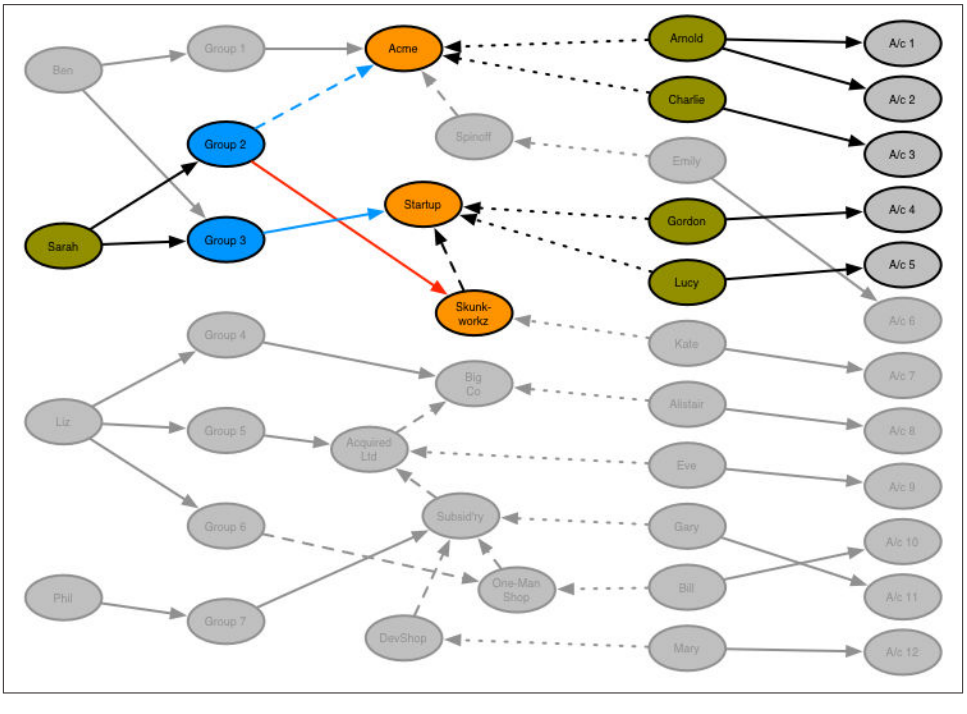


Figure 7-9. Finding all accessible resources for a user

Determining whether an administrator has access to a resource

The query we've just looked at returned a list of employees and accounts that an administrator can manage. In a Web application, each of these resources (employee, account) is accessible through its own URI. Given a friendly URI (e.g. *http://saasnet/*

accounts/5436), what's to stop someone from hacking a URI and gaining illegal access to an account?

What's needed is a query that will determine whether an administrator has access toa specific resource. This is that query:

```
START admin=node:administrator(name={adminName}),
      company=node:company(resourceName={resourceName})
MATCH p=admin-[:MEMBER_OF]->()-[:ALLOWED_INHERIT]->()<-[:CHILD_OF*0..3]-company
WHERE NOT (admin-[:MEMBER_OF]->()-[:DENIED]->()<-[:CHILD_OF*0..3]-company)
RETURN COUNT(p) AS accessCount
UNION
START admin=node:administrator(name={adminName}),
      company=node:company(resourceName={resourceName})
MATCH p=admin-[:MEMBER_OF]->()-[:ALLOWED_DO_NOT_INHERIT]->company
RETURN COUNT(p) AS accessCount
```

This query works by determining whether an administrator has access to the company to which an employee or an account belongs. How do we identify to which company an employee or account belongs? Through clever use of indexes.

In the SaasNet data model, companies are indexed both by their name, and by the names of their employees and employee accounts. Given a company name, employee name, or account name, we can, therefore, lookup the relevant company node in the company index.

Given that bit of insight, we can see that this resource authorization check query is similar to the query for finding all companies, employees and accounts, but for several small differences:

- `company` is bound in the `START` clause, not the `MATCH` clause. Using the indexing strategy described above, we lookup the node representing the relevant company based on the name of the resource to be authorized—whether employee or account.
- We don't match any further than company. Since `company` has already been bound based on an employee or account name, there's no need to drill further into the graph to match that employee or account.
- The `RETURN` clauses for the queries before and after the `UNION` operator return a count of the number of matches. For an administrator to have access to a resource, one or other of these `accessCount` values must be greater than 0.

The `WHERE` clause in the query before the `UNION` operator once again eliminates matches where the `company` in question is connected to one of the administrator's groups by way of a `DENIED` relationship.

Because the `UNION` operator eliminates duplicate results, the overall result set for this query can contain either one or two values. The client-side logic for determining whether an administrator has access to a resource looks like this in Java:

```
private boolean isAuthorized( ExecutionResult result )
{
    Iterator<Long> accessCountIterator = result.columnAs( "accessCount" );
    while ( accessCountIterator.hasNext() )
    {
        if (accessCountIterator.next() > 0L)
        {
            return true;
        }
    }
    return false;
}
```

Finding administrators for an account

The previous two queries represent "top-down" views of the graph. The last SaasNet query we'll discuss here provides a "bottom-up" view of the data. Given a resource—an employee or account—which administrators can manage it? Here's the query:

```
START resource=node:resource(name={resourceName})
MATCH p=resource-[:WORKS_FOR|HAS_ACCOUNT*1..2]-company
        -[:CHILD_OF*0..3]->()<-[:ALLOWED_INHERIT]-()<-[:MEMBER_OF]-admin
WHERE NOT (admin-[:MEMBER_OF]->()-[:DENIED]->()<-[:CHILD_OF*0..3]-company)
RETURN admin.name AS admin
UNION
START resource=node:resource(name={resourceName})
MATCH p=resource-[:WORKS_FOR|HAS_ACCOUNT*1..2]-company
        <-[:ALLOWED_DO_NOT_INHERIT]-()<-[:MEMBER_OF]-admin
RETURN admin.name AS admin
```

As before, the query consists of two independent queries joined by a `UNION` operator. Of particular note are the following clauses:

- The `START` clauses use a `resource` index, a named index for both employees and accounts.
- The `MATCH` clauses contain a variable length path expression that uses the `|` operator to specify a path that is one or two relationships deep, and whose relationship types comprise `WORKS_FOR` and `HAS_ACCOUNT`. This expression accomodates the fact that the subject of the query may be either an employee or an account.

Figure 7-10 shows the portions of the graph matched by the query when asked to find the administrators for `Account 10`.

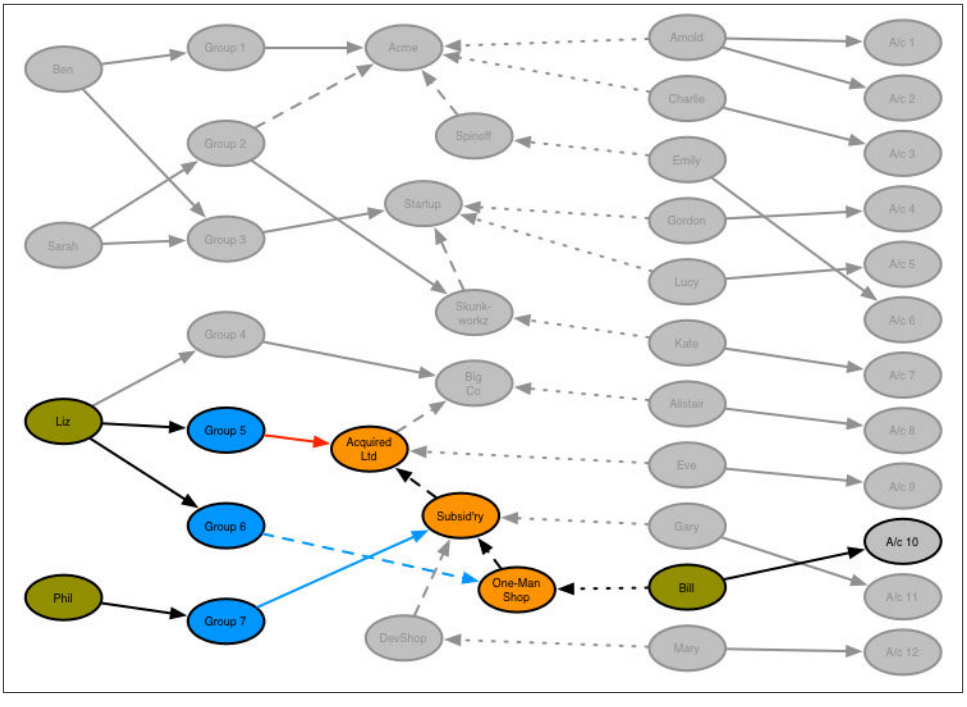


Figure 7-10. Finding an administrator for a specific account

Logistics

LogisticsNet is a large courier company whose domestic operation delivers millions of parcels to over 30 million addresses each day. In recent years, as a result of the rise in internet shopping, the number of parcels has increased significantly. Amazon and eBay deliveries now account for over half the parcels routed and delivered by LogisticsNet each day.

With parcel volumes continuing to grow, and facing strong competition from other courier services, LogisticsNet has begun a large change programme to upgrade all aspects of its parcel network, including buildings, equipment, systems and processes.

One of the most important and time-critical components in the parcel network is the route calculation engine. Between one and three thousand parcels enter the network each second. As parcels enter the network they are mechanically sorted according to their destination. To maintain a steady flow during this process the engine must calculate a parcel's route before it reaches a point where the sorting equipment has to make a choice, which happens only seconds after the parcel has entered the network—hence the strict time requirements on the engine.

As well as being time critical, the engine must also be time sensitive. Parcel routes change throughout the year, with more trucks, delivery people and collections over the Christmas period than during the summer, for example. The engine must, therefore, apply its calculations using only those routes that are available for a particular period.

On top of accommodating different routes and levels of parcel traffic, the new parcel network must also allow for far more significant change and evolution. The platform that LogisticsNet develops today will form the business-critical basis of its operations for the next ten years or more. During that time, the company anticipates that large portions of the network—including equipment, premises, and transport routes—will change considerably to match changes in the business environment. The data model underlying the route calculation engine must, therefore, allow for rapid and significant schema evolution.

LogisticsNet data model

Figure 7-11 shows a simple example of the LogisticsNet parcel network. The network comprises parcel centers, which are connected to delivery bases, each of which covers several delivery areas; these delivery areas in turn are subdivided into delivery segments covering many delivery units. There are around 25 national parcel centers and roughly two million delivery units (corresponding to postal or zip codes).

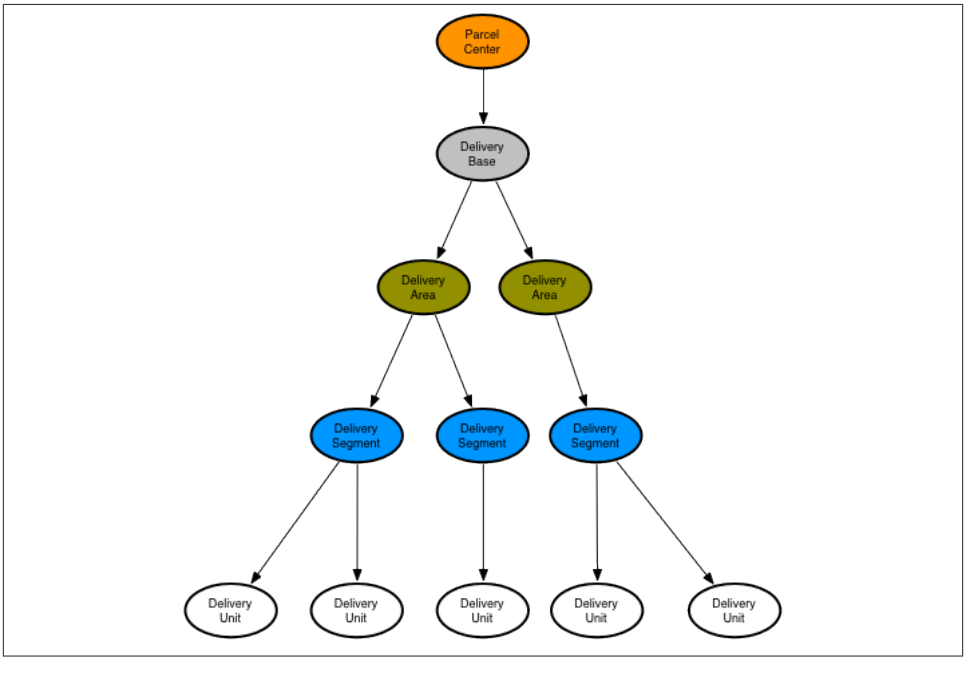


Figure 7-11. Elements in the LogisticsNet network

Over time, the delivery routes change. Figure 7-12, Figure 7-13 and Figure 7-14 show three distinct delivery periods. For any given period there is at most one route between a delivery base and any particular delivery area or segment. In contrast, there may be multiple routes between delivery bases and parcel centers. For any given point in time, therefore, the lower portions of the graph—the individual subgraphs that depend from each delivery base—comprise simple tree structures, whereas the upper portions of the graph—made up of delivery bases and parcel centers—are more complexly structured.

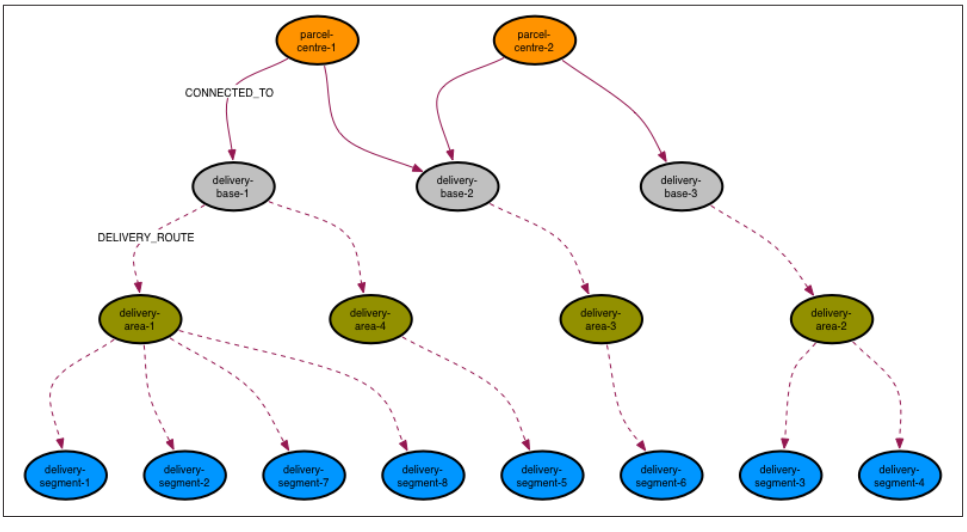


Figure 7-12. Structure of the delivery network for Period 1

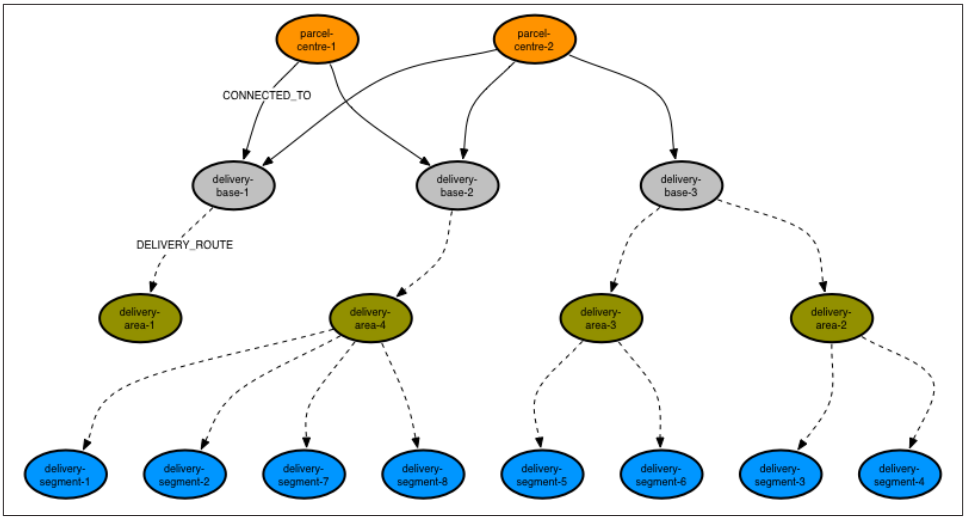


Figure 7-13. Structure of the delivery network for Period 2

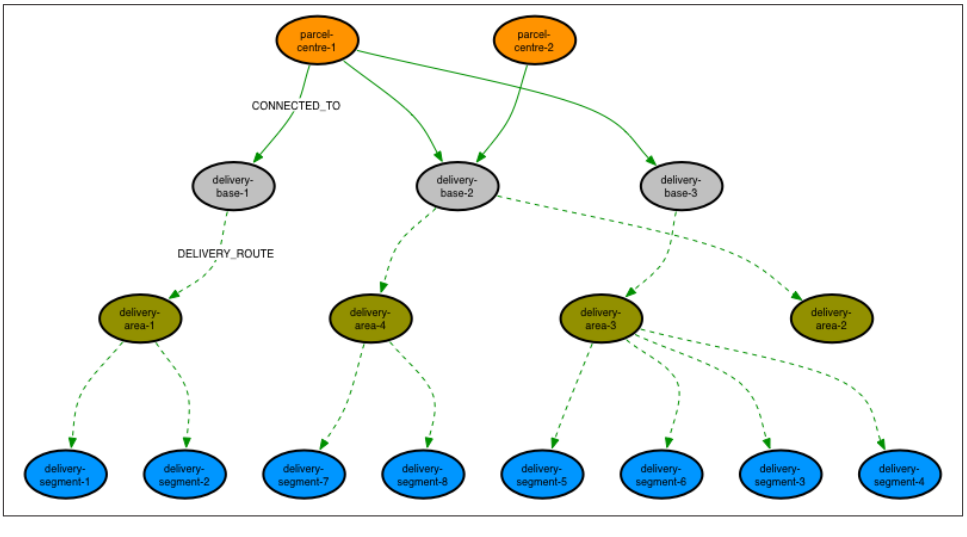


Figure 7-14. Structure of the delivery network for Period 3

You'll notice that delivery units are not included in the production data. This is because each delivery unit is always associated with the same delivery segments, irrespective of the period. Because of this invariant, it is possible to index each delivery segment by its many delivery units. To calculate the route to a particular delivery unit, the system need only actually calculate the route to its associated delivery segment, which can be recovered from the index using the delivery unit as a key. This optimisation helps both reduce

the size of the production graph, and reduce the number of traversals needed to calculate a route.

The production database contains the details of many different delivery periods, as shown in Figure 7-15. This multi-period graph demonstrates some of the overall complexity present in the data.

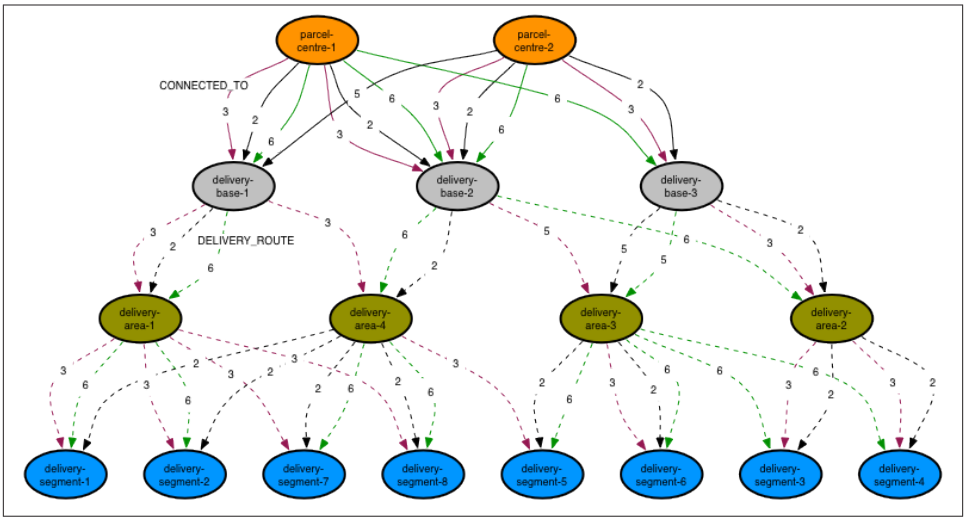


Figure 7-15. Sample of the LogisticsNet network

In the production data, nodes are connected by multiple relationships, each of which is timestamped with a `start_date` and `end_date` property, whose long value represents the number of milliseconds since `1970-01-01T00:00:00Z`. Relationships are of two types: `CONNECTED_TO`, which connect parcel centers and delivery bases, and `DELIVERY_ROUTE`, which connect delivery bases to delivery areas, and delivery areas to delivery segments. These two different types of relationship effectively partition the graph into its upper and lower parts, which, as we'll see shortly provides for efficient traversals. Figure 7-16 shows three of the timestamped `CONNECTED_TO` relationships connecting a parcel center to a delivery base:

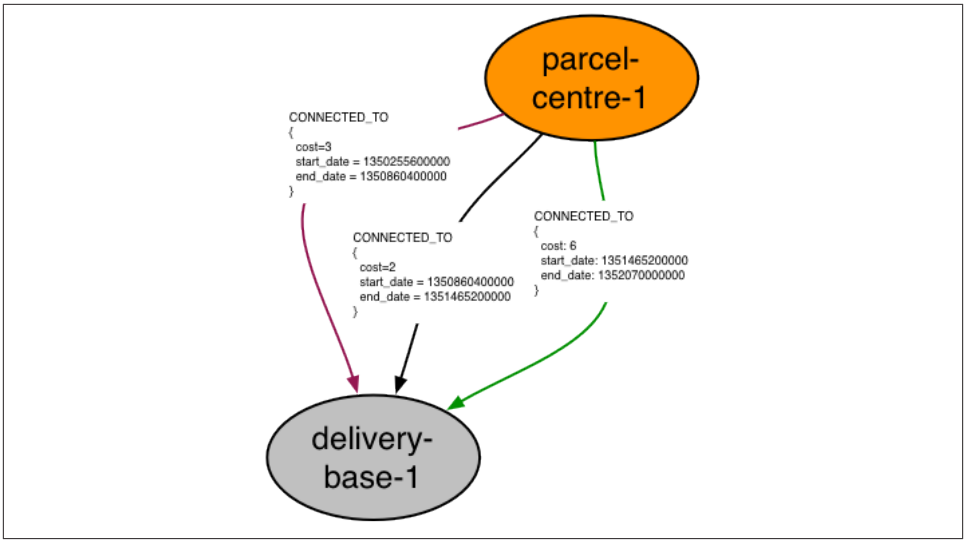


Figure 7-16. Timestamp properties on relationships

Route calculation

As described in the previous section, the `CONNECTED_TO` and `DELIVERY_ROUTE` relationships partition the graph into *upper* and *lower* parts, with the upper parts made up of complexly connected parcel centers and delivery centers, the lower parts of delivery bases, delivery areas and delivery segments organised—for any given period—in simple tree structures.

Route calculations involve finding the cheapest route between two locations in the lower portions of the graph. The starting location is typically a delivery segment or delivery area, whereas the end location is always a delivery segment (indexed, as we discussed earlier, by its delivery units). Irrespective of the start and end locations, the calculated route must go via at least one parcel center in the upper part of the graph.

In terms of traversing the graph, a calculation can be split into three legs. Legs one and two, shown in Figure 7-17, work their way upwards from the start and end locations respectively, with each terminating at a delivery center. Because there is at most one route between any two elements in the lower portion of the graph for any given delivery period, traversing from one element to the next is simply a matter of finding an incoming `DELIVERY ROUTE` relationship whose interval timestamps *encompass* the current delivery period. By following these relationships, the traversals for legs one and two navigate a pair of tree structures rooted at two different delivery centers. These two delivery centers then form the start and and end locations for the third leg, which crosses the upper portion of the graph.

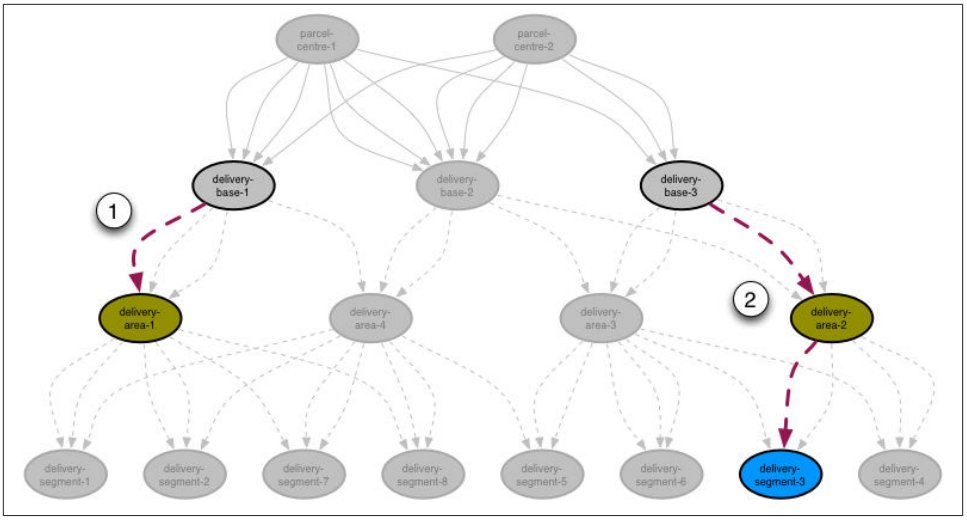


Figure 7-17. Shortest path to delivery bases from start and end points

As with legs one and two, the traversal for leg three, as shown in Figure 7-18, looks for relationships—this time, `CONNECTED_TO` relationships—whose timestamps *encompass* the current delivery period. Even with this time filtering in place, however, there are for any given period potentially several routes between any two delivery centers in the upper portion of the graph; thus the third leg traversal must sum the cost of each route, and select the cheapest, making this a *shortest weighted path* calculation.

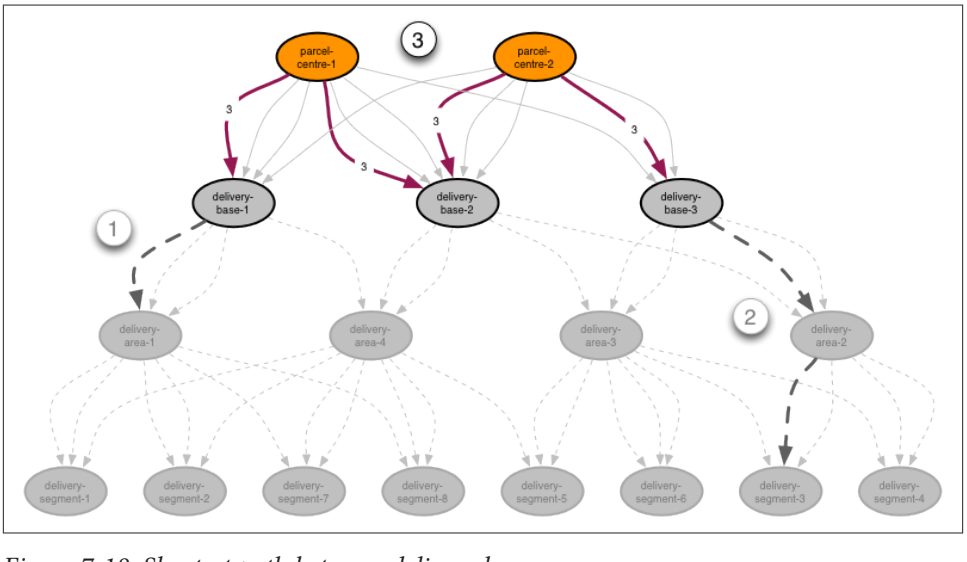


Figure 7-18. Shortest path between delivery bases

To complete the calculation, we need then simply add the paths for legs one, three and two, which gives the full path from the start to the end location.

Finding the shortest delivery route using Cypher

The Cypher query to implement the parcel route calculation engine is as follows:

```
START s=node:location(name={startLocation}),
      e=node:location(name={endLocation})
MATCH p1 = s<-[:DELIVERY_ROUTE*1..2]-db1
WHERE ALL(r in relationships(p1)
          WHERE r.start_date <= {intervalStart}
          AND r.end_date >= {intervalEnd})
WITH  e, p1, db1
MATCH p2 = db2-[:DELIVERY_ROUTE*1..2]->e
WHERE ALL(r in relationships(p2)
          WHERE r.start_date <= {intervalStart}
          AND r.end_date >= {intervalEnd})
WITH  db1, db2, p1, p2
MATCH p3 = db1<-[:CONNECTED_TO]-()-[:CONNECTED_TO*1..3]-db2
WHERE ALL(r in relationships(p3)
          WHERE r.start_date <= {intervalStart}
          AND r.end_date >= {intervalEnd})
WITH  p1, p2, p3,
      REDUCE(weight=0, r in relationships(p3) : weight+r.cost) AS score
      ORDER BY score ASC
      LIMIT 1
RETURN (nodes(p1) + tail(nodes(p3)) + tail(nodes(p2))) AS n
```

At first glance this query appears quite complex. It is, however, made up of four simpler queries joined together with `WITH` clauses. We'll look at each of these subqueries in turn.

Here's the first subquery:

```
START s=node:location(name={startLocation}),
      e=node:location(name={endLocation})
MATCH p1 = s<-[:DELIVERY_ROUTE*1..2]-db1
WHERE ALL(r in relationships(p1)
          WHERE r.start_date <= {intervalStart}
          AND r.end_date >= {intervalEnd})
```

This query calculates the first leg of the overall route. It can be broken down as follows:

- The `START` clause finds the start and end locations in an index, binding them to the `s` and `e` identifiers respectively. (Remember, the `endLocation` lookup term describes a delivery unit, but the node returned from the `location` index represents a delivery segment.)
- The `MATCH` clause finds the route from the start location, `s`, to a delivery base using a directed, variable length `DELIVERY_ROUTE` path. This path is then bound to the identifier `p1`. Because delivery bases are always the root nodes of `DELIVERY_ROUTE`

trees, and therefore have no incoming DELIVERY_ROUTE relationships, we can be confident that the db1 node at the end of this variable length path represents a delivery base and not some other parcel network element.

- The WHERE clause applies additional constraints to the path p1, ensuring that we only match DELIVERY_ROUTE relationships whose start_date and end_date properties encompass the supplied delivery period.

The second subquery calculates the second leg of the route, which comprises the path from the end location to another delivery center elsewhere in the network. This query is very similar to the first:

```
WITH  e, p1, db1
MATCH p2 = db2-[:DELIVERY_ROUTE*1..2]->e
WHERE ALL(r in relationships(p2)
          WHERE r.start_date <= {intervalStart}
          AND r.end_date >= {intervalEnd})
```

The WITH clause here chains the first subquery to the second, piping the end location and the first leg's path and delivery base to the second subquery. The second subquery uses only the end location, e, in its MATCH clause; the rest is provided so that it can be piped to subsequent queries.

The third subquery identifies *all* candidate paths for the third leg of the route, as follows:

```
WITH  db1, db2, p1, p2
MATCH p3 = db1<-[:CONNECTED_TO]-()-[:CONNECTED_TO*1..3]-db2
WHERE ALL(r in relationships(p3)
          WHERE r.start_date <= {intervalStart}
          AND r.end_date >= {intervalEnd})
```

This subquery is broken down as follows:

- The WITH clause chains this subquery to the previous one, piping the delivery bases and paths identified in legs one and two to the current query.
- The MATCH clause identifies *all* paths between the first and second leg delivery bases, to a maximum depth of four, and binds them to the p3 identifier.
- The WHERE clause constrains the p3 paths to those whose start_date and end_date properties encompass the supplied delivery period.

The fourth and final subquery selects the shortest path for leg three, and then calculates the overall route:

```
WITH  p1, p2, p3,
      REDUCE(weight=0, r in relationships(p3) : weight+r.cost) AS score
      ORDER BY score ASC
      LIMIT 1
RETURN (nodes(p1) + tail(nodes(p3)) + tail(nodes(p2))) AS n
```

This subquery works as follows:

- The `WITH` clause pipes one or more triples, comprising `p1`, `p2` and `p3`, to the current query. There will be one triple for each of the paths matched by the third subquery, with each path being bound to `p3` in successive triples (the paths bound to `p1` and `p2` will stay the same, since the first and second subqueries matched only one path each). Each triple is accompanied by a `score` for the path bound to `p3` for that triple. This score is calculated using Cypher's `REDUCE` function, which for each triple sums the `cost` properties on the relationships in the path currently bound to `p3`. The triples are then ordered by this `score`, lowest first, and then limited to the first triple in the sorted list.
- The `RETURN` clause sums the nodes in the paths `p1`, `p3` and `p2` to produce the final results. The `tail` function drops the first node in each of paths `p3` and `p2`, since that node will already be present in the preceding path.

Implementing route calculation with the traversal framework

The time-critical nature of the route calculation coupled with the high throughput of the parcel network impose strict demands on the route calculation engine. As long as the individual query latencies are low enough, it's always possible to scale horizontally for increased throughput. The Cypher-based solution is fast, but with such high sustained throughput, every millisecond impacts the cluster footprint. For this reason, LogisticsNet adopted an alternative approach: calculating routes using the Java traversal framework.

A traversal-based implementation of the route calculation engine must solve two problems: finding shortest paths, and filtering paths based on time period. We'll look at filtering paths based on time period first.

Traversals should only follow relationships which are valid for the specified delivery period. In other words, as the traversal progresses through the graph, it should only be presented with relationships whose periods of validity, as defined by their `start_date` and `end_date` properties, contain the specified delivery period.

We implement this relationship filtering using a `PathExpander`. Given a path from a traversal's start node to the node where it is currently positioned, a `PathExpander`'s `expand()` method returns the relationships that can be used to traverse further. This method is called by the traversal framework each time the framework advances another node into the graph. If needed, the client can supply some initial state that flows along each branch; `expand()` can then use (and even change) this state in the course of deciding which relationships to return. The route calculator's `ValidPathExpander` implementation uses this branch state to supply the delivery period to the expander.

Here's the code for the `ValidPathExpander`:

```
private static class ValidPathExpander implements PathExpander<Interval>
{
  private final RelationshipType relationshipType;
  private final Direction direction;

  private ValidPathExpander( RelationshipType relationshipType, Direction direction )
  {
      this.relationshipType = relationshipType;
      this.direction = direction;
  }

  @Override
  public Iterable<Relationship> expand( Path path, BranchState<Interval> deliveryInterval )
  {
      List<Relationship> results = new ArrayList<Relationship>();
      for ( Relationship r : path.endNode().getRelationships( relationshipType, direction ) )
      {
          Interval relationshipInterval = new Interval(
                  (Long) r.getProperty( "start_date" ),
                  (Long) r.getProperty( "end_date" ) );
          if ( relationshipInterval.contains( deliveryInterval.getState() ) )
          {
              results.add( r );
          }
      }

      return results;
  }

  @Override
  public PathExpander<Interval> reverse()
  {
      return null;
  }
}
```

The `ValidPathExpander`'s constructor takes two arguments: a `relationshipType` and a `direction`. This allows the expander to be re-used for different types of relationship: in the case of the LogisticsNet graph, the expander will be used to filter both `CONNECTED_TO` and `DELIVERY_ROUTE` relationships.

The expander's `expand()` method takes the `path` to the current node, and the `deliveryInterval` as supplied by the client. Each time it is called, `expand()` iterates the relevant relationships on the current node (the current node is given by `path.endNode()`). For each relationship, the method then compares the relationship's interval with the delivery interval. If the relationship interval *contains* the delivery interval, the relationship is added to the results.

Having looked at the `ValidPathExpander`, we can now turn to the `ParcelRouteCalculator` itself. This class encapsulates all the logic necessary to calculate a route between the point where a parcel enters the network and the final delivery destination. It employs a similar strategy to the Cypher query we've already looked at: it works its way up the graph from both the start node and the end node in two separate traversals, until it finds a delivery base for each leg. It then performs a shortest weighted path search that joins these two delivery bases.

Here's the beginning of the `ParcelRouteCalculator` class:

```
public class ParcelRouteCalculator
{
    private static final PathExpander<Interval> DELIVERY_ROUTE_EXPANDER =
            new ValidPathExpander( withName( "DELIVERY_ROUTE" ), Direction.INCOMING );

    private static final PathExpander<Interval> CONNECTED_TO_EXPANDER =
            new ValidPathExpander( withName( "CONNECTED_TO" ), Direction.BOTH );

    private static final TraversalDescription TRAVERSAL = Traversal.description()
            .depthFirst()
            .evaluator( new Evaluator()
            {
                private final RelationshipType DELIVERY_ROUTE = withName( "DELIVERY_ROUTE");

                @Override
                public Evaluation evaluate( Path path )
                {
                    if ( !path.endNode()
                            .hasRelationship( DELIVERY_ROUTE, Direction.INCOMING )  )
                    {
                        return Evaluation.INCLUDE_AND_PRUNE;
                    }

                    return Evaluation.INCLUDE_AND_CONTINUE;
                }
            } );

    private static final CostEvaluator<Double> COST_EVALUATOR =
            CommonEvaluators.doubleCostEvaluator( "cost" );

    private final Index<Node> locationIndex;

    public ParcelRouteCalculator( GraphDatabaseService db )
    {
        this.locationIndex = db.index().forNodes( "location" );
    }

    ...

}
```

Here we define two expanders—one for DELIVERY_ROUTE relationships, another for CONNECTED_TO relationships—and the traversal that will find the two legs of our route. This traversal terminates whenever it encounters a node with no incoming DELIVERY_ROUTE relationships. Since each delivery base is at the root of a delivery route tree, we can infer that a node without any incoming DELIVERY_ROUTE relationships represents a delivery base in our graph.

The constructor for ParcelRouteCalculator accepts the current database instance. From this it obtains the location index, which it stores in a member variable.

Each route calculation engine maintains a single instance of this route calculator. This instance is capable of servicing multiple requests. For each route to be calculated, the client calls the calculator's calculateRoute() method, passing in the names of the start and end points, and the interval for which the route is to be calculated:

```
public Iterable<Node> calculateRoute( String start, String end, Interval interval )
{
    TraversalDescription deliveryBaseFinder = createDeliveryBaseFinder( interval );

    Collection<Node> upLeg = findRouteToDeliveryBase( start, deliveryBaseFinder );
    Collection<Node> downLeg = findRouteToDeliveryBase( end, deliveryBaseFinder );
    Collection<Node> topRoute = findRouteBetweenDeliveryBases(
            IteratorUtil.last( upLeg ),
            IteratorUtil.last( downLeg ),
            interval );

    return combineRoutes( upLeg, downLeg, topRoute );
}
```

calculateRoute() first obtains a deliveryBaseFinder for the specified interval, which it then uses to the find the routes for the two legs. Next, it finds the route between the delivery bases at the top of each leg, these being the last nodes in each leg's node collection. Finally, it combines these route to generate the final results.

The createDeliveryBaseFinder() helper method creates a traversal description configured with the supplied interval:

```
private TraversalDescription createDeliveryBaseFinder( Interval interval )
{
    return TRAVERSAL.expand( DELIVERY_ROUTE_EXPANDER,
            new InitialBranchState.State<Interval>( interval, interval ) );
}
```

This traversal description is built by expanding the ParcelRouteCalculator's static TRAVERSAL description using the DELIVERY_ROUTE_EXPANDER. The branch state for the expander is initialised at this point with the interval supplied by the client.

Properly configured with an interval, the traversal description is then supplied to `findRouteToDeliveryBase()`, which looks up a starting node in the location index, and then executes the traversal:

```
private Collection<Node> findRouteToDeliveryBase( String startPosition,
                                TraversalDescription deliveryBaseFinder )
{
    Node startNode = locationIndex.get( "name", startPosition ).getSingle();
    return IteratorUtil.asCollection( deliveryBaseFinder.traverse( startNode )
                                                        .nodes() );
}
```

That's the two legs taken care of. The last part of the calculation requires us to find the shortest path between the delivery bases at the top of each of the legs. `calculateRoute()` takes the last node from each leg's node collection, and supplies these two nodes together with the client-supplied interval to `findRouteBetweenDeliveryBases()`. Here's the implementation of `findRouteBetweenDeliveryBases()`:

```
private Collection<Node> findRouteBetweenDeliveryBases( Node deliveryBase1,
                                                        Node deliveryBase2,
                                                        Interval interval )
{
    PathFinder<WeightedPath> routeBetweenDeliveryBasesFinder = GraphAlgoFactory.dijkstra(
            CONNECTED_TO_EXPANDER,
            new InitialBranchState.State<Interval>( interval, interval ),
            COST_EVALUATOR );

    return IteratorUtil.asCollection( routeBetweenDeliveryBasesFinder
            .findSinglePath( deliveryBase1, deliveryBase2 )
            .nodes() );
}
```

Rather than use a traversal description to find the shortest route between two nodes, this method uses a shortest weighted path algorithm from Neo4j's graph algorithm library—in this instance, we're using the Dijkstra algorithm (see "Path-Finding with Dijkstra's Algorithm" in Chapter 8 for more details of the Dijkstra algorithm). This algorithm is configured with `ParcelRouteCalculator`'s static `CONNECTED_TO_EXPANDER`, which in turn is initialised with the client-supplied branch state interval. The algorithm is also configured with a cost evaluator (another static member), which simply identifies the property on a relationship representing that relationship's weight or cost. A call to `findSinglePath` on the Dijkstra path finder returns the shortest path between the two delivery bases.

That's the hard work done. All that remains to do is to join these routes to form the final results. This is relatively straightforward, the only wrinkle being that the down leg's node collection must be reversed before being added to the results (the leg was calculated from final destination upwards, whereas it should appear in the results delivery base downwards):

```
private Set<Node> combineRoutes( Collection<Node> upNodes,
                                 Collection<Node> downNodes,
                                 Collection<Node> topNodes )
{
    Set<Node> results = new LinkedHashSet<Node>();
    results.addAll( upNodes );
    results.addAll( topNodes );
    List<Node> downNodesList = new ArrayList<Node>( downNodes );
    Collections.reverse( downNodesList );
    results.addAll( downNodesList );
    return results;
}
```

CHAPTER 8

Predictive Analysis with Graph Theory

In this chapter we're going consider analytical techniques and algorithms for processing graph data. Both graph theory and graph algorithms are mature and well-understood fields of computing science and we'll demonstrate how both can can be used to mine sophisticated information from graph databases. While the reader with a background in computing science will no doubt recognize (and reminisce about) these algorithms and techniques, the discussion in this chapter is handled as humanely as possible — with virtually no mathematics — to encourage the curious reader to dive right in.

Depth- and Breadth-First Search

Before we can dive into higher-order analytical techniques we need to re-acquaint ourselves with the fundamental *breadth-first search* algorithm, which is used as the basis for iterating over an entire graph. Most of the queries we've seen throughout this book have been *depth-first* rather than *breadth-first* in nature. That is, they traverse outwards from a starting node to some end node before repeating a similar search down a different path from the same start node. Depth-first is a good strategy when we're trying to follow a path to discover discrete information goals.

Informed depth-first search

The classic depth-first algorithm search is said to be *uninformed* in that it simply searches a path until it finds the end of the graph before backtracking to the start node and trying a different path. However graph databases are semantically rich, so terminating a search along a particular branch (for example having found a node with no compatible outgoing relationships, or having traversed "far enough") can happen early resulting in typically lower execution times for such *informed* cases — exactly the kind of situation for which we optimized our Cypher matches and traversals in previous chapters.

Though we've used depth-first search as our underlying strategy for general graph traversals, many interesting "higher order" algorithms traverse the (entire) graph in a *breadth-first* manner. That is they explore the graph layer by layer exploring the nearest neighbor nodes of a starting node before the next nearest and so on until the search terminates.

Breadth-first search by contrast explores the graph one layer at a time, by first visiting each node at depth 1 from the start node, then depth 2, then depth 3 and so on potentially until the entire graph has been visited. This progression is easily visualized starting at the node labelled `0` (for *origin*) and progressing outwards a layer at a time as per Figure 8-1.

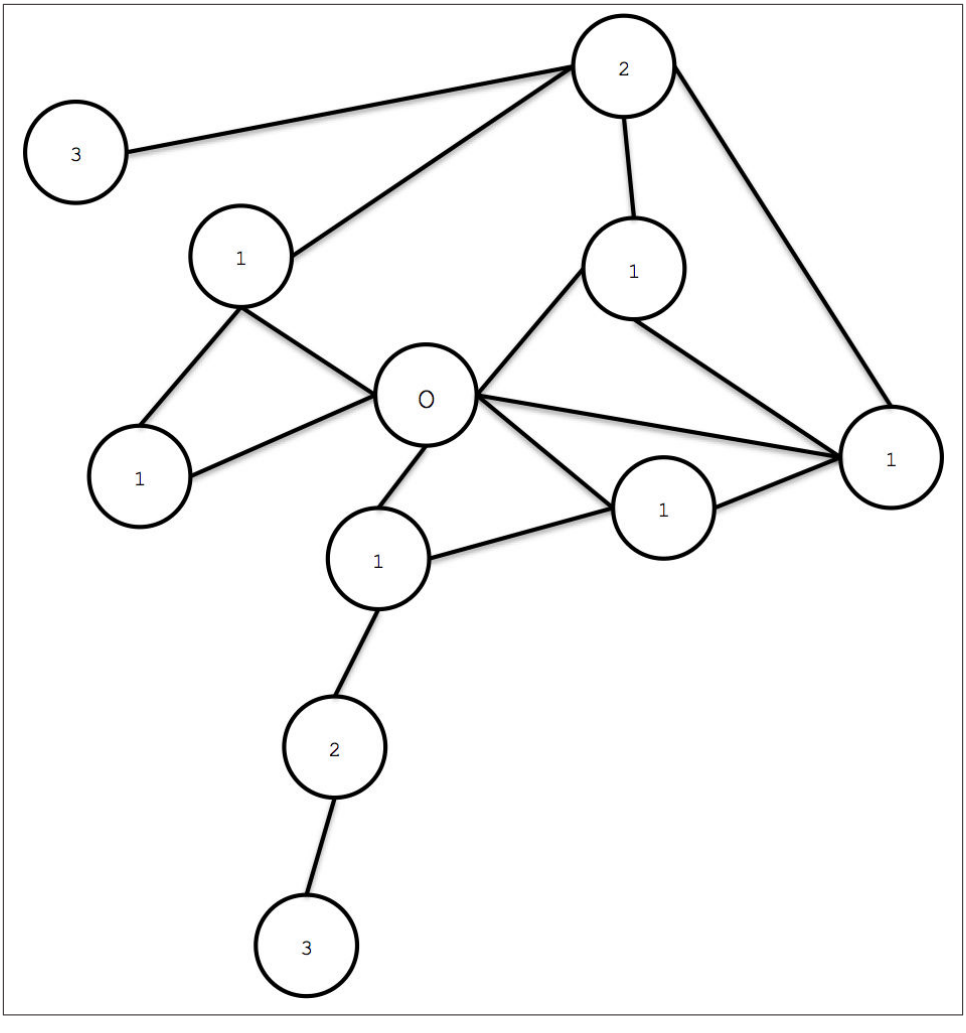


Figure 8-1. The progression of a breadth-first search

Where the search terminates depends on the algorithm being executed (since most useful algorithms aren't pure breadth-first search but are informed to some extent), but breadth-first search is often used in path-finding algorithms and when the entire graph needs to be systematically searched (for the likes of *graph global* algorithms we discussed in Chapter 5).

Path-Finding with Dijkstra's Algorithm

Breadth-first search underpins numerous classical graph algorithms, including Dijkstra's algorithm. Dijkstra (as it is often abbreviated) is used to find the shortest path

between two nodes in a graph. Dijkstra's algorithm is mature (having first been published in 1956) and widely studied and optimized by computer scientists and behaves as follows:

1. Pick the start and end nodes, and add the start node to the set of *solved* nodes (nodes with known shortest path from the start node) with value 0, since the start node is by definition 0 path length away from itself.
2. From the starting node, traverse (breadth-first) to the nearest neighbors and record the path length against each neigbor node.
3. Take the shortest path found to one of the neighbors (picking arbitrarily in the case of ties) and mark that node as solved since we now know the shortest path from the start node to this neighbor.
4. From the set of solved nodes, visit the nearest neighbors (notice the breath-first progression) and record the path lengths from the start node against those neigbors — but don't both visiting any neighboring solved nodes since we know the shortest paths to them already.
5. Repeat steps 3 and 4 until the destination node has been marked solved.

Efficiency of Dijkstra's algorithm

Dijkstra's algorithm is quite efficient since it computes only the lengths of a (relatively small) subset of the possible paths through the graph. When we've solved a node, the shortest path from the start node is then known meaning all subsequent paths can safely build on that knowledge.

In fact we know Dijkstra's algorithm has performance of O(|R| + |N| log |N|) [1] fastest known worst-case implementation though the original was O(|R|^2) [2].

Since Dijkstra is often used to find real-world shortest paths (e.g. for navigation), it's fun to see how it plays out in a real country, certainly more so than the rather dry algorithmic description. In Figure 8-2 we see a logical map of the (rather large) country of Australia, and our challenge is to discover the shortest driving route between Sydney on the East coast (marked *SYD*) and Perth — a continent away — on the West coast of the country marked *PER*. The other major towns and cities are marked with their respective airport codes and we'll discover many of them along the way.

1. That is, the algorithm runs in time proportional to the number of relationships in the graph plus the size of the number of nodes multiplied by the log of the size of the node set
2. The algorithm runs in time proportional to the square of the size of the number of relationships in the graph

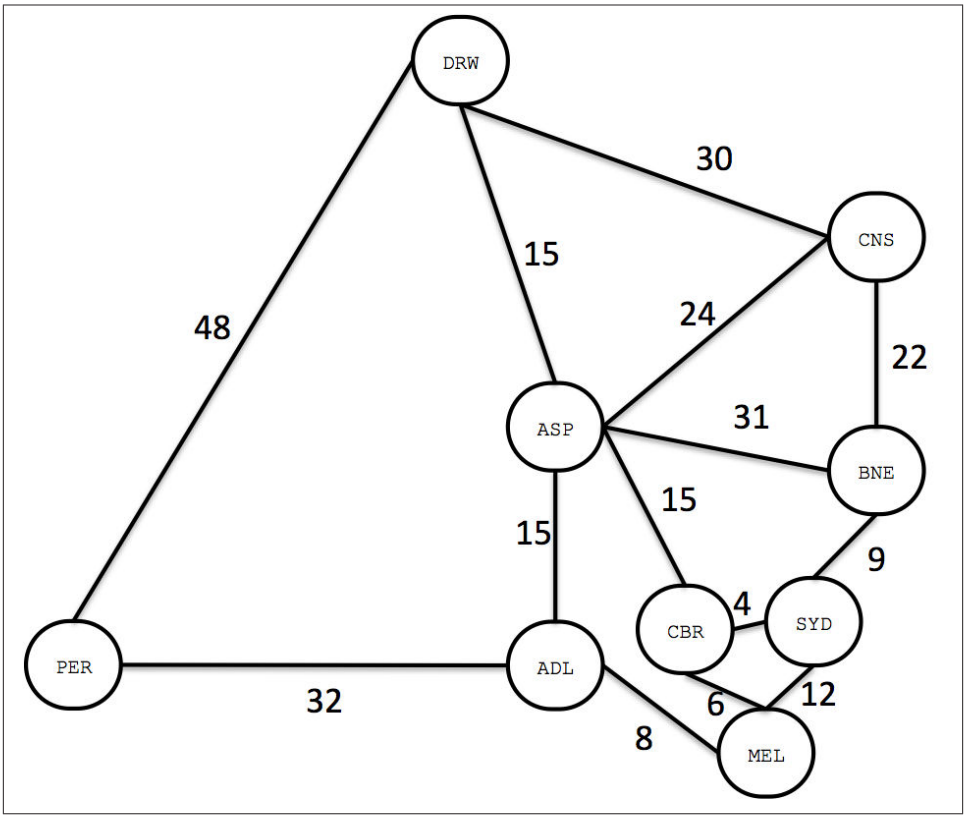


Figure 8-2. A logical representation of Australia and its arterial road network.

Starting at the node representing Sydney in Figure 8-3, we know the shortest path to Sydney is 0 hours since we're already there. Recall that in terms of Dijkstra's algorithm Sydney is now said to be *solved* insofar as we know the shortest path from Sydney to Sydney (which is probably not all that helpful for drivers!). Accordingly we've greyed out the node representing Sydney, added the path length (0), and thickened the node's border — a convention that we'll maintain throughout the remainder of this example.

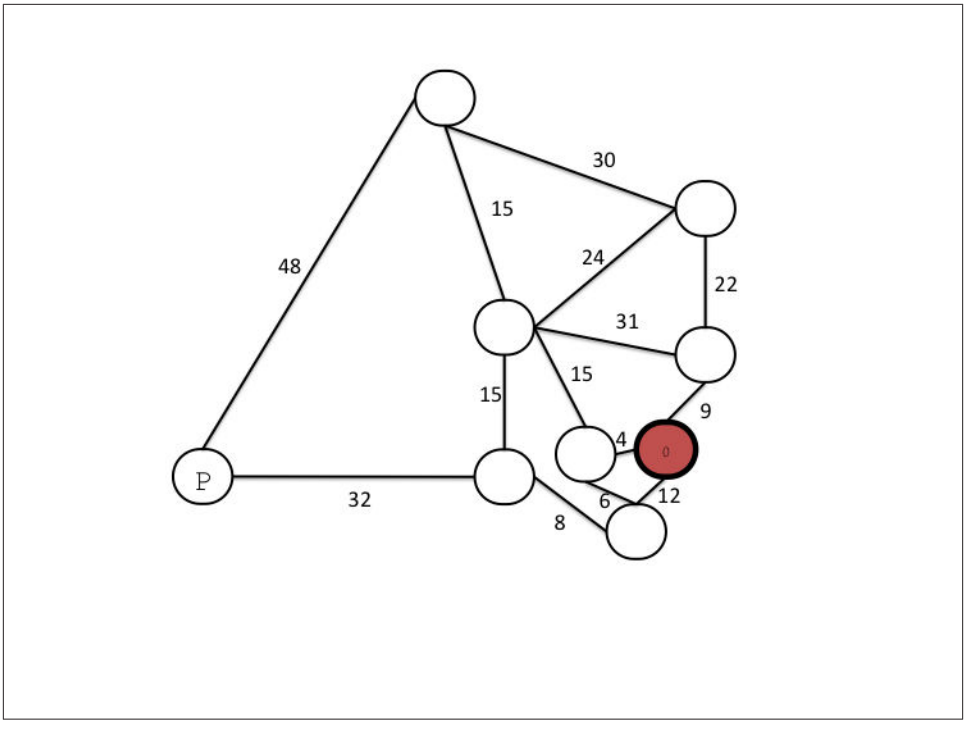


Figure 8-3. The shortest path from Sydney to Sydney is unsurprisingly 0 hours.

Moving one level out from Sydney, our candidate cities are Brisbane to the North by 9 hours, Canberra (surprisingly Australia's capital city) 4 hours to the West and Melbourne 12 hours to the South.

The shortest path we can find is Sydney to Canberra at 4 hours, and so we consider Canberra to be solved as we see in Figure 8-4.

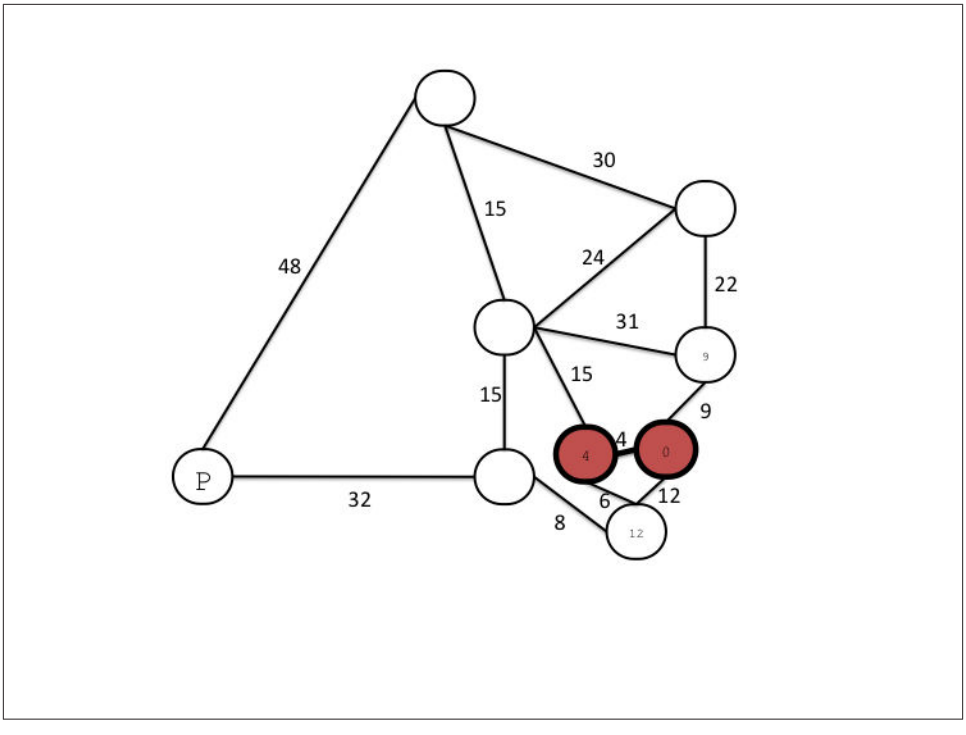


Figure 8-4. Canberra is the closest city to Sydney

The next nodes out are Melbourne at 10 hours from Sydney via Canberra or 12 hours from Sydney directly as we've already seen. We also have Alice Springs at 15 hours from Canberra (19 hours from Sydney) or Brisbane at 9 hours direct from Sydney.

Accordingly we explore the shortest path which is 9 hours from Sydney to Brisbane and consider Brisbane solved at 9 hours in Figure 8-5.

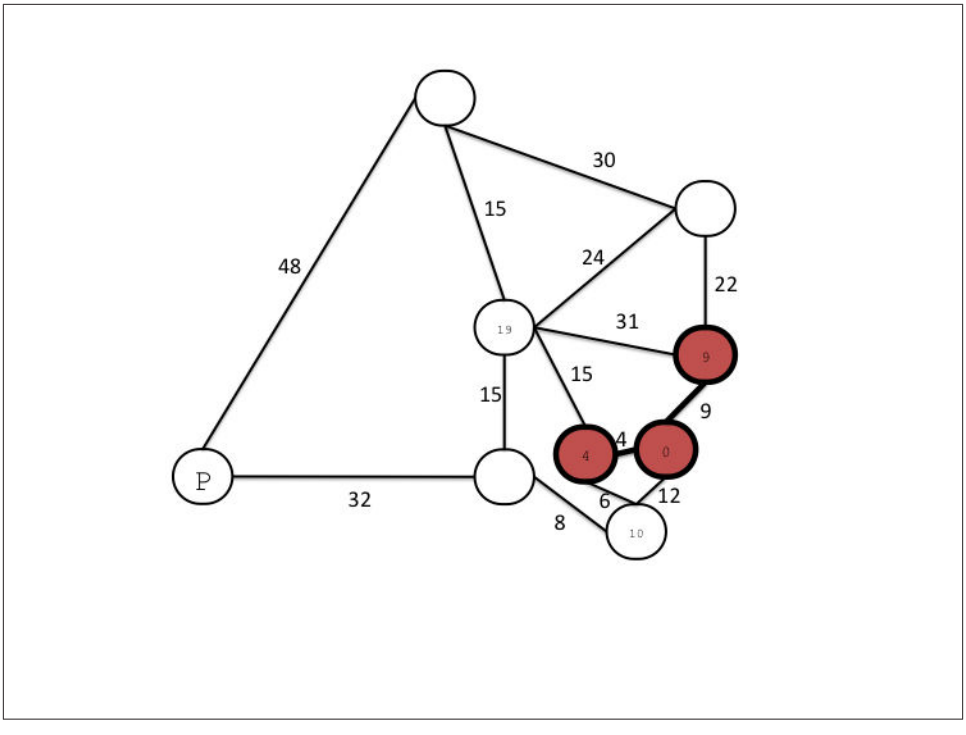


Figure 8-5. Brisbane is the next closest city

The next neighboring nodes from our solved ones are Melbourne (at 10 hours via Canberra or 12 hours direct from Sydney along a different road), Cairns at 31 hours from Sydney via Brisbane, Alice Springs at 40 hours via Brisbane or 19 hours via Canberra — some eye-reddening drive times before we've even left the East coast!

Accordingly we choose the shortest path which is Melbourne at 10 hours from Sydney via Canberra which is shorter than the existing 12 hours direct link, and now consider Melbourne solved as shown in Figure 8-6.

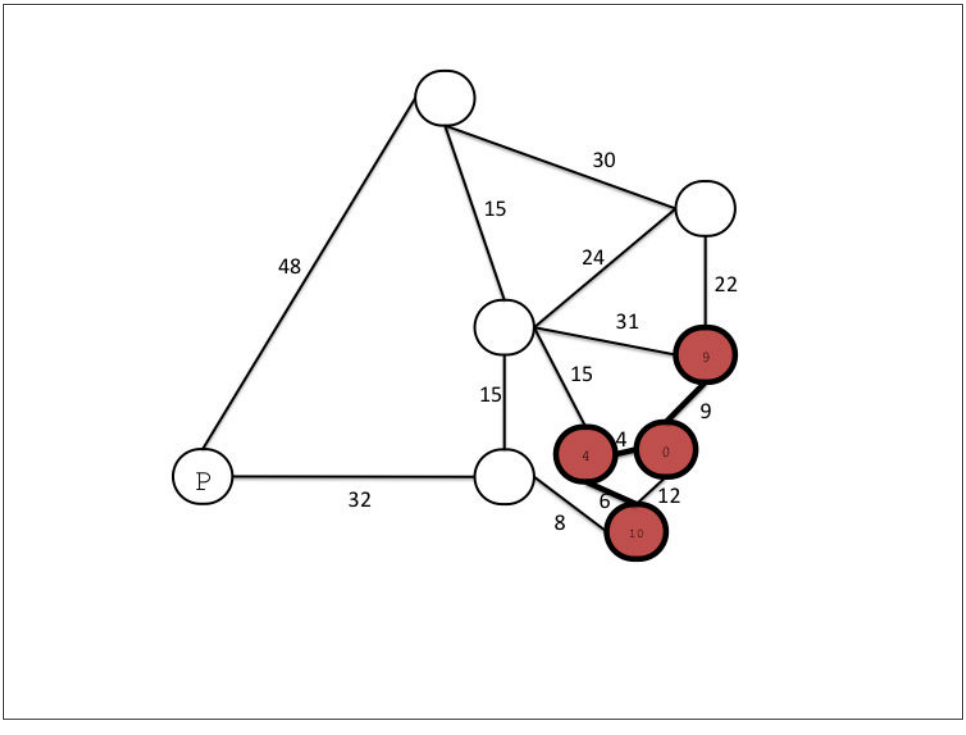


Figure 8-6. Reaching Melbourne, the third-closest city to the start node at representing Sydney

In Figure 8-7 the next layer of neighboring nodes from our solved ones are Adelaide at 18 hours from Sydney (via Canberra and Melbourne), Cairns at 31 hours from Sydney (via Brisbane), and Alice Springs at 19 or 40 hours from Sydney via Canberra or Brisbane respectively. Accordingly we choose Adelaide and consider it solved at a cost of 18 hours.

We don't consider the path Melbourne → Sydney since its destination is a solved node (in fact in this case the start node, Sydney).

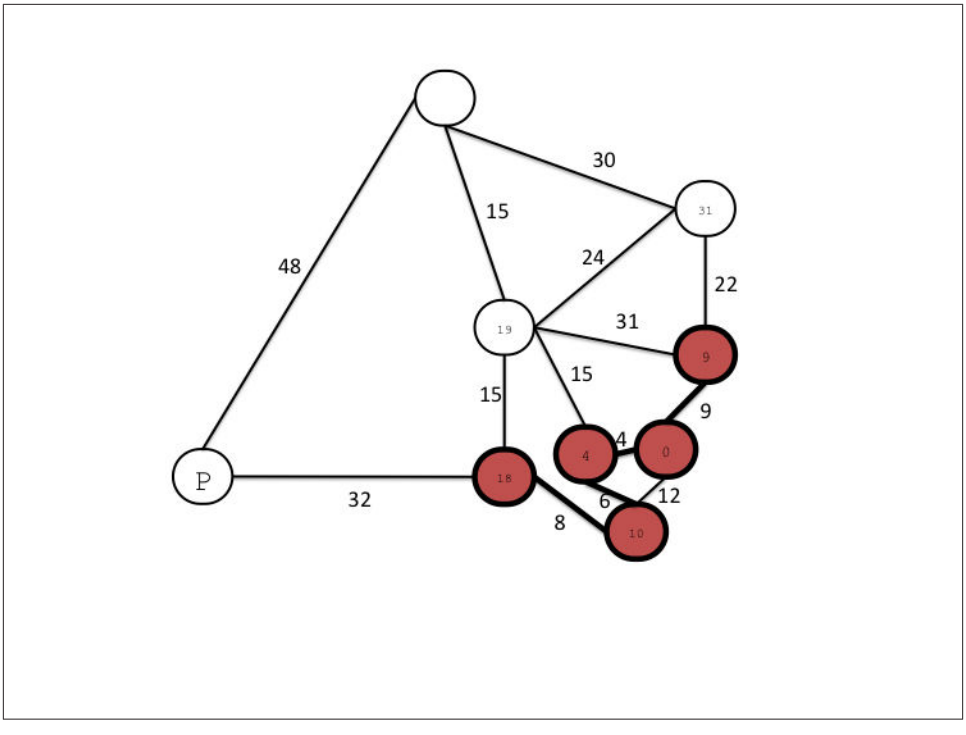


Figure 8-7. Solving Adelaide

The next layer of neighboring nodes from our solved ones are Perth — our final destination — at 50 hours from Sydney (via Adelaide), Alice Springs at 19 hours from Sydney (via Canberra) or 33 hours via Adelaide, and Cairns at 31 hours from Sydney (via Brisbane).

We choose Alice Springs in the case since it has the current shortest path, even though with a god's eye view we know that actually it'll be shorter in the end to go from Adelaide to Perth — just ask any passing bushman. Our cost is 19 hours as shown in Figure 8-8.

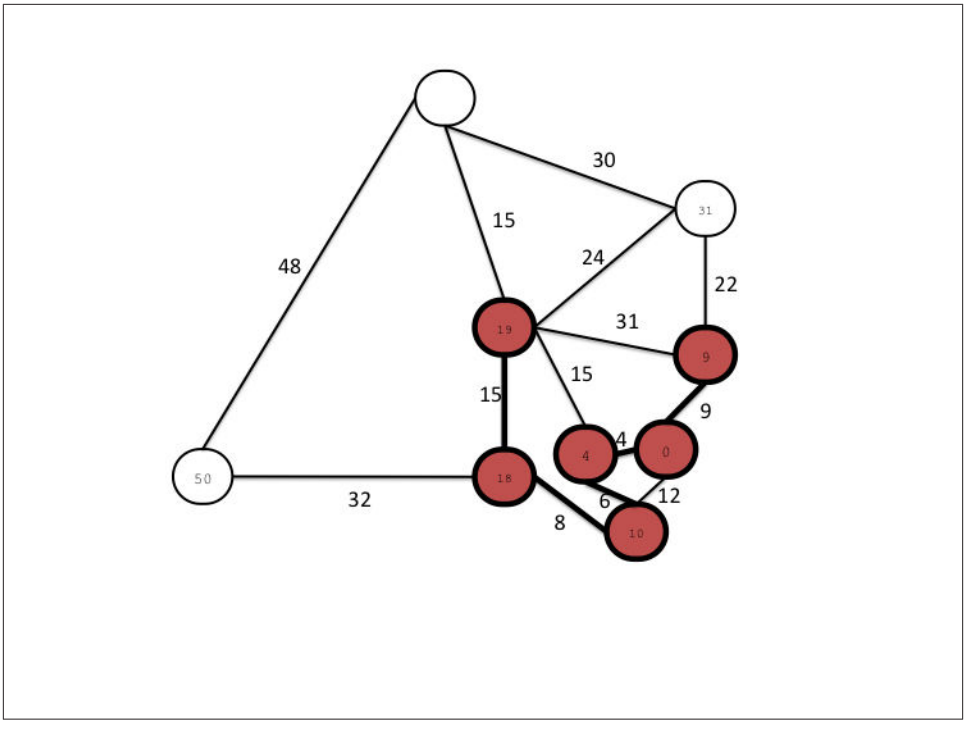


Figure 8-8. Taking a "detour" through Alice Springs

In Figure 8-9, the next layer of neighboring nodes from our solved ones are Cairns at 31 hours via Brisbane or 43 hours via Alice Springs, or Darwin at 34 hours via Alice Springs, or Perth via Adelaide at 50 hours. So we'll take the route to Cairns via Brisbane and consider Cairns solved with a shortest driving time from Sydney at 31 hours.

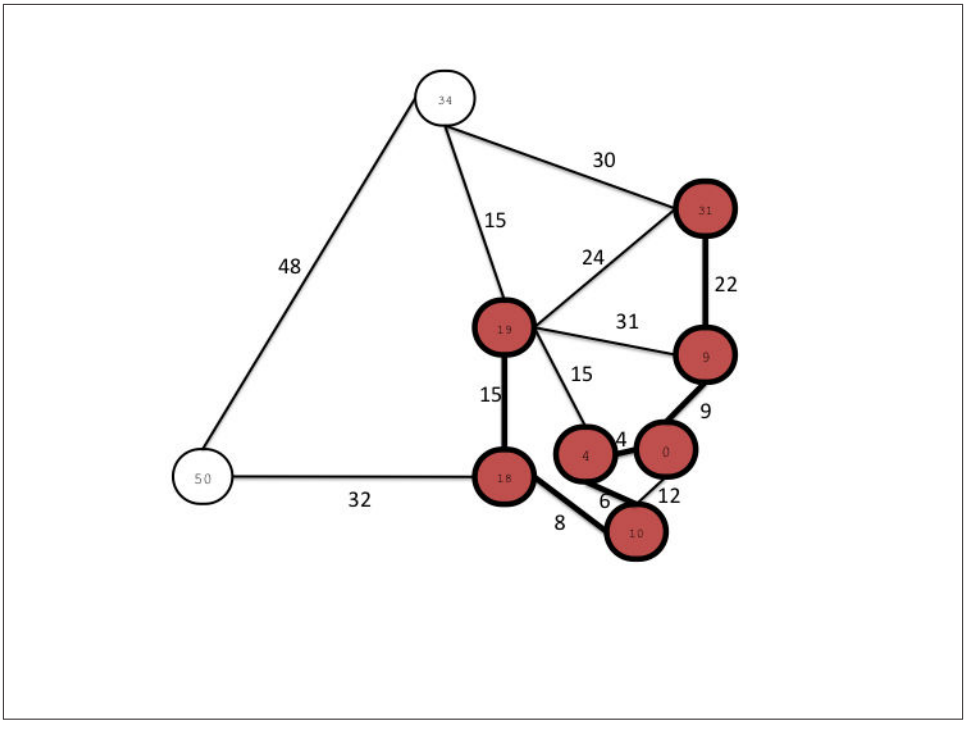


Figure 8-9. Back to Cairns on the East coast

The next layer of neighboring nodes from our solved ones are Darwin at 34 hours from Alice Springs, 61 hours via Cairns, or Perth via Adelaide at 50 hours. Accordingly, we choose the path to Darwin from Alice Springs at a cost of 34 hours and consider Darwin solved as we can see in Figure 8-10.

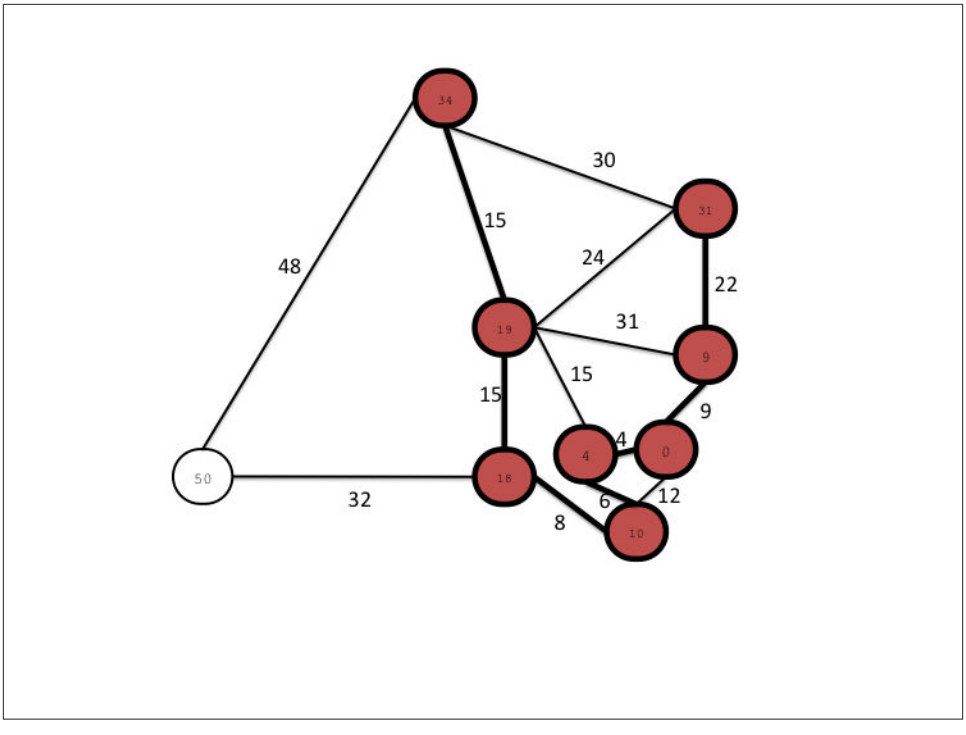


Figure 8-10. To Darwin in Australia's "top-end"

Finally the only neighboring node left is Perth itself as we can see in Figure 8-11. It is accessible via Adelaide at a cost of 50 hours or via Darwin at a cost of 82 hours. Accordingly we choose the route via Adelaide and consider Perth from Sydney solved at a shortest path of 50 hours.

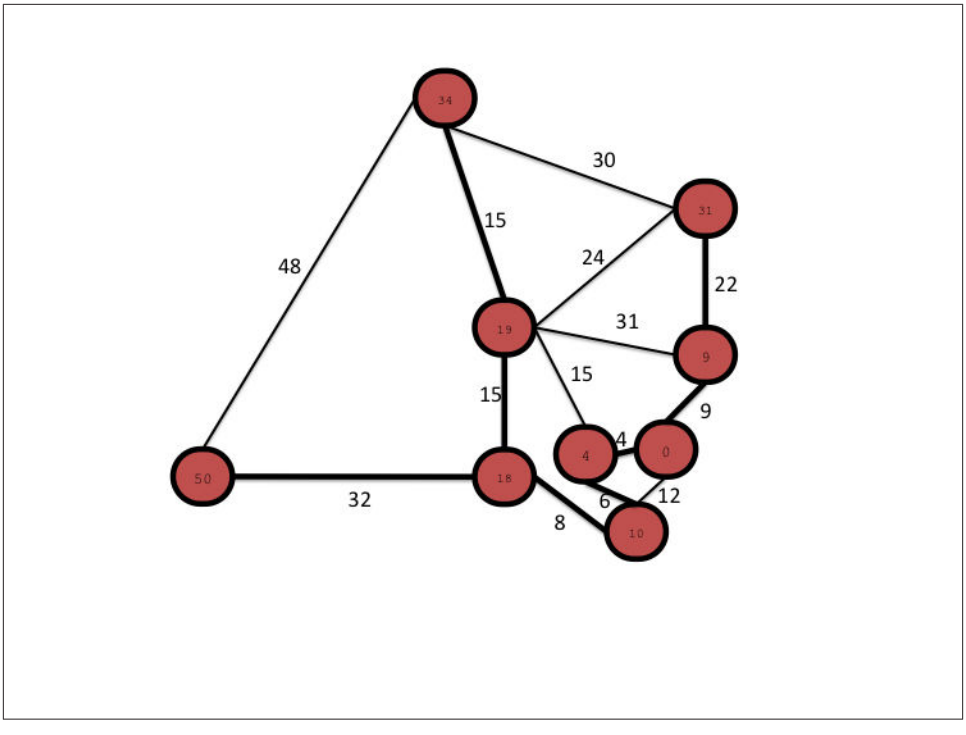


Figure 8-11. Finally reaching Perth, a mere 50 driving hours from Sydney

Dijkstra's algorithm is a good approach but since it's exploration is undirected, there are some pathological graph topologies that can cause worst-case performance problems (where we explore more of the graph that is intuitively necessary, up to the entire graph). Since each possible node is considered one at a time in relative isolation, the algorithm necessarily follows paths that intuitively will never contribute to the final shortest path.

Despite successfully computing the shortest path between Sydney and Perth using Dijkstra's algorithm, anyone with any intuition about map reading would likely not have chosen, for instance, to explore the route northwards from Adelaide since it *feels* longer. If we had some heuristic mechanism to guide the algorithm like in a best-first search (e.g. prefer to head west over east, prefer south over north) we could have perhaps avoided the side-trips to Brisbane, Cairns, Alice Springs, and Darwin in this example. However best-first searches are greedy and try to move towards the destination node even if there is an obstacle (e.g. dirt track), but we can do better.

The A* Algorithm

The A* (pronounced "A-star") algorithm is an improvement on the classic Dijkstra algorithm based on the observation that some searches are *informed* and that by being

informed we can make better choices about paths to take through the graph (for example that we don't reach Perth from Sydney traversing the height of an entire continent to Darwin first). A* is both like Dijkstra in that it can potentially search a large swathes of a graph, but it's also like a greedy best-first search insofar as it uses a heuristic to guide it. A* thus combines aspects of Dijkstra's algorithm (which prefers nodes close to the current starting point) and best-first search (which prefers nodes closer to the destination) to provide a provably optimal solution for finding shortest paths in a graph.

In A* we split the path cost into two parts, one *g(n)* that is the cost of the path from the starting point to some node *n* and *h(n)* which represents the estimated cost of the path from the node *n* to the destination node, computed by heuristic (an intelligent guess). The A* algorithm balances *g(n)* and *h(n)* as it iterates the graph ensuring that at each iteration it chooses the node with the lowest overall cost $f(n) = g(n) + h(n)$.

But breadth-first algorithms aren't only good for path finding. In fact using breadth-first search as our method for iterating over all elements of a graph, we can now consider a number of interesting higher-order algorithms from graph theory that yield significant (predictive) insight into the behavior of connected data.

Graph Theory and Predictive Modeling

Graph theory is a mature and well-understood field of study concerning the nature of networks (or from our point of view, connected data). The analytic techniques that have been developed by graph theoreticians can be brought to bear on a range of interesting problems, and since the low-level traversal mechanisms (like breadth-first search) are already known to us, we can start to consider higher order analyses.

What's particularly appealing about applying graph theory techniques to data analysis is their broad applicability. They're especially valuable where we're presented with a domain into which we have little insight, or little comprehension of what insight we're even able to extract. In such cases there are a range of techniques from graph theory and social sciences which we can straightforwardly apply to gain insight.

In the next few sections we'll introduce some of the key concepts in social graph theory. And we'll introduce those concepts in a humane and (mostly) non-mathematical way

in a typical social domain based on the works of sociologist Mark Granovetter [3] and Easley/Kleinberg [4].

Property graphs and graph theory

Much of the work on graph theory assumes a slightly different model to the property graphs we've been using throughout this book. Typical problem spaces ignore direction and labeling of graphs, instead assuming a single label derived from the domain (e.g. `friend` or `colleague`) and ignoring direction.

We're going to take a hybrid approach here. Supporting relationship names is helpful for naming various kinds of relationships from graph theory with more domain-specific meaning. However where it suits, we'll ignore (or not) relationship direction.

Importantly, the same principles are upheld by the graph structure whether we're using a property graph or a semantically poorer graph model.

Triadic Closures

A triadic closure is a property of (social) graphs whereby if two nodes are connected via a path involving a third node, there is an increased likelihood that the two nodes will become directly connected in future. This is a familiar enough situation for us in a social setting whereby if we happen to be friends with two people, ultimately there's an increased chance that those people will become direct friends too, since by being our friend in the first place, it's an indication of social similarity and suitability.

From his analysis, Granovetter noted that a subgraph upholds the *strong triadic closure property* if it has a node *A* with strong relationships to two other nodes *B* and *C*, and those two other nodes *B* and *C* then have at least a *weak* and potentially a *strong* relationship between them. This is an incredibly strong assertion and will not be typically upheld by all subgraphs in a graph. Nonetheless it is sufficiently commonplace (particularly in social networks) to be trusted as a predictive aid.

Strong and weak relationships

We purposely won't explicitly define *weak* and *strong* relationships, since they're domain-specific characteristics. But for example a strong social relationship might be

3. In particular his pivotal work on the strength of weak ties in social communities, see: *http://sociology.stanford.edu/people/mgranovetter/documents/granstrengthweakties.pdf*

4. See: *http://www.cs.cornell.edu/home/kleinber/networks-book/*

friends who've exchanged phone calls in the last week, whereas a weak social relationship might be friends who've merely observed one-another's Facebook status.

Let's see how the strong triadic closure property works as a predictive aid in a workplace graph. Let's start with a simple organizational hierarchy where Alice managed Bob and Charlie, but where there are not (yet) any connections between her subordinates as in Figure 8-12.

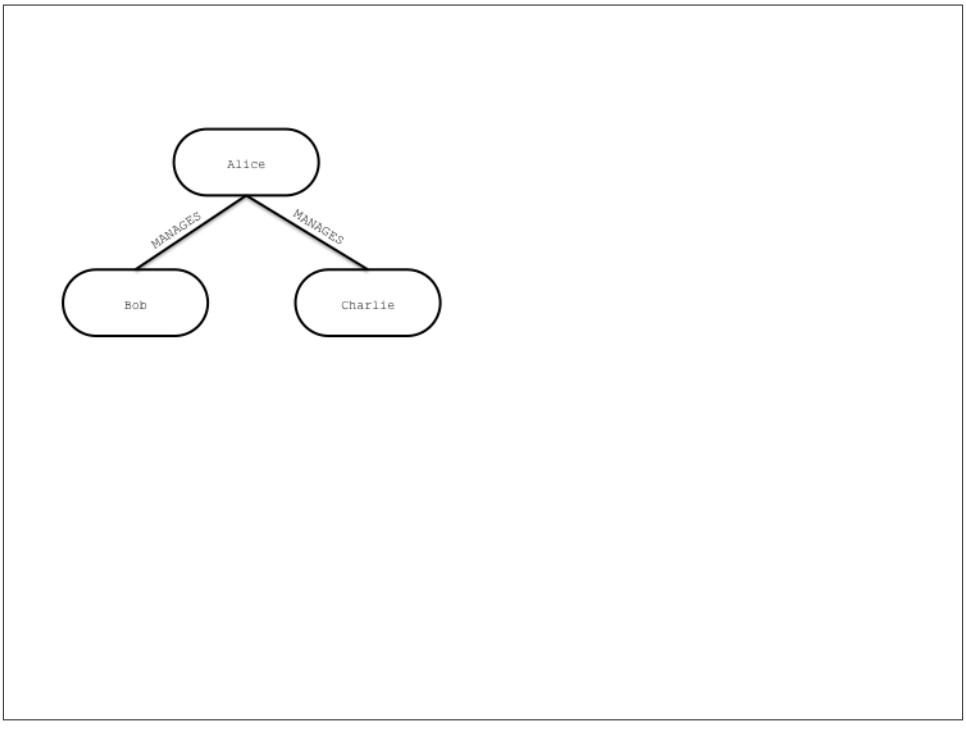


Figure 8-12. Alice manages Bob and Charlie

This is a rather strange situation in the workplace, since it's unlikely that Bob and Charlie will be disconnected strangers. Whether they're high level executives and will therefore be peers under Alice's executive management or whether they're assembly line workers and therefore will be close colleagues again under Alice who acts as foreman, even informally we expect Bob and Charlie to be somehow connected.

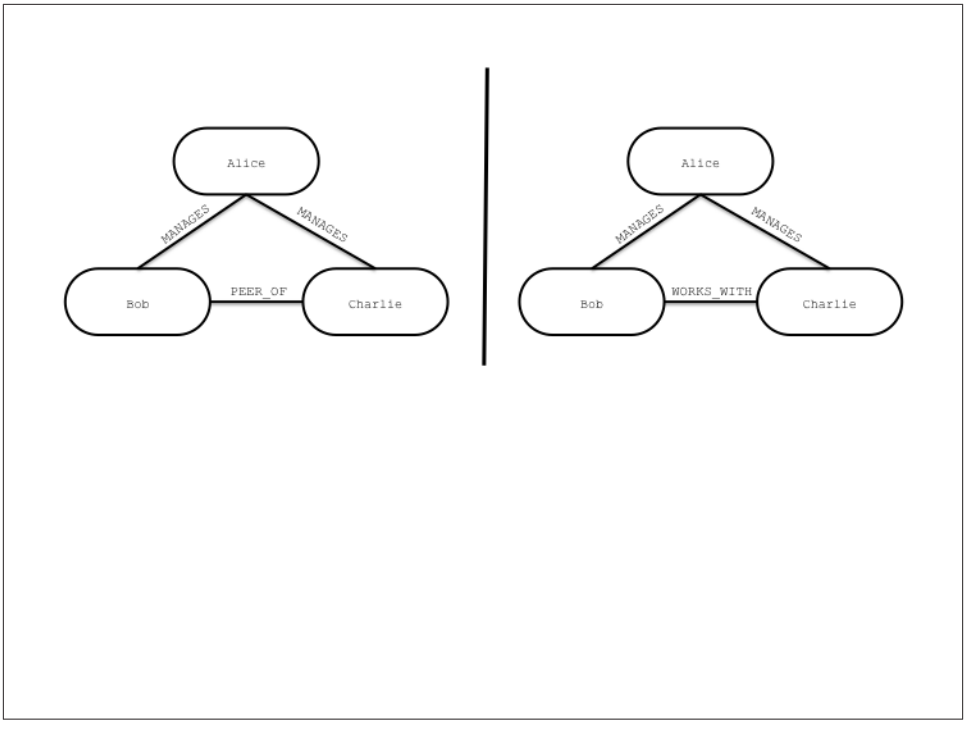


Figure 8-13. Bob and Charlie work together under Alice

However since Bob and Charlie both work with Alice, there's a strong possibility they're going to end up working together as we see in Figure 8-13. This is consistent with the strong triadic closure property, which suggests that either Bob is a peer of Charlie (we'll call this a *weak* relationship) or Bob works with Charlie (which we'll term a *strong* relationship). Adding the third `WORKS_WITH` or `PEER_OF` relationship between Bob and Charlie closes the triangle and hence the term *triadic closure*.

The empirical evidence from many domains including sociology, public health, psychology, anthropology and even technology (e.g. Facebook, Twitter, LinkedIn) suggests that the tendency towards triadic closure is real and substantial, consistent with anecdotal evidence and sentiment. But simple geometry isn't all that's at work here, the sentiment of the relationships involved in a graph also have a significant bearing on the formation of *stable* triadic closures.

Structural Balance

If we recall Figure 8-12 it's intuitive to see how Bob and Charlie can become co-workers (or peers) under Alice's management. For example purposes, we're going to make an assumption that the `MANAGES` relationship is somewhat negative (after all people don't

like getting bossed around) whereas the PEER_OF and WORKS_WITH relationship are positive (since people like their peers and the folks they work with).

We know from our previous discussion on the strong triadic closure principle that in Figure 8-12 where Alice MANAGES Bob and Charlie, a triadic closure should be formed. That is in the absence of any other constraints, we would expect at least a PEER_OF, a WORKS_WITH or even a MANAGES relationship between Bob and Charlie.

A similar tendency towards creating a triadic closure exists if Alice MANAGES Bob who in turn WORKS_WITH Charlie as we can see in Figure 8-14. Anecdotally this rings true: if Bob and Charlie work together it makes sense for them to share a manager, especially if the organization seemingly allows Charlie to function without managerial supervision.

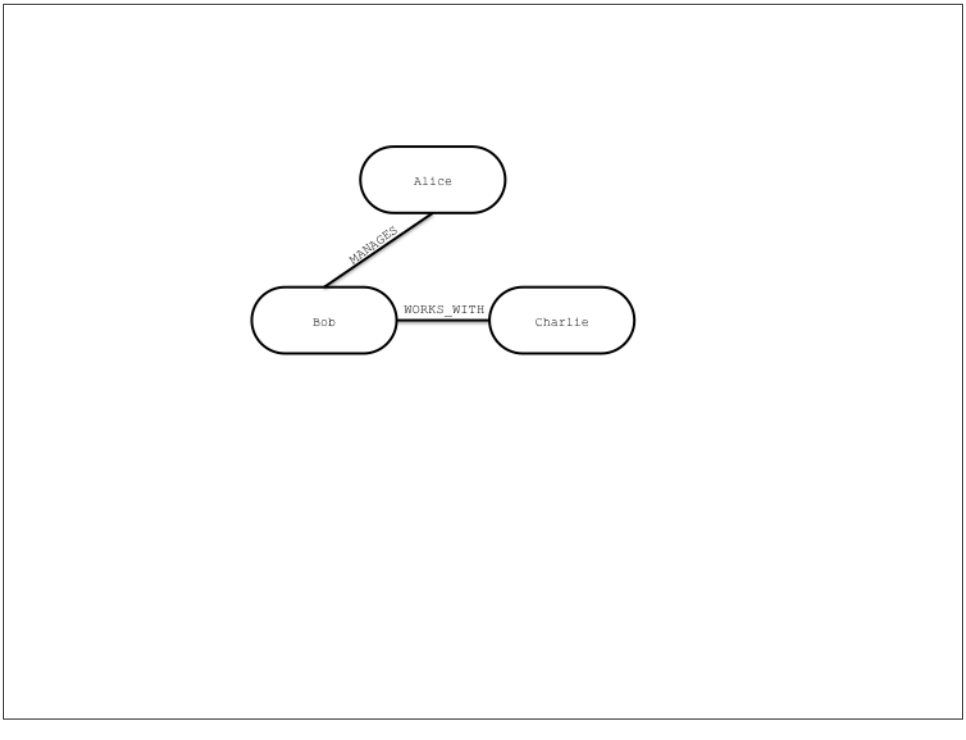


Figure 8-14. Alice manages Bob who works with Charlie

However, applying the strong triadic closure principle blindly can lead to some rather odd and uncomfortable-looking organization hierarchies. For instance, If Alice MANAGES Bob and Charlie but Bob also MANAGES Charlie, we have a recipe for discontent. Nobody would wish it upon Charlie that he's managed both by his boss and his boss' boss as in Figure 8-15!

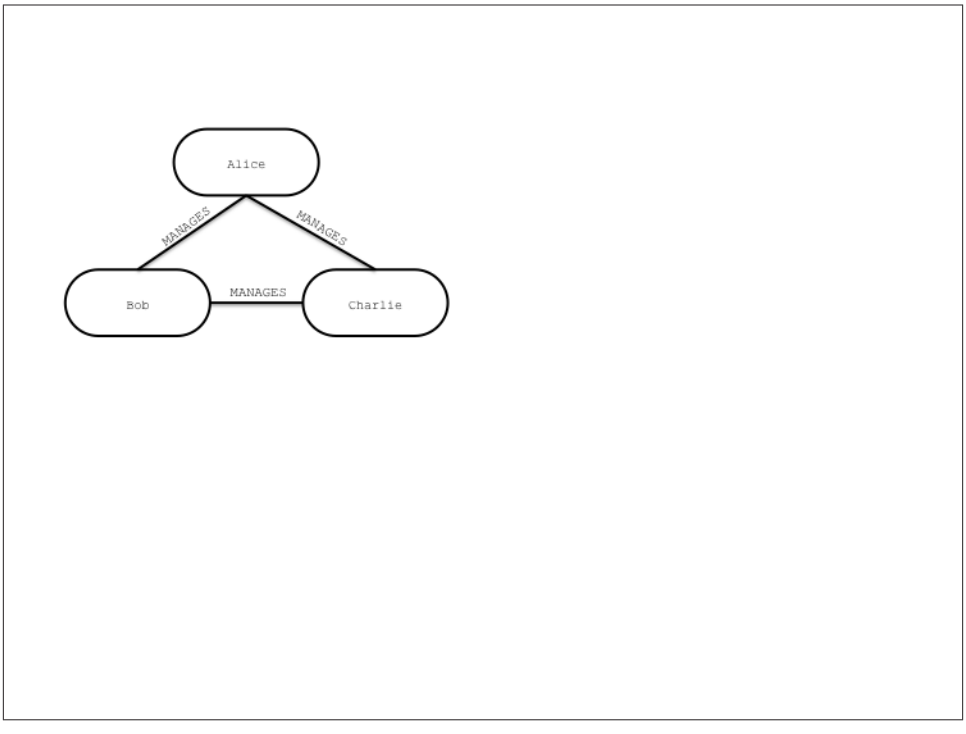


Figure 8-15. Alice manages Bob and Charlie, while Bob also manages Charlie

Similarly it's uncomfortable for Bob if he is managed by Alice while working with Charlie who is also Alice's workmate. This cuts awkwardly across organization layers as we see in Figure 8-16. It also means Bob could never safely let off steam about Alice's management style!

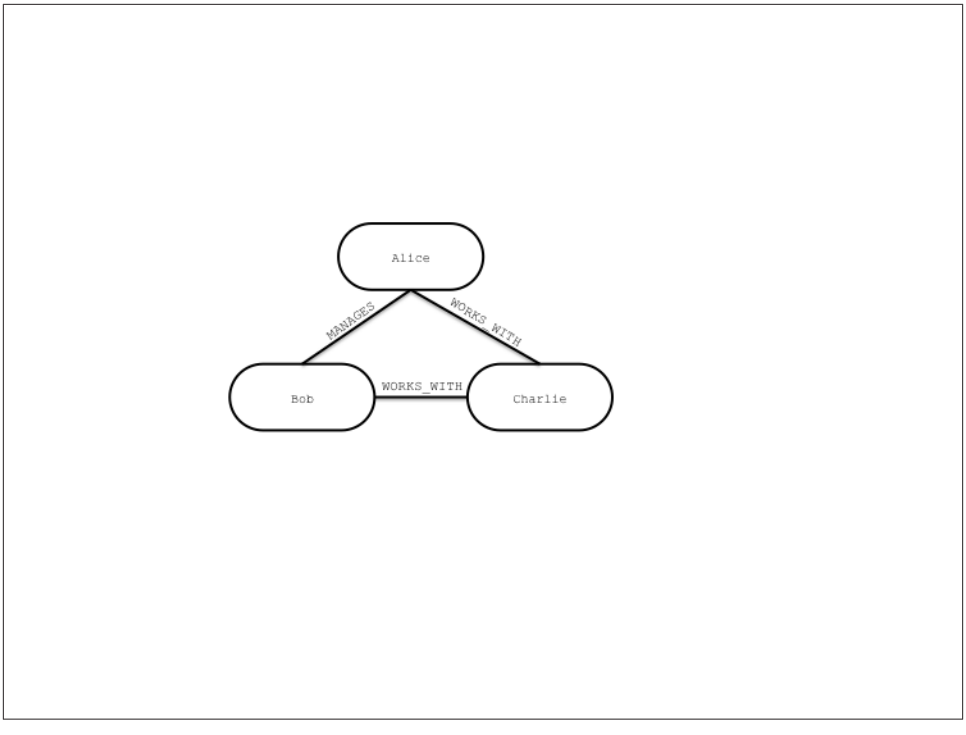


Figure 8-16. Alice manages Bob who works with Charlie, while also working with Charlie

The awkward hierarchy in Figure 8-16 whereby Charlie is both a peer of the boss and a peer of another worker is unlikely to be socially pleasant so Charlie and Alice will agitate against it (either wanting to be a boss or a worker). It's similar for Bob who doesn't know for sure whether to treat Charlie in the same way he treat his manager Alice (since Charlie and Alice peers) or as his own direct peer.

It's clear that the triadic closures in Figure 8-15 and Figure 8-16 are palpably uncomfortable to us, eschewing our innate preference for structural symmetry and rational layering. This preference is given a name in graph theory: *structural balance*.

Anecdotally, there's a much more acceptable — structurally balanced — triadic closure if Alice `MANAGES` Bob and Charlie, but where Bob and Charlie are themselves workmates connected by a `WORKS_WITH` relationships as we can see in Figure 8-17.

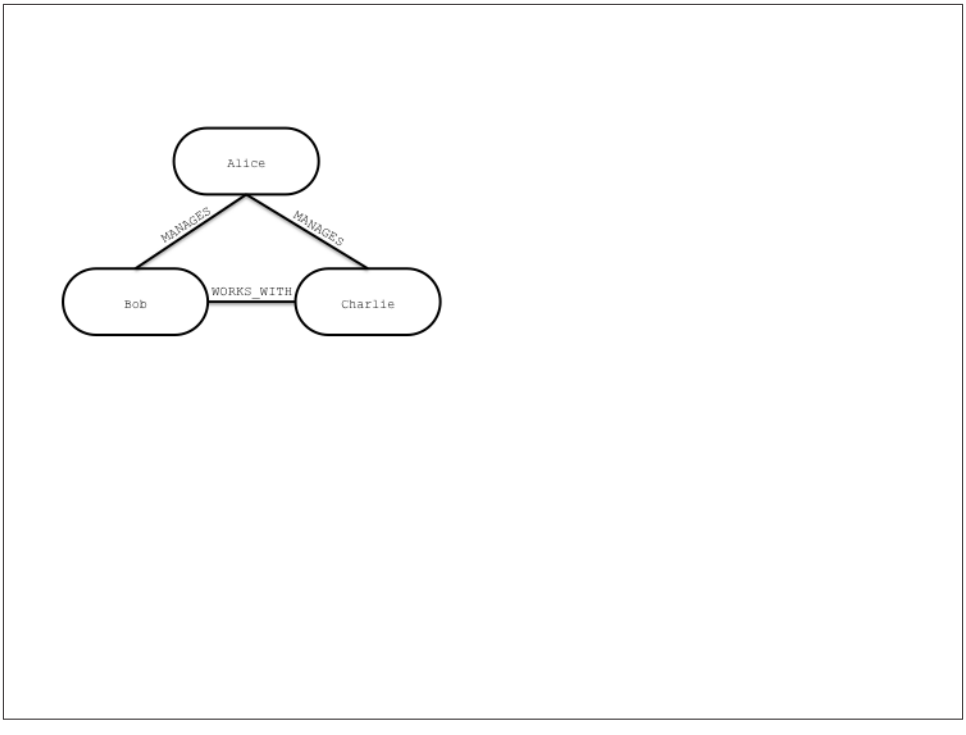


Figure 8-17. Workmates Bob and Charlie are managed by Alice

The same structural balance manifests itself in an equally acceptable triadic closure where Alice, Bob, and Charlie are all workmates. In this arrangement the workers are in it together which can be a socially amicable arrangement that engenders camaraderie as in Figure 8-18.

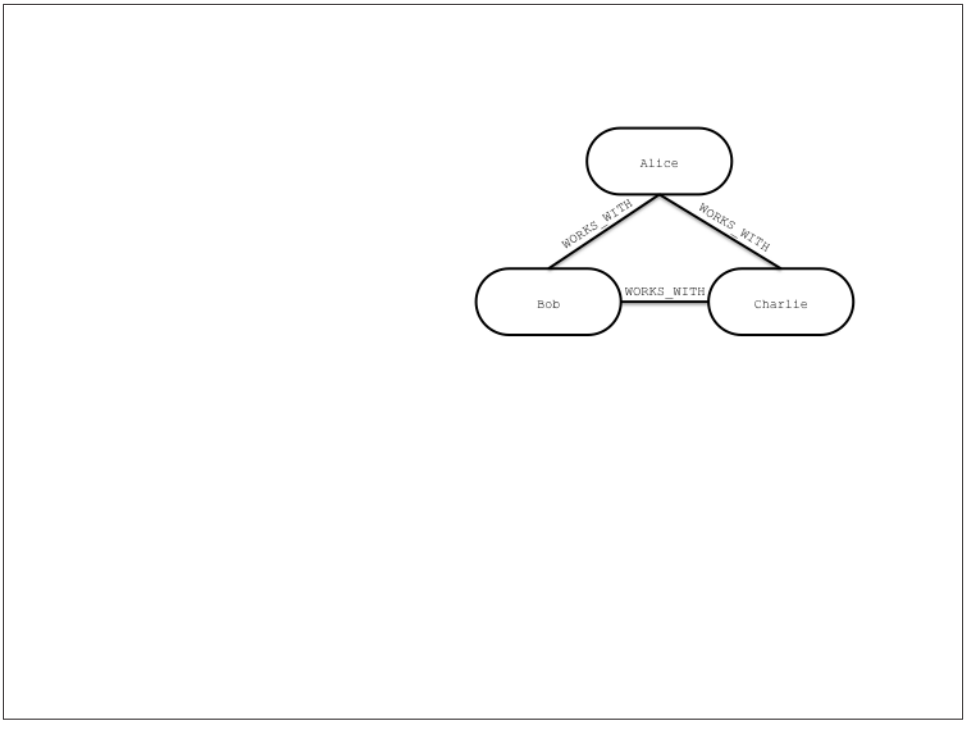


Figure 8-18. Alice, Bob and Charlie are all workmates

In Figure 8-17 and Figure 8-18 the triadic closures are idiomatic and constructed with either three `WORKS_WITH` relationships or two `MANAGES` and a single `WORKS_WITH` relationship. They are all *balanced* triadic closures. To understand what it means to have balanced and unbalanced triadic closures, we'll add more semantic richness to the model by declaring that the `WORKS_WITH` relational is socially positive (since co-workers spend a lot of time interacting) while `MANAGES` is negative relationship since managers spend overall less of their time interacting with individuals in their charge.

Given this new dimension of positive and negative sentiment, we can now ask the question "What is so special about these balanced structures?" It's clear that strong triadic closure is still at work, but that's not the only driver. In this case the notion of *structural balance* also has an effect. A structurally balanced triadic closure consists of relationships of all strong sentiments (our `WORKS_WITH` or `PEER_OF` relationships) or two relationships have negative sentiments (`MANAGES` in our case) with a single positive relationship.

We see this often in the real world. If we have two good friends, then social pressure tends towards those good friends themselves becoming good friends. It's unusual that those two friends themselves are enemies because that puts a strain on our friendships.

One friend cannot express their dislike of the other to us, since the other person is our friend too! Given those pressures, it's reasonably likely that ultimately the group will resolve its differences and good friends will emerge.

This would change our unbalanced triadic closure (two relationships with positive sentiments and one negative) to a balanced closure since all relationships would be of a positive sentiment much like our collaborative scheme where Alice, Bob, and Charlie all work together in Figure 8-18.

However another plausible (though arguably less pleasant) outcome would be where we take sides in the dispute between our "friends" creating two relationships with negative sentiments — effectively ganging up on an individual. Now we can engage in gossip about our mutual dislike of a former friend and the closure again becomes balanced. Equally we see this reflected in the organizational scenario where Alice, by managing Bob and Charlie, becomes in effect their workplace enemy as in Figure 8-17.

Mining organizational data in the real world

We don't have to derive these graphs from analyzing organizational charts, since that's a static and often inconsistent view of how an organization really works [5]. A practical and timely way of generating the graph would instead be to run these kinds of analyses over the history of email exchanges between individuals within a company.

We'd easily be able to identify a graph at large scale and from that we'd be able to make predictive analyses about the evolution of organizational structure by looking for opportunities to create balanced closures. Such structures might be a boon – that we observe employees are already self-organizing for a successful outcome – or they might be indicative of some malpractice – that some employees moving into shadowy corners to commit corporate fraud! Either way the predictive power of graphs enable us to engage those issues proactively.

Balanced closures add another dimension to the predictive power of graphs. Simply by looking for opportunities to create balanced closures across a graph, even at very large scale, we can modify the graph structure for accurate predictive analyses. But we can go further, and in the next section we'll bring in the notion of *local bridges* which give us valuable insight into the communications flow of our organization and from that knowledge comes the ability to adapt it to meet future challenges.

5. With due respect to our colleagues in HR, the map is not the territory.

Local Bridges

An organization of only three people as we've been using is anomalous, and the graphs we've studied in this section are best thought of as small sub-graphs as part of a larger organizational hierarchy. When we start to consider managing a larger organization we expect a much more complex graph structure, but we can also apply other heuristics to the structure to help make sense of the business. In fact once we have introduced other parts of the organization into the graph, we can reason about global properties of the graph based on the locally-acting strong triadic closure principle.

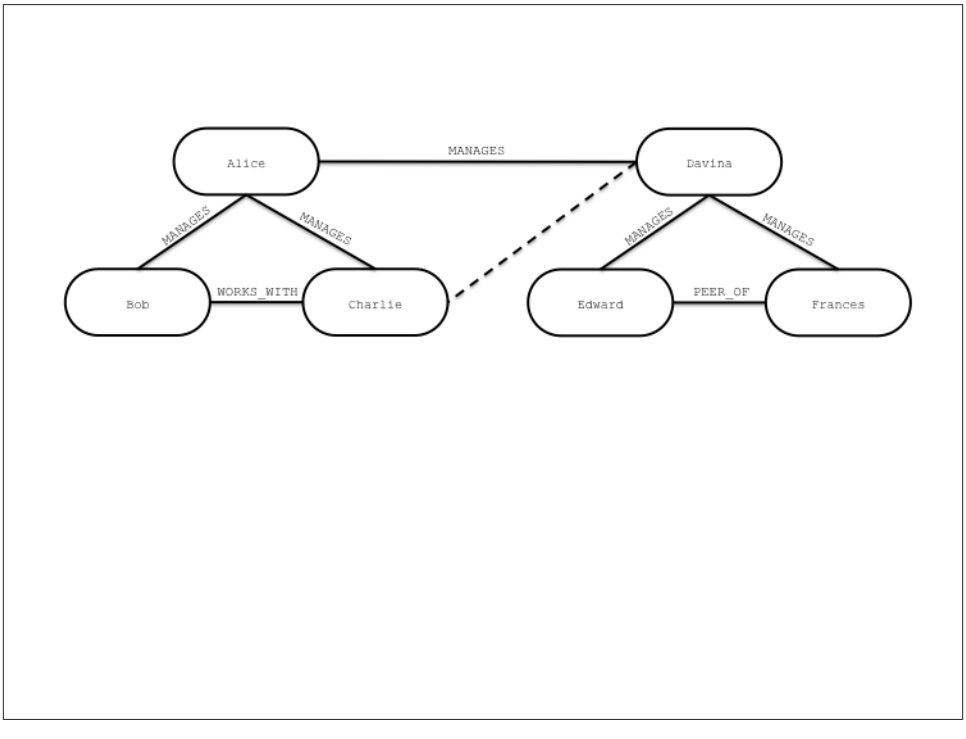


Figure 8-19. Alice has skewed line management responsibility

In Figure 8-19 we're presented with a counter-intuitive scenario where two groups in the organization are managed by Alice and Davina respectively. However, we have the slightly awkward structure that Alice not only runs a team with Bob and Charlie but also manages Davina. While this isn't beyond the realms of possibility (Alice may indeed have such responsibilities) it feels intuitively awkward from an organization design perspective.

From a graph theory perspective it's also unlikely. Since Alice participates in two strong relationships, she `MANAGES` Charlie (and Bob) and `MANAGES` Davina, naturally we'd like

to create a triadic closure by adding at least a PEER_OF relationship between Davina and Charlie (and Bob). But Alice is also involved in a *local bridge* to Davina – together they're a sole communication path between groups in the organization. Having the relationship Alice MANAGES Davina means we'd in fact have to create the closure. These two properties – local bridge and strong triadic closure – are in opposition.

Yet if Alice and Davina are peers (a weak relationship), then the strong triadic closure principle isn't activated since there's only one strong relationship — the MANAGES relationship to Bob (or Charlie) — and the local bridge property is valid as we can see in Figure 8-20.

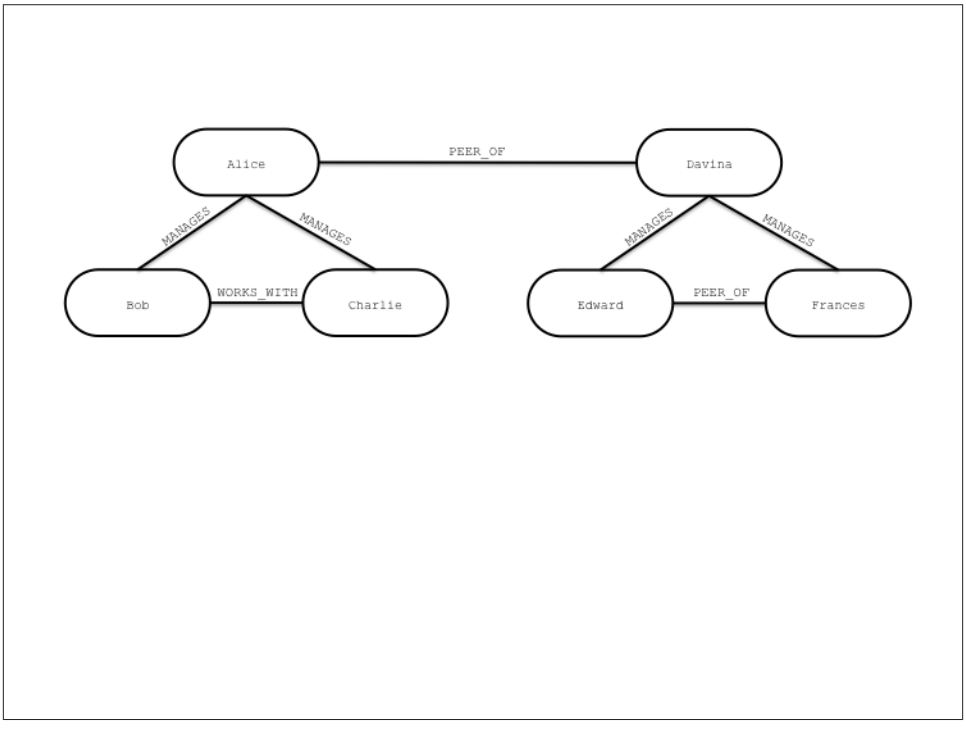


Figure 8-20. Alice and Davina are connected by a local bridge

What's interesting about this local bridge is that it describes a communication channel between groups in our organization and those channels are extremely important to the vitality of our enterprise. In particular to ensure the health of our company we'd make

sure that local bridge relationships are healthy and active, or equally we might keep an eye on local bridges to ensure no impropriety (embezzlement, fraud, etc) occurs.

Finding your next job

This notion of weak social links is particularly pertinent in algorithms like (social) job search. The remarkable thing about job searches is that it's rarely a close friend that provides the best recommendation, but a looser acquaintance.

Why should this be? Our close friends share a similar world view (they're in the same *graph component*) and have similar access to the available jobs data and hold similar opinions about those jobs. A friend across a local bridge is clearly in a different social network (a different component) and have correspondingly different access to jobs and a different viewpoint about them. So if you're going to find a job, look across a local bridge since that's where people who have different knowledge to you and your friends hang out.

This same property of local bridges being weak links (`PEER_OF` in our example organization) is a property that is prevalent throughout social graphs. This means we can start to make predictive analyses of how our organization will evolve based on empirically-derived local bridge and strong triadic closure notions. So given an arbitrary organizational graph we can see how the business structure is likely to evolve and plan for those eventualities.

Summary

Graphs are a remarkable structure and are well studied and often fruitfully applied to modeling and analysis. We now have native graph databases which project the powerful graph model into our applications, bringing with them numerous powerful algorithmic and graph theory techniques which we can use to build better, smarter information systems.

As we've seen throughout this book, graph algorithms and analytical techniques from graph theory are not demanding: we need only understand how to *apply* them to achieve our goals since the fundamental techniques are mature and well-studied. We leave this book with a simple call to arms: to ask the reader to embrace graph data and to apply the modeling, processing, and analytical patterns we've shown in this book to create truly pioneering and impactful information systems.