Build Flexible Applications with Graph Data

Programming the Semantic Web

Toby Segaran,
Colin Evans & Jamie Taylor

O'REILLY®

Programming the Semantic Web

With this book, the promise of the semantic web—in which machines can find, share, and combine data on the Web—is not just a technical possibility, but a practical reality. *Programming the Semantic Web* demonstrates several ways to implement semantic web applications, using current and emerging standards and technologies. You'll learn how to incorporate existing data sources into semantically aware applications and publish rich semantic data.

Each chapter walks you through a single piece of semantic technology and explains how you can use it to solve real problems. Whether you're writing a simple mashup or maintaining a high-performance enterprise solution, *Programming the Semantic Web* provides a standard, flexible approach for integrating and future-proofing systems and data.

THIS BOOK WILL HELP YOU:

- Learn how the semantic web allows new and unexpected uses of data to emerge
- Understand how semantic technologies promote data portability with a simple, abstract model for knowledge representation
- Become familiar with semantic standards, such as the Resource Description Framework (RDF) and the Web Ontology Language (OWL)
- Make use of semantic programming techniques to both enrich and simplify current web applications

INTRODUCTORY INTERMEDIATE ADVANCED

Previous web programming experience recommended

"The technologies are here, the tools are ready, and this book will show you how to make it all work for you."

—**Jim Hendler**
AI researcher and one of the originators of the semantic web

"Programming the Semantic Web *is hands-down the best practical introduction to the semantic web, a must-read for all engineers new to the Web of data. This book will give you the tools to both publish data on the Web and to exploit metadata in your own applications.*"

—**Peter Mika**
Researcher and data architect, Yahoo!

Toby Segaran, author of *Programming Collective Intelligence* (O'Reilly), was the founder of Incellico, a biotech software company.

Colin Evans combines machine learning and semantic analysis at Metaweb Technologies.

Jamie Taylor has a Ph.D. in economics. He works on Freebase and teaches computers about semantics.

US $39.99 CAN $49.99
ISBN: 978-0-596-15381-6

Safari Books Online
Free online edition
for 45 days with purchase of this book. Details on last page.

O'REILLY®
oreilly.com

Download at Boykma.Com

PART I

Semantic Data

Download at Boykma.Com

CHAPTER 1

Why Semantics?

Natural language is amazing. Without any effort you can ask a stranger how to find the nearest coffee shop; you can share your knowledge of music and martini making with your community of friends; you can go to the library, pick up a book, and learn from an author who lived hundreds of years ago. It is hard to imagine a better API for knowledge.

As a simple example, think about the following two sentences. Both are of the form "subject-verb-object," one of the simplest possible grammatical structures:

1. Colin enjoys mushrooms.
2. Mushrooms scare Jamie.

Each of these sentences represents a piece of information. The words "Jamie" and "Colin" refer to specific people, the word "mushroom" refers to a class of organisms, and the words "enjoys" and "scare" tell you the relationship between the person and the organism. Because you know from previous experience what the verbs "enjoy" and "scare" mean, and you've probably seen a mushroom before, you're able to understand the two sentences. And now that you've read them, you're equipped with new knowledge of the world. This is an example of semantics: symbols can refer to things or concepts, and sequences of symbols convey meaning. You can now use the meaning that you derived from the two sentences to answer simple questions such as "Who likes mushrooms?"

Semantics is the process of communicating enough meaning to result in an action. A sequence of symbols can be used to communicate meaning, and this communication can then affect behavior. For example, as you read this page, you are integrating the ideas expressed in these words with all that you already know. If the semantics of our writing in this book is clear, it should help you create new software, solve hard problems, and do great things.

But this book isn't about natural language; rather, it's about using semantics to represent, combine, and share knowledge between communities of machines, and how to write systems that act on that knowledge.

Download at Boykma.Com

If you have ever written a program that used even a single variable, then you have programmed with semantics. As a programmer, you knew that this variable represented a value, and you built your program to respond to changes in the variable. Hopefully you also provided some comments in the code that explained what the variable represented and where it was used so other programmers could understand your code more easily. This relationship between the value of the variable, the meaning of the value, and the action of the program is important, but it's also implicit in the design of the system.

With a little work you can make the semantic relationships in your data explicit, and program in a way that allows the behavior of your systems to change based on the meaning of the data. With the semantics made explicit, other programs, even those not written by you, can seamlessly use your data. Similarly, when you write programs that understand semantic data, your programs can operate on datasets that you didn't anticipate when you designed your system.

Data Integration Across the Web

For applications that run on a single machine, documenting the semantics of variables in comments and documentation is adequate. The only people who need to understand the meaning of a variable are the programmers reading the source code. However, when applications participate in larger networks, the meanings of the messages they exchange need to be explicit.

Before the World Wide Web, when a user wanted to use an Internet application, he would install a tool capable of handling specific types of network messages on his machine. If a user wanted to locate users on other networks, he would install an application capable of utilizing the FINGER protocol. If a user wanted to exchange email across a network, he would install an application capable of utilizing the SMTP protocol. Each tool understood the message formats and protocols specific to its task and knew how best to display the information to the user.

Application developers would agree on the format of the messages and the behavior of applications through the circulation of RFC (Request For Comments) documents. These RFCs were written in English and made the semantics of the data contained in the messages explicit, frequently walking the reader through sample data exchanges to eliminate ambiguity. Over time, the developer community would refine the semantics of the messages to improve the capabilities of the applications. RFCs would be amended to reflect the new semantics, and application developers would update applications to make use of the new messages. Eventually users would update the applications on their machines and benefit from these new capabilities.

The emergence of the Web represented a radical change in how most people used the Internet. The Web shielded users from having to think about the applications handling the Internet messages. All you had to do was install a web browser on your machine,

Download at Boykma.Com

and any application on the Web was at your command. For developers, the Web provided a single, simple abstraction for delivering applications and made it possible for an application running in a fixed location and maintained by a stable set of developers to service all Internet users.

Underlying the Web is a set of messages that developers of web infrastructure have agreed to treat in a standard manner. It is well understood that when a web server speaking HTTP receives a GET request, it should send back data corresponding to the path portion of the request message. The semantics of these messages have been thoroughly defined by standards committees and documented in RFCs and W3C recommendations. This standardized infrastructure allows web application developers to operate behind a facade that separates them from the details of how application data is transmitted between machines, and focus on how their applications appear to users. Web application developers no longer need to coordinate with other developers about message formats or how applications should behave in the presence of certain data.

While this facade has facilitated an explosion in applications available to users, the decoupling of data transmission from applications has caused data to become compartmentalized into stovepipe systems, hidden behind web interfaces. The web facade has, in effect, prevented much of the data fueling web applications from being shared and integrated into other Internet applications.

Applications that combine data in new ways and allow users to make connections and understand relationships that were previously hidden are very powerful and compelling. These applications can be as simple as plotting crime statistics on a map or as informative as showing which cuisines are available within walking distance of a film that you want to watch. But currently the process to build these applications is highly specialized and idiosyncratic, with each application using hand-tuned and ad-hoc techniques for harvesting and integrating information due to the hidden nature of data on the Web.

This book introduces repeatable approaches to these data integration problems through the use of simple mechanisms that explicitly expose the semantics of data. These mechanisms provide standardized ways for data to be published and combined, allowing developers to focus on building data-rich applications rather than getting stuck on problems of obtaining and integrating data.

Traditional Data-Modeling Methods

There are many ways to model data, some of them very well researched and mature. In this book we explore new ways to model data, but we're certainly not trying to convince you that the old ways are wrong. There are many ways to think about data, and it is important to have a wide range of tools available so you can pick the best one for the task at hand.

Download at Boykma.Com

In this section, we'll look at common methods that you've likely encountered and consider their strengths and weaknesses when it comes to integrating data across the Web and in the face of quickly changing requirements.

Tabular Data

The simplest kind of dataset, and one that almost everyone is familiar with, is tabular data. Tabular data is any data kept in a table, such as an Excel spreadsheet or an HTML table. Tabular data has the advantage of being very simple to read and manipulate. Consider the restaurant data shown in Table 1-1.

Table 1-1. A table of restaurants

Restaurant	Address	Cuisine	Price	Open
Deli Llama	Peachtree Rd	Deli	$	Mon, Tue, Wed, Thu, Fri
Peking Inn	Lake St	Chinese	$$$	Thu, Fri, Sat
Thai Tanic	Branch Dr	Thai	$$	Tue, Wed, Thu, Fri, Sat, Sun
Lord of the Fries	Flower Ave	Fast Food	$$	Tue, Wed, Thu, Fri, Sat, Sun
Marquis de Salade	Main St	French	$$$	Thu, Fri, Sat
Wok This Way	Second St	Chinese	$	Mon, Tue, Wed, Thu, Fri, Sat, Sun
Luna Sea	Autumn Dr	Seafood	$$$	Tue, Thu, Fri, Sat
Pita Pan	Thunder Rd	Middle Eastern	$$	Mon, Tue, Wed, Thu, Fri, Sat, Sun
Award Weiners	Dorfold Mews	Fast Food	$	Mon, Tue, Wed, Thu, Fri, Sat
Lettuce Eat	Rustic Parkway	Deli	$$	Mon, Tue, Wed, Thu, Fri

Data kept in a table is generally easy to display, sort, print, and edit. In fact, you might not even think of data in an Excel spreadsheet as "modeled" at all, but the placement of the data in rows and columns gives each piece a particular meaning. Unlike the modeling methods we'll see later, there's not really much variation in the ways you might look at tabular data. It's often said that most "databases" used in business settings are simply spreadsheets.

It's interesting to note that there are *semantics* in a data table or spreadsheet: the row and column in which you choose to put the data—for example, a restaurant's cuisine—explains what the name means to a person reading the data. The fact that *Chinese* is in the row *Peking Inn* and in the column *Cuisine* tells us immediately that "Peking Inn serves Chinese food." You know this because you understand what restaurants and cuisines are and because you've previously learned how to read a table. This may seem like a trivial point, but it's important to keep in mind as we explore different ways to think about data.

Download at Boykma.Com

Data stored this way has obvious limitations. Consider the last column, *Open*. You can see that we've crammed a list of days of the week into a single column. This is fine if all we're planning to do is read the table, but it breaks down if we want to add more information such as the open hours or nightly specials. In theory, it's possible to add this information in parentheses after the days, as shown in Table 1-2.

Table 1-2. Forcing too much data into a spreadsheet

Restaurant	Address	Cuisine	Price	Open
Deli Llama	Peachtree Rd	Deli	$	Mon (11a–4p), Tue (11–4), Wed (11–4), Thu (11–7), Fri (11–8)
Peking Inn	Lake St	Chinese	$$$	Thu (5p–10p), Fri (5–11), Sat (5–11)

However, we can't use this data in a spreadsheet program to find the restaurants that will be open late on Friday night. Sorting on the columns simply doesn't capture the deeper meaning of the text we've entered. The program doesn't understand that we've used individual fields in the *Open* column to store multiple distinct information values.

The problems with spreadsheets are compounded when we have multiple spreadsheets that make reference to the same data. For instance, if we have a spreadsheet of our friends' reviews of the restaurants listed earlier, there would be no easy way to search across both documents to find restaurants near our homes that our friends recommend. Although Excel experts can often use macros and lookup tables to get the spreadsheet to approximate this desired behavior, the models are rigid, limited, and usually not changeable by other users.

The need for a more sophisticated way to model data becomes obvious very quickly.

Relational Data

It's almost impossible for a programmer to be unfamiliar with relational databases, which are used in all kinds of applications in every industry. Products like Oracle DB, MySQL, and PostgreSQL are very mature and are the result of years of research and optimization. Relational databases are very fast and powerful tools for storing large sets of data where the data model is well understood and the usage patterns are fairly predictable.

Essentially, a relational database allows multiple tables to be joined in a standardized way. To store our restaurant data, we might define a schema like the one shown in Figure 1-1. This allows our restaurant data to be represented in a more useful and flexible way, as shown in Figure 1-2.

Download at Boykma.Com

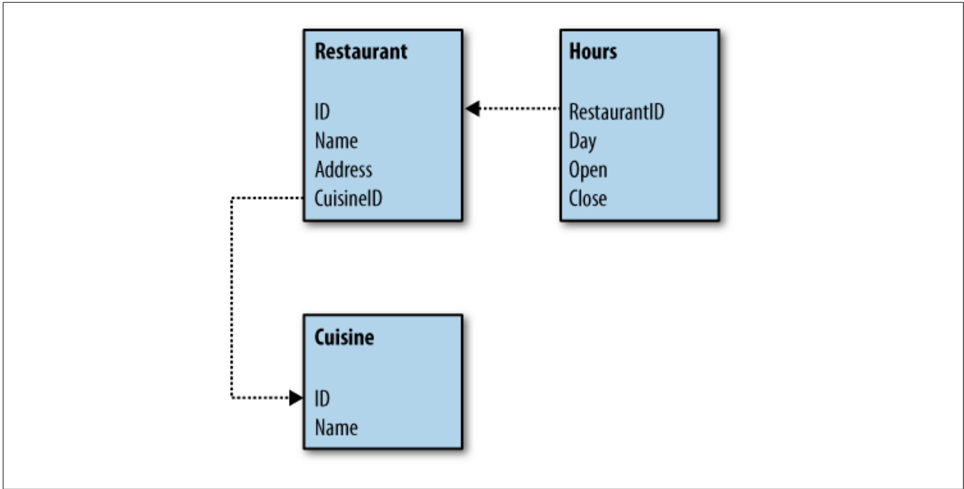


Figure 1-1. Simple restaurant schema

Restaurant

ID	Name	Address	Price	CuisineID
1	Deli Llama	Peachtree Rd	$	1
2	Peking Inn	Lake St	$$$	2

Cuisine

ID	Name
1	Deli
2	Chinese
3	Thai
4	Fast Food

Hours

RestID	Day	Open	Close
1	Mon	11	16
1	Tue	11	16
1	Wed	11	16
1	Thu	11	19
1	Fri	11	20
2	Thu	5	22
2	Fri	5	23
2	Sat	5	23

Figure 1-2. Relational restaurant data

Now, instead of sorting or filtering on a single column, we can do more sophisticated queries. A query to find all the restaurants that will be open at 10 p.m. on a Friday can be expressed using SQL like this:

```
SELECT Restaurant.Name, Cuisine.Name, Hours.Open, Hours.Close
FROM Restaurant, Cuisine, Hours
WHERE Restaurant.CuisineID=Cuisine.ID
AND Restaurant.ID=Hours.RestaurantID
AND Hours.Day="Fri"
AND Hours.Open<22
AND Hours.Close>22
```


Download at Boykma.Com

which gives a result like this:

```
Restaurant.Name    |   Cuisine.Name      |    Hours.Open     |   Hours.Close     |
--------------------------------------------------------------------------
Peking Inn         |   Chinese           |    17             |   23              |
```

Notice that in our relational data model, the semantics of the data have been made more explicit. The meanings of the values are actually described by the schema: someone looking at the tables can immediately see that there are several types of entities modeled—a type called "restaurant" and a type called "days"—and that they have specific relationships between them. Furthermore, even though the database doesn't really know what a "restaurant" is, it can respond to requests to see all the restaurants with given properties. Each datum is labeled with what it means by virtue of the table and column that it's in.

Evolving and Refactoring Schemas

The previous section mentioned that relational databases are great for datasets where the data model is understood up front and there is some understanding of how the data will be used. Many applications, such as product catalogs or contact lists, lend themselves well to relational schemas, since there are generally a fixed set of fields and a set of fairly typical usage patterns.

However, as we've been discussing, data integration across the Web is characterized by rapidly changing types of data, and programmers can never quite know what will be available and how people might want to use it. As a simple example, let's assume we have our restaurant database up and running, and then we receive a new database of bars with additional information not in our restaurant schema, as shown in Table 1-3.

Table 1-3. A new dataset of bars

Bar	Address	DJ	Specialty drink
The Bitter End	14th Ave	No	Beer
Peking Inn	Lake St	No	Scorpion Bowl
Hammer Time	Wildcat Dr	Yes	Hennessey
Marquis de Salade	Main St	Yes	Martini

Of course, many restaurants also have bars, and as it gets later in the evening, they may stop serving food entirely and only serve drinks. The table of bars in this case shows that, in addition to being a French restaurant, Marquis de Salade is also a bar with a DJ. The table also shows specialty drinks, which gives us additional information about Marquis. As of right now, these databases are separate, but it's certainly possible that someone might want to query across them—for example, to find a place to get a French meal and stay later for martinis.

Download at Boykma.Com

So how do we update our database so that it supports the new bar data? Well, we could just link the tables with another table, which has the upside of not forcing us to change the existing structure. Figure 1-3 shows a database structure with an additional table, RB_Link, that links the existing tables, telling you when a restaurant and a bar are actually the same place.

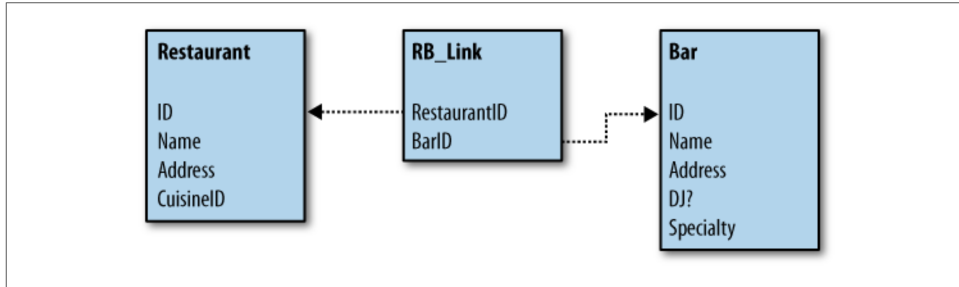


Figure 1-3. Linking bars to the existing schema

This works, and certainly makes our query possible, but it introduces a problem: there are now two names and addresses in our database for establishments that are both bars and restaurants, and just a link telling us that they're the same place. If you want to query by address, you need to look at both tables. Also, the type of food served is attached to the restaurant type, but not to its bar type. Adding and updating data is much more complicated.

Perhaps a more accurate way to model this would be to have a Venue table with bar and restaurant types separated out, like the one shown in Figure 1-4.

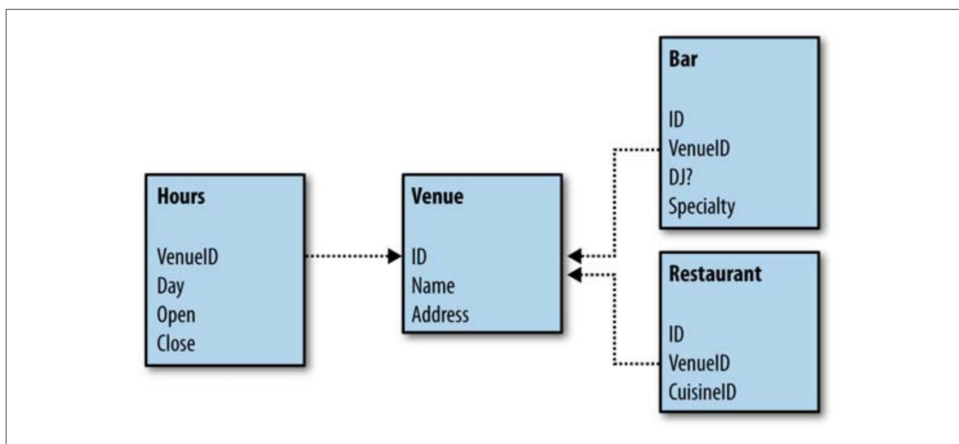


Figure 1-4. Normalized schema that separates a venue from its purposes

Download at Boykma.Com

This seems to solve our issues, but remember that all the existing data is still in our old data model and needs to be transformed to the new data model. This process is called *schema migration* and is often a huge headache. Not only does the data have to be migrated, but all the queries and dependent code that was written assuming a certain table structure have to be changed as well. If we have built a restaurant website on top of our old schema, then we need to figure out how to update all of our existing code, queries, and the database without imposing significant downtime on the website. A whole discipline of software engineering has emerged to deal with these issues, using techniques like stored database procedures and object-relational mapping (ORM) layers to try to decouple the underlying schema from the business-logic layer and lowering the cost of schema changes. These techniques are useful, but they impose their own complexities and problems as well.

It's easy to imagine that, as our restaurant application matures, these venues could also have all kinds of other uses such as a live music hall or a rental space for events. When dealing with data integration across the entire Web, or even in smaller environments that are constantly facing new datasets, migrating the schema each time a new type of data is encountered is simply not tractable. Too often, people have to resort to manual lookups, overly convoluted linked spreadsheets, or just setting the data aside until they can decide what to do with it.

Very Complicated Schemas

In addition to having to migrate as the data evolves, another problem one runs into with relational databases is that the schemas can get incredibly complicated when dealing with many different kinds of data. For example, Figure 1-5 shows a small section of the schema for a Customer Relationship Management (CRM) product.

A CRM system is used to store information about customer leads and relationships with current customers. This is obviously a big application, but to put things in perspective, it is a very small piece of what is required to run a business. An ERP (Enterprise Resource Planning) application, such as SAP, can cover many more of the data needs of a large business. However, the schemas for ERP applications are so inaccessible that there is a whole industry of consulting companies that exclusively deal with them.

The complexity is barely manageable in well-understood industry domains like CRM and ERP, but it becomes even worse in rapidly evolving fields such as biotechnology and international development. Instead of a few long lists of well-characterized data, we instead have hundreds or thousands of datasets, all of which talk about very different things. Trying to normalize these to a single schema is a labor-intensive and painful process.

Download at Boykma.Com

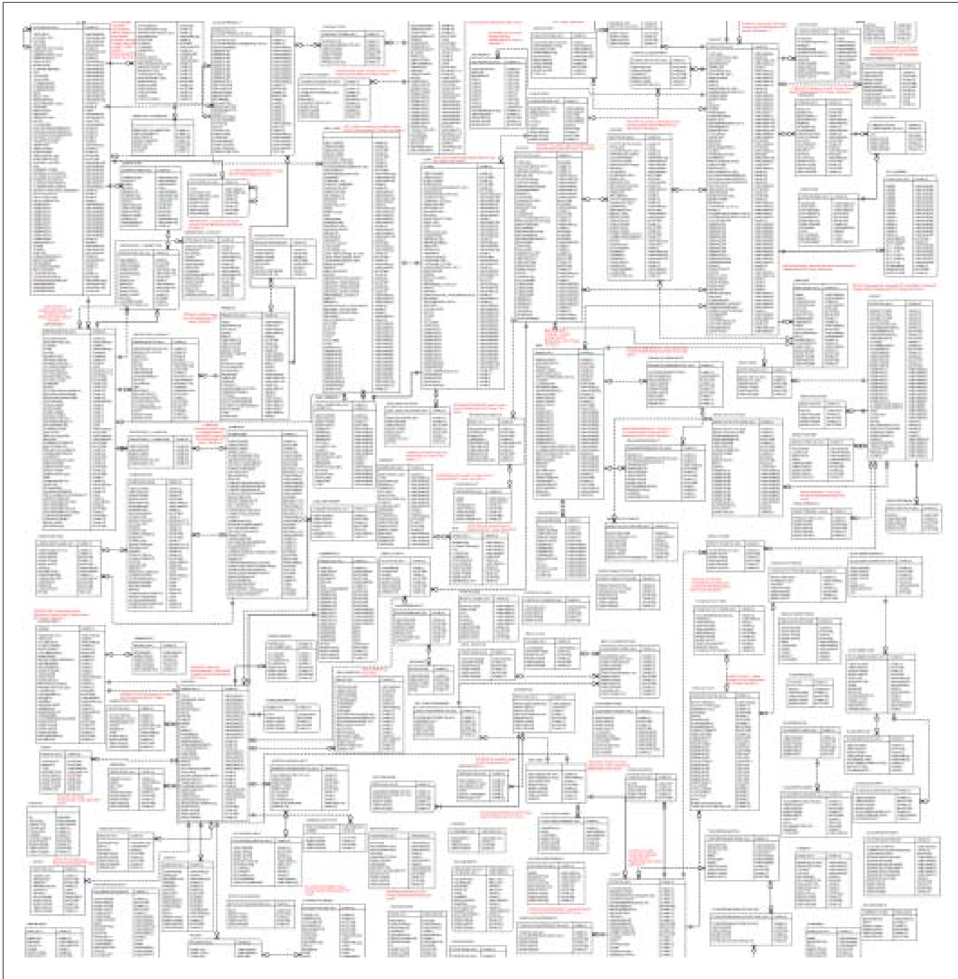


Figure 1-5. Example of a big schema

Getting It Right the First Time

This brings us to the question of whether it's possible to define a schema so that it's flexible enough to handle a wide variety of ever-changing types of data, while still maintaining a certain level of readability. Maybe the schema could be designed to be open to new venue purposes and offer custom fields for them, something like Figure 1-6. The schema has lost the concepts of bars and restaurants entirely, now containing just a list of venues and custom properties for them.

Download at Boykma.Com

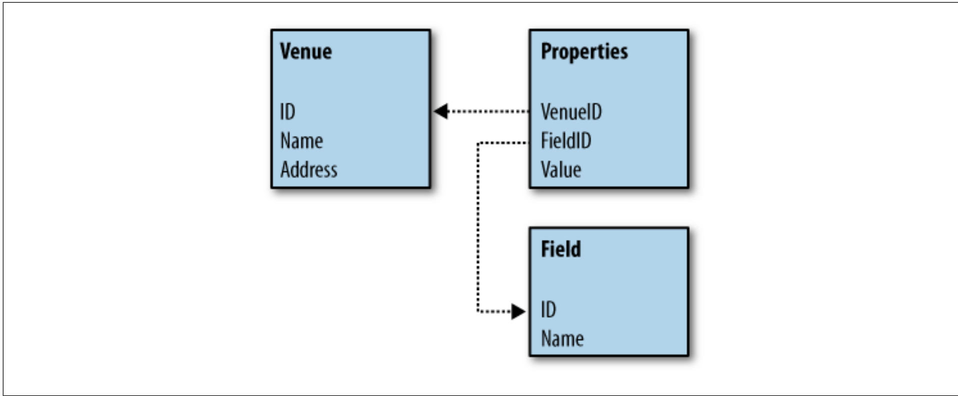


Figure 1-6. Venue schema with completely custom properties

This is not usually recommended, as it gets rid of a lot of the normalization that was possible before and will likely degrade the performance of the database. However, it allows us to express the data in a way that allows for new venue purposes to come along, an example of which is shown in Figure 1-7. Notice how the Properties table contains all the custom information for each of the venues.

Venue

ID	Name	Address
1	Deli Llama	Peachtree Rd
2	Peking Inn	Lake St
3	Thai Tanic	Branch Dr

Properties

VenueID	FieldID	Value
1	1	Deli
1	2	$
2	1	Chinese
2	2	$$$
2	3	Scorpion Bowl
2	4	No

Field

ID	Name
1	Cuisine
2	Price
3	Specialty Cocktail
4	DJ?

Figure 1-7. Venue data in more flexible form

This means that the application can be extended to include, for example, concert venues. Maybe we're visiting a city and looking for a place to stay close to cheap food and cool concert venues. We could create new fields in the Field table, and then add custom properties to any of the existing venues. Figure 1-8 shows an example where we've added the information that Thai Tanic has live jazz music. There are two extra fields, "Live Music?" and "Music Genre", and two more rows in the Properties table.

Download at Boykma.Com

Venue

ID	Name	Address
1	Deli Llama	Peachtree Rd
2	Peking Inn	Lake St
3	Thai Tanic	Branch Dr

Properties

VenueID	FieldID	Value
1	1	Deli
1	2	$
2	1	Chinese
2	2	$$$
2	3	Scorpion Bowl
2	4	No
3	5	Yes
3	6	Jazz

Field

ID	Name
1	Cuisine
2	Price
3	Specialty Cocktail
4	DJ?
5	Live Music
6	Music Genre

Figure 1-8. Adding a concert venue without changing the schema

This type of key/value schema extension is nothing new, and many people stumble into this kind of representation when they have sparse relationships to represent. In fact, many "customizable" data applications such as Saleforce.com represent data this way internally. However, because this type of representation turns the database tables "on their sides," database performance frequently suffers, and therefore it is generally not considered a good idea (i.e., best practice). We've also lost a lot of the normalization we were able to do before, because we've flattened everything to key/value pairs.

Semantic Relationships

Even though it might not be considered a best practice, let's continue with this progression and see what happens. Why not move all the relationships expressed in standard table rows into this parameterized key/value format? From this perspective, the venue name and address are just properties of the venue, so let's move the columns in the Venue table into key/value rows in the Properties table. Figure 1-9 shows what this might look like.

Download at Boykma.Com

Properties		
VenueID	FieldID	Value
1	1	Deli
1	2	$
1	7	Deli Lliama
1	8	Peachtree Rd
2	1	Chinese
2	2	$$$
2	3	Scorpion Bowl
2	4	No
2	7	Peking Inn
2	8	Lake St
3	5	Yes
3	6	Jazz
3	7	Thai Tanic
3	8	Branch Dr

Field	
ID	Name
1	Cuisine
2	Price
3	Specialty Cocktail
4	DJ?
5	Live Music
6	Music Genre
7	Name
8	Address

Figure 1-9. Parameterized venues

That's interesting, but the relationship between the Properties table and the Field table is still only known through the knowledge trapped in the logic of our join query. Let's make that knowledge explicit by preforming the join and displaying the result set in the same parameterized way (Table 1-4).

Table 1-4. Fully parameterized venues

Properties		
VenueID	Field	Value
1	Cuisine	Deli
1	Price	$
1	Name	Deli Llama
1	Address	Peachtree Rd
2	Cuisine	Chinese
2	Price	$$$
2	Specialty Cocktail	Scorpion Bowl
2	DJ?	No
2	Name	Peking Inn
2	Address	Lake St
3	Live Music?	Yes
3	Music Genre	Jazz
3	Name	Thai Tanic
3	Address	Branch Dr

Download at Boykma.Com

Now each datum is described alongside the property that defines it. In doing this, we've taken the semantic relationships that previously were inferred from the table and column and made them data in the table. This is the essence of semantic data modeling: flexible schemas where the relationships are described *by the data itself*. In the remainder of this book, you'll see how you can move *all* of the semantics into the data. We'll show you how to represent data in this manner, and we'll introduce tools especially designed for storing, visualizing, and querying semantic data.

Metadata Is Data

One of the challenges of using someone else's relational data is understanding how the various tables relate to one another. This information—the data about the data representation—is often called metadata and represents knowledge about how the data can be used. This knowledge is generally represented explicitly in the data definition through foreign key relationships, or implicitly in the logic of the queries. Too frequently, data is archived, published, or shared without this critical metadata. While rediscovering these relationships can be an exciting exercise for the user, schemas need not become very large before metadata recovery becomes nearly impossible.

In our earlier example, parameterizing the venue data made the model extremely flexible. When we learn of a new characteristic for a venue, we simply need to add a new row to the table, even if we've never seen that characteristic before. Parameterizing the data also made it trivial to use. You need very little knowledge about the organization of the data to make use of it. Once you know that rows sharing identical VenueIDs relate to one another, you can learn everything there is to know about a venue by selecting all rows with the same VenueID. From this perspective, we can think of the parameterized venue data as "self-describing data." The metadata of the relational schema, describing which columns go together to describe a single entity, has become part of the data itself.

Building for the Unexpected

By packing the data and metadata together in a single representation, we have not only made the schema future-proof, we have also isolated our applications from "knowing" too much about the form of the data. The only thing our application needs to know about the data is that a venue will have an ID in the first column, the properties of the venue appear in the second column, and the third column represents the value of each property. When we add a totally new property to a venue, the application can either choose to ignore the property or handle it in a standard manner.

Because our data is represented in a flexible model, it is easy for someone else to integrate information about espresso machine locations, allowing our application to cover not only restaurants and bars, but also coffee shops, book stores, and gas stations (at least in the greater Seattle area). A well-designed application should be able to

Download at Boykma.Com

seamlessly integrate new semantic data, and semantic datasets should be able to work with a wide variety of applications.

Many content and image creation tools now support XMP (Extensible Metadata Platform) data for tracking information about the author and licensing of creative content. The XMP standard, developed by Adobe Systems, provides a standard set of schemas and allows users to extend the data model in exactly the way we extended the venue data. By using a self-describing model, the tools used to inspect content for XMP data need not change, even if the types of content change significantly in the future. Since image creation tools are fundamentally for creative expression, it is essential that users not be limited to a fixed set of descriptive fields.

"Perpetual Beta"

It's clear that the Web changed the economics of application development. The web facade greatly reduced coordination costs by cutting applications free from the complexity of managing low-level network data messages. With a single application capable of servicing all the users on the Internet, the software development deadlines imposed by manufacturing lead time and channel distribution are quaint memories for most of us. Applications are now free to improve on a continuous and independent basis. Development cycles that update application code on a monthly, weekly, or even daily basis are no longer considered unusual. The phrase "perpetual beta" reflects this sentiment that software is never "frozen" on the Web. As applications continually improve, continuous release processes allow users to instantaneously benefit.

Compressed release cycles are a part of staying competitive at large portal sites. For example, Yahoo! has a wide variety of media sites covering topics such as health, kids, travel, and movies. Content is continually changing as news breaks, editorial processes complete, and users annotate information. In an effort to reduce the time necessary to produce specialized websites and enable new types of personalization and search, Yahoo! has begun to add semantic metadata to their content using extensible schemas not unlike the examples developed here. As data and metadata become one, new applications can add their own annotations without modification to the underlying schema. This freedom to extend the existing metadata enables constantly evolving features without affecting legacy applications, and it allows one application to benefit from the information provided by another.

This shift to continually improving and evolving applications has been accompanied by a greater interest in what were previously considered "scripting" languages such as Python, Perl, and Ruby. The ease of getting something up and running with minimal upfront design and the ease of quick iterations to add new features gives these languages an advantage over heavier static languages that were designed for more traditional approaches to software engineering. However, most frameworks that use these languages still rely on relational databases for storage, and thus still require upfront data

Download at Boykma.Com

modeling and commitment to a schema that may not support the new data sources that future application features require.

So, while perpetual beta is a great benefits to users, rapid development cycles can be a challenge for data management. As new application features evolve, data schemas are frequently forced to evolve as well. As we will see throughout the remainder of this book, flexible semantic data structures and the application patterns that work with them are well designed for life in a world of perpetual beta.

Download at Boykma.Com

CHAPTER 2
Expressing Meaning

In the previous chapter we showed you a simple yet flexible data structure for describing restaurants, bars, and music venues. In this chapter we will develop some code to efficiently handle these types of data structures. But before we start working on the code, let's see if we can make our data structure a bit more robust.

In its current form, our "fully parameterized venue" table allows us to represent arbitrary facts about food and music venues. But why limit the table to describing just these kinds of items? There is nothing specific about the form of the table that restricts it to food and music venues, and we should be able to represent facts about other entities in this same three-column format.

In fact, this three-column format is known as a *triple*, and it forms the fundamental building block of semantic representations. Each triple is composed of a subject, a predicate, and an object. You can think of triples as representing simple linguistic statements, with each element corresponding to a piece of grammar that would be used to diagram a short sentence (see Figure 2-1).

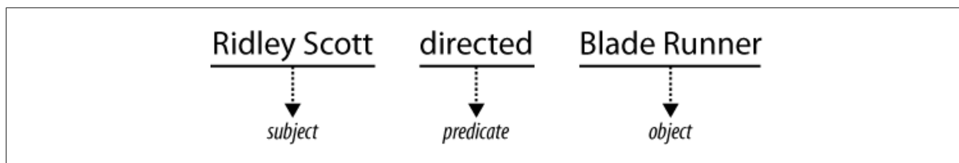


Figure 2-1. Sentence diagram showing a subject-predicate-object relationship

Generally, the subject in a triple corresponds to an entity—a "thing" for which we have a conceptual class. People, places, and other concrete objects are entities, as are less concrete things like periods of time and ideas. Predicates are a property of the entity to which they are attached. A person's name or birth date or a business's stock symbol or mailing address are all examples of predicates. Objects fall into two classes: entities that can be the subject in other triples, and literal values such as strings or numbers.

Multiple triples can be tied together by using the same subjects and objects in different triples, and as we assemble these chains of relationships, they form a directed graph.

Download at Boykma.Com

Directed graphs are well-known data structures in computer science and mathematics, and we'll be using them to represent our data.

Let's apply our graph model to our venue data by relaxing the meaning of the first column and asserting that IDs can represent any entity. We can then add neighborhood information to the same table as our restaurant data (see Table 2-1).

Table 2-1. Extending the Venue table to include neighborhoods

Subject	Predicate	Object
S1	Cuisine	"Deli"
S1	Price	"$"
S1	Name	"Deli Llama"
S1	Address	"Peachtree Rd"
S2	Cuisine	"Chinese"
S2	Price	"$$$"
S2	Specialty Cocktail	"Scorpion Bowl"
S2	DJ?	"No"
S2	Name	"Peking Inn"
S2	Address	"Lake St"
S3	Live Music?	"Yes"
S3	Music Genre	"Jazz"
S3	Name	"Thai Tanic"
S3	Address	"Branch Dr"
S4	Name	"North Beach"
S4	Contained-by	"San Francisco"
S5	Name	"SOMA"
S5	Contained-by	"San Francisco"
S6	Name	"Gourmet Ghetto"
S6	Contained-by	"Berkeley"

Now we have venues and neighborhoods represented using the same model, but nothing connects them. Since objects in one triple can be subjects in another triple, we can add assertions that specify which neighborhood each venue is in (see Table 2-2).

Table 2-2. The triples that connect venues to neighborhoods

Subject	Predicate	Object
S1	Has Location	S4
S2	Has Location	S6
S3	Has Location	S5

Download at Boykma.Com

Figure 2-2 is a diagram of some of our triples structured as a graph, with subjects and objects as nodes and predicates as directed arcs.

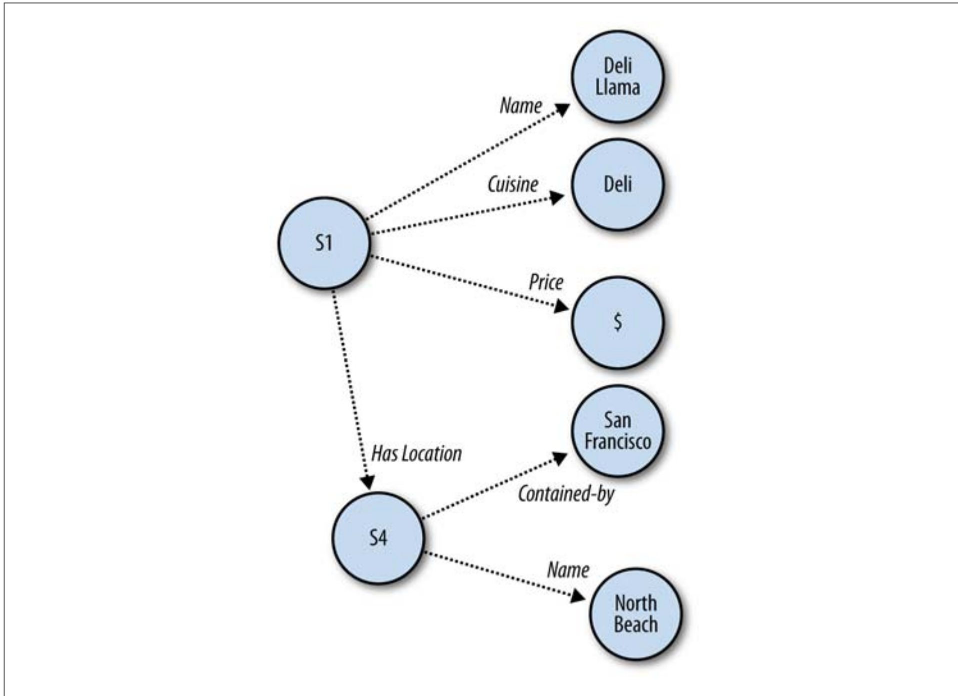


Figure 2-2. A graph of triples showing information about a restaurant

Now, by following the chain of assertions, we can determine that it is possible to eat cheaply in San Francisco. You just need to know where to look.

An Example: Movie Data

We can use this triple model to build a simple representation of a movie. Let's start by representing the title of the movie *Blade Runner* with the triple (`blade_runner name "Blade Runner"`). You can think of this triple as an arc representing the predicate called `name`, connecting the subject `blade_runner` to an object, in this case a string, representing the value "Blade Runner" (see Figure 2-3).

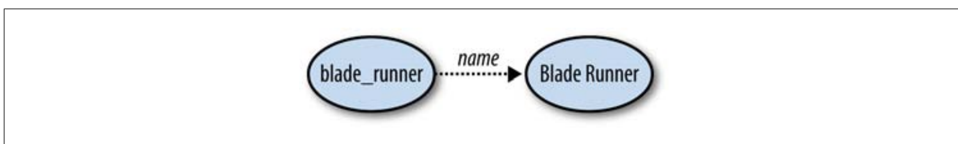


Figure 2-3. A triple describing the title of the movie Blade Runner

Download at Boykma.Com

Now let's add the release date of the film. This can be done with another triple (`blade_runner release_date "June 25, 1982"`). We use the same ID `blade_runner`, which indicates that we're referring to the same subject when making these statements. It is by using the same IDs in subjects and objects that a graph is built—otherwise, we would have a bunch of disconnected facts and no way of knowing that they concern the same entities.

Next, we want to assert that Ridley Scott directed the movie. The simplest way to do this would be to add the triple (`blade_runner directed_by "Ridley Scott"`). There is a problem with this approach, though—we haven't assigned Ridley Scott an ID, so he can't be the source of new assertions, and we can't connect him to other movies he has directed. Additionally, if there happen to be other people named "Ridley Scott", we won't be able to distinguish them by name alone.

Ridley Scott is a person and a director, among other things, and that definitely qualifies as an entity. If we give him the ID `ridley_scott`, we can assert some facts about him: (`ridley_scott name "Ridley Scott"`), and (`blade_runner directed_by ridley_scott`). Notice that we reused the `name` property from earlier. Both entities, "Blade Runner" and "Ridley Scott", have names, so it makes sense to reuse the `name` property as long as it is consistent with other uses. Notice also that we asserted a triple that connected two entities, instead of just recording a literal value. See Figure 2-4.

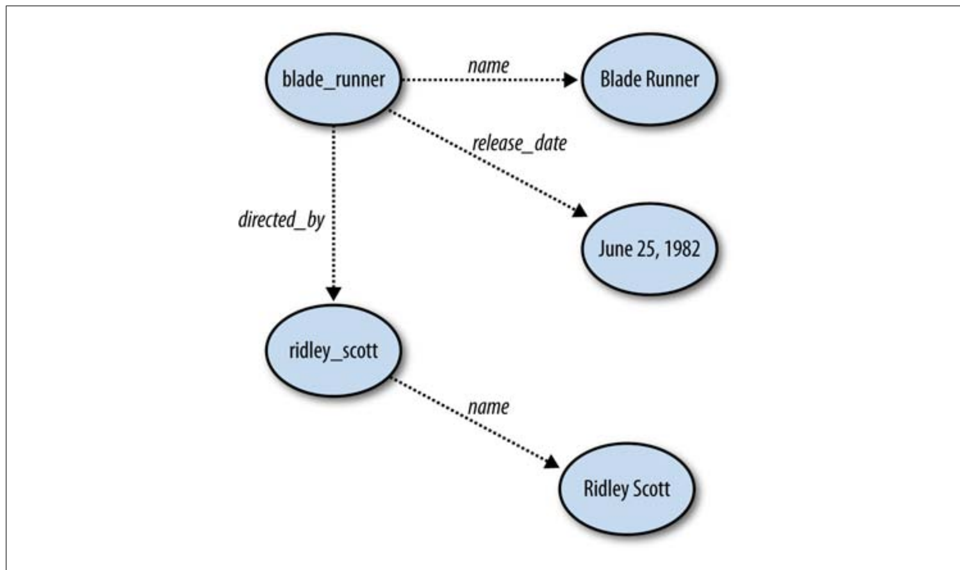


Figure 2-4. A graph describing information about the movie Blade Runner

Download at Boykma.Com

Building a Simple Triplestore

In this section we will build a simple, cross-indexed triplestore. Since there are many excellent semantic toolkits available (which we will explore in more detail in later chapters), there is really no need to write a triplestore yourself. But by working through the scaled-down code in this section, you will gain a better understanding of how these systems work.

Our system will use a common triplestore design: cross-indexing the subject, predicate, and object in three different permutations so that all triple queries can be answered through lookups. All the code in this section is available to download at *http://semprog.com/psw/chapter2/simpletriple.py*. You can either download the code and just read the section to understand what it's doing, or you can work through the tutorial to create the same file.

The examples in this section and throughout the book are in Python. We chose to use Python because it's a very simple language to read and understand, it's concise enough to fit easily into short code blocks, and it has a number of useful toolkits for semantic web programming. The code itself is fairly simple, so programmers of other languages should be able to translate the examples into the language of their choice.

Indexes

To begin with, create a file called *simplegraph.py*. The first thing we'll do is create a class that will be our triplestore and add an initialization method that creates the three indexes:

```
class SimpleGraph:
    def __init__(self):
        self._spo = {}
        self._pos = {}
        self._osp = {}
```

Each of the three indexes holds a different permutation of each triple that is stored in the graph. The name of the index indicates the ordering of the terms in the index (i.e., the `pos` index stores the **p**redicate, then the **o**bject, and then the **s**ubject, in that order). The index is structured using a dictionary containing dictionaries that in turn contain sets, with the first dictionary keyed off of the first term, the second dictionary keyed off of the second term, and the set containing the third terms for the index. For example, the `pos` index could be instantiated with a new triple like so:

```
self._pos = {predicate:{object:set([subject])}}
```

A query for all triples with a specific predicate and object could be answered like so:

```
for subject in self._pos[predicate][object]: yield (subject, predicate, object)
```

Each triple is represented in each index using a different permutation, and this allows any query across the triples to be answered simply by iterating over a single index.

Download at Boykma.Com

The add and remove Methods

The `add` method permutes the subject, predicate, and object to match the ordering of each index:

```
def add(self, (sub, pred, obj)):
    self._addToIndex(self._spo, sub, pred, obj)
    self._addToIndex(self._pos, pred, obj, sub)
    self._addToIndex(self._osp, obj, sub, pred)
```

The `_addToIndex` method adds the terms to the index, creating a dictionary and set if the terms are not already in the index:

```
def _addToIndex(self, index, a, b, c):
    if a not in index: index[a] = {b:set([c])}
    else:
        if b not in index[a]: index[a][b] = set([c])
        else: index[a][b].add(c)
```

The `remove` method finds all triples that match a pattern, permutes them, and removes them from each index:

```
def remove(self, (sub, pred, obj)):
    triples = list(self.triples((sub, pred, obj)))
    for (delSub, delPred, delObj) in triples:
        self._removeFromIndex(self._spo, delSub, delPred, delObj)
        self._removeFromIndex(self._pos, delPred, delObj, delSub)
        self._removeFromIndex(self._osp, delObj, delSub, delPred)
```

The `_removeFromIndex` walks down the index, cleaning up empty intermediate dictionaries and sets while removing the terms of the triple:

```
def _removeFromIndex(self, index, a, b, c):
    try:
        bs = index[a]
        cset = bs[b]
        cset.remove(c)
        if len(cset) == 0: del bs[b]
        if len(bs) == 0: del index[a]
    # KeyErrors occur if a term was missing, which means that it wasn't a
    # valid delete:
    except KeyError:
        pass
```

Finally, we'll add methods for loading and saving the triples in the graph to comma-separated files. Make sure to import the `csv` module at the top of your file:

```
def load(self, filename):
    f = open(filename, "rb")
    reader = csv.reader(f)
    for sub, pred, obj in reader:
        sub = unicode(sub, "UTF-8")
        pred = unicode(pred, "UTF-8")
        obj = unicode(obj, "UTF-8")
        self.add((sub, pred, obj))
    f.close()
```


Download at Boykma.Com

```
def save(self, filename):
    f = open(filename, "wb")
    writer = csv.writer(f)
    for sub, pred, obj in self.triples((None, None, None)):
        writer.writerow([sub.encode("UTF-8"), pred.encode("UTF-8"), \
            obj.encode("UTF-8")])
    f.close()
```

Querying

The basic query method takes a (subject, predicate, object) pattern and returns all triples that match the pattern. Terms in the triple that are set to `None` are treated as wildcards. The `triples` method determines which index to use based on which terms of the triple are wildcarded, and then iterates over the appropriate index, yielding triples that match the pattern:

```
def triples(self, (sub, pred, obj)):
    # check which terms are present in order to use the correct index:
    try:
        if sub != None:
            if pred != None:
                # sub pred obj
                if obj != None:
                    if obj in self._spo[sub][pred]:
                        yield (sub, pred, obj)
                # sub pred None
                else:
                    for retObj in self._spo[sub][pred]:
                         yield (sub, pred, retObj)
            else:
                # sub None obj
                if obj != None:
                    for retPred in self._osp[obj][sub]:
                         yield (sub, retPred, obj)
                # sub None None
                else:
                    for retPred, objSet in self._spo[sub].items():
                        for retObj in objSet:
                            yield (sub, retPred, retObj)
        else:
            if pred != None:
                # None pred obj
                if obj != None:
                    for retSub in self._pos[pred][obj]:
                        yield (retSub, pred, obj)
                # None pred None
                else:
                    for retObj, subSet in self._pos[pred].items():
                        for retSub in subSet:
                            yield (retSub, pred, retObj)
            else:
                # None None obj
                if obj != None:
```


Download at Boykma.Com

```
                for retSub, predSet in self._osp[obj].items():
                    for retPred in predSet:
                        yield (retSub, retPred, obj)
            # None None None
            else:
                for retSub, predSet in self._spo.items():
                    for retPred, objSet in predSet.items():
                        for retObj in objSet:
                            yield (retSub, retPred, retObj)
    # KeyErrors occur if a query term wasn't in the index,
    # so we yield nothing:
    except KeyError:
        pass
```

Now, we'll add a convenience method for querying a single value of a single triple:

```
def value(self, sub=None, pred=None, obj=None):
     for retSub, retPred, retObj in self.triples((sub, pred, obj)):
         if sub is None: return retSub
         if pred is None: return retPred
         if obj is None: return retObj
         break
     return None
```

That's all you need for a basic in-memory triplestore. Although you'll see more sophisticated implementations throughout this book, this code is sufficient for storing and querying all kinds of semantic information. Because of the indexing, the performance will be perfectly acceptable for tens of thousands of triples.

Launch a Python prompt to try it out:

```
>>> from simplegraph import SimpleGraph
>>> movie_graph=SimpleGraph()
>>> movie_graph.add(('blade_runner','name','Blade Runner'))
>>> movie_graph.add(('blade_runner','directed_by','ridley_scott'))
>>> movie_graph.add(('ridley_scott','name','Ridley Scott'))
>>> list(movie_graph.triples(('blade_runner','directed_by',None)))
[('blade_runner', 'directed_by', 'ridley_scott')]
>>> list(movie_graph.triples((None,'name',None)))
[('ridley_scott', 'name', 'Ridley Scott'), ('blade_runner', 'name', 'Blade Runner')]
>>> movie_graph.value('blade_runner','directed_by',None)
ridley_scott
```

Merging Graphs

One of the marvelous properties of using graphs to model information is that if you have two separate graphs with a consistent system of identifiers for subjects and objects, you can merge the two graphs with no effort. This is because nodes and relationships in graphs are first-class entities, and each triple can stand on its own as a piece of meaningful data. Additionally, if a triple is in both graphs, the two triples merge together transparently, because they are identical. Figures 2-5 and 2-6 illustrate the ease of merging arbitrary datasets.

Download at Boykma.Com

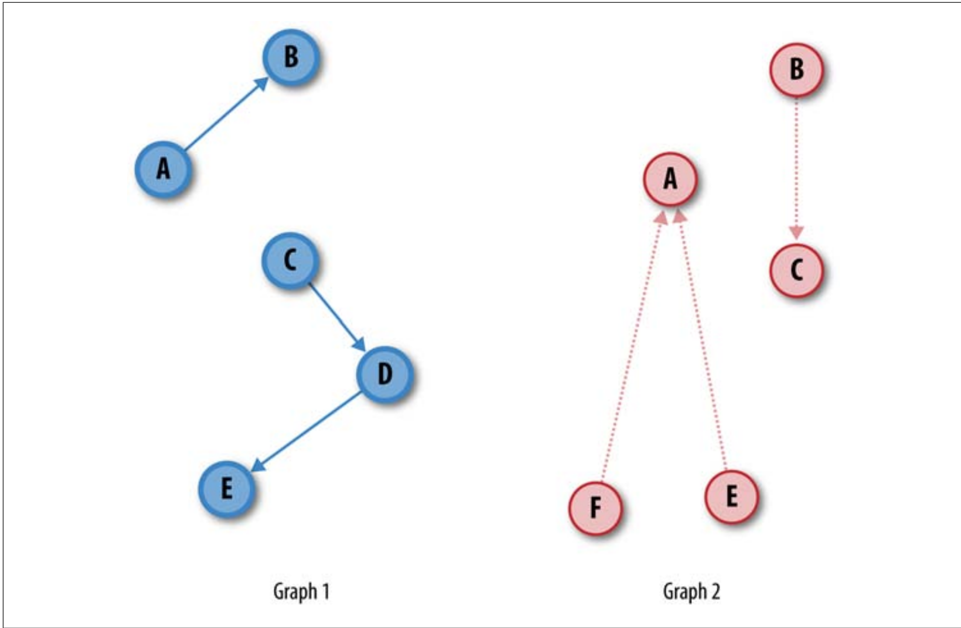


Figure 2-5. Two separate graphs that share some identifiers

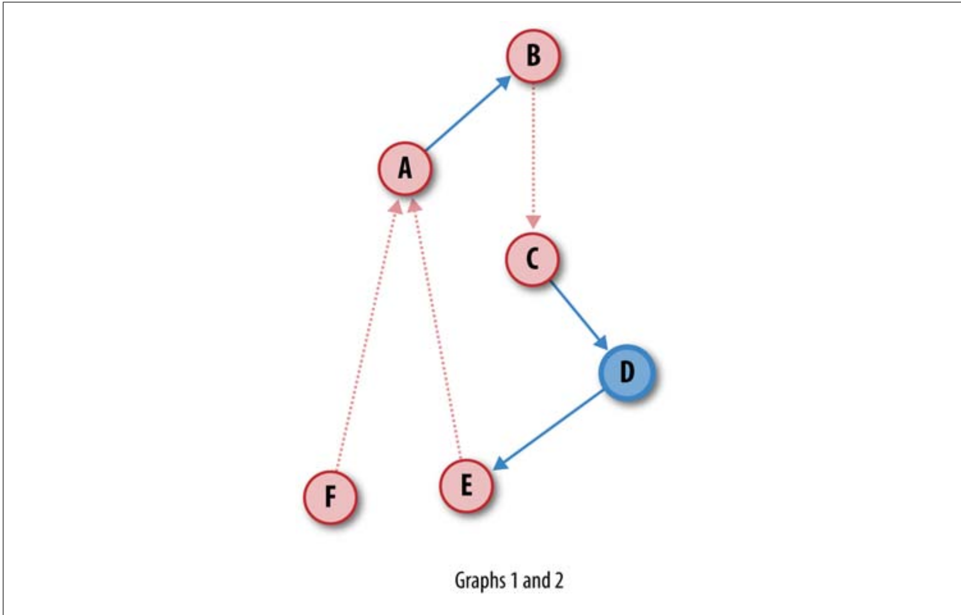


Figure 2-6. The merged graph that is the union of triples

Download at Boykma.Com

In the case of our simple graph, this example will merge two graphs into a single third graph:

```
>>> graph1 = SimpleGraph()
>>> graph2 = SimpleGraph()
... load data into the graphs ...
>>> mergegraph = SimpleGraph()
>>> for sub, pred, obj in graph1:
...     mergegraph.triples((None, None, None)).add((sub, pred, obj))
>>> for sub, pred, obj in graph2:
...     mergegraph.triples((None, None, None)).add((sub, pred, obj))
```

Adding and Querying Movie Data

Now we're going to load a large set of movies, actors, and directors. The *movies.csv* file available at *http://semprog.com/psw/chapter2/movies.csv* contains over 20,000 movies and is taken from Freebase.com. The predicates that we'll be using are `name`, `directed_by` for directors, and `starring` for actors. The IDs for all of the entities are the internal IDs used at Freebase.com. Here's how we load it into a graph:

```
>>> import simplegraph
>>> graph = simplegraph.SimpleGraph()
>>> graph.load("movies.csv")
```

Next, we'll find the names of all the actors in the movie *Blade Runner*. We do this by first finding the ID for the movie named "Blade Runner", then finding the IDs of all the actors in the movie, and finally looking up the names of those actors:

```
>>> bladerunnerId = graph.value(None, "name", "Blade Runner")
>>> print bladerunnerId
/en/blade_runner
>>> bladerunnerActorIds = [actorId for _, _, actorId in \
... graph.triples((bladerunnerId, "starring", None))]
>>> print bladerunnerActorIds
[u'/en/edward_james_olmos', u'/en/william_sanderson', u'/en/joanna_cassidy',
u'/en/harrison_ford', u'/en/rutger_hauer', u'/en/daryl_hannah', ...
>>> [graph.value(actorId, "name", None) for actorId in bladerunnerActorIds]
[u'Edward James Olmos',u'William Sanderson', u'Joanna Cassidy', u'Harrison Ford',
u'Rutger Hauer', u'Daryl Hannah', ...
```

Next, we'll explore what other movies Harrison Ford has been in besides *Blade Runner*:

```
>>> harrisonfordId = graph.value(None, "name", "Harrison Ford")
>>> [graph.value(movieId, "name", None) for movieId, _, _ in \
... graph.triples((None, "starring", harrisonfordId))]
[u'Star Wars Episode IV: A New Hope', u'American Graffiti',
u'The Fugitive', u'The Conversation', u'Clear and Present Danger',...
```

Using Python set intersection, we can find all of the movies in which Harrison Ford has acted that were directed by Steven Spielberg:

```
>>> spielbergId = graph.value(None, "name", "Steven Spielberg")
>>> spielbergMovieIds = set([movieId for movieId, _, _  in \
... graph.triples((None, "directed_by", spielbergId))])
```


Download at Boykma.Com

```
>>> harrisonfordId = graph.value(None, "name", "Harrison Ford")
>>> harrisonfordMovieIds = set([movieId for movieId, _, _  in \
... graph.triples((None, "starring", harrisonfordId))])
>>> [graph.value(movieId, "name", None) for movieId in \
... spielbergMovieIds.intersection(harrisonfordMovieIds)]
[u'Raiders of the Lost Ark', u'Indiana Jones and the Kingdom of the Crystal Skull',
 u'Indiana Jones and the Last Crusade', u'Indiana Jones and the Temple of Doom']
```

It's a little tedious to write code just to do queries like that, so in the next chapter we'll show you how to build a much more sophisticated query language that can filter and retrieve more complicated queries. In the meantime, let's look at a few more graph examples.

Other Examples

Now that you've learned how to represent data as a graph, and worked through an example with movie data, we'll look at some other types of data and see how they can also be represented as graphs. This section aims to show that graph representations can be used for a wide variety of purposes. We'll specifically take you through examples in which the different kinds of information could easily grow.

We'll look at data about places, celebrities, and businesses. In each case, we'll explore ways to represent the data in a graph and provide some data for you to download. All these triples were generated from Freebase.com.

Places

Places are particularly interesting because there is so much data about cities and countries available from various sources. Places also provide context for many things, such as news stories or biographical information, so it's easy to imagine wanting to link other datasets into comprehensive information about locations. Places can be difficult to model, however, in part because of the wide variety of types of data available, and also because there's no clear way to define them—a city's name can refer to its metro area or just to its limits, and concepts like counties, states, parishes, neighborhoods, and provinces vary throughout the world.

Figure 2-7 shows a graph centered around "San Francisco". You can see that San Francisco is in California, and California is in the United States. By structuring the places as a graphical hierarchy we avoid the complications of defining a city as being in a state, which is true in some places but not in others. We also have the option to add more information, such as neighborhoods, to the hierarchy if it becomes available. The figure shows various information about San Francisco through relationships to people and numbers and also the city's geolocation (longitude and latitude), which is important for mapping applications and distance calculations.

Download at Boykma.Com

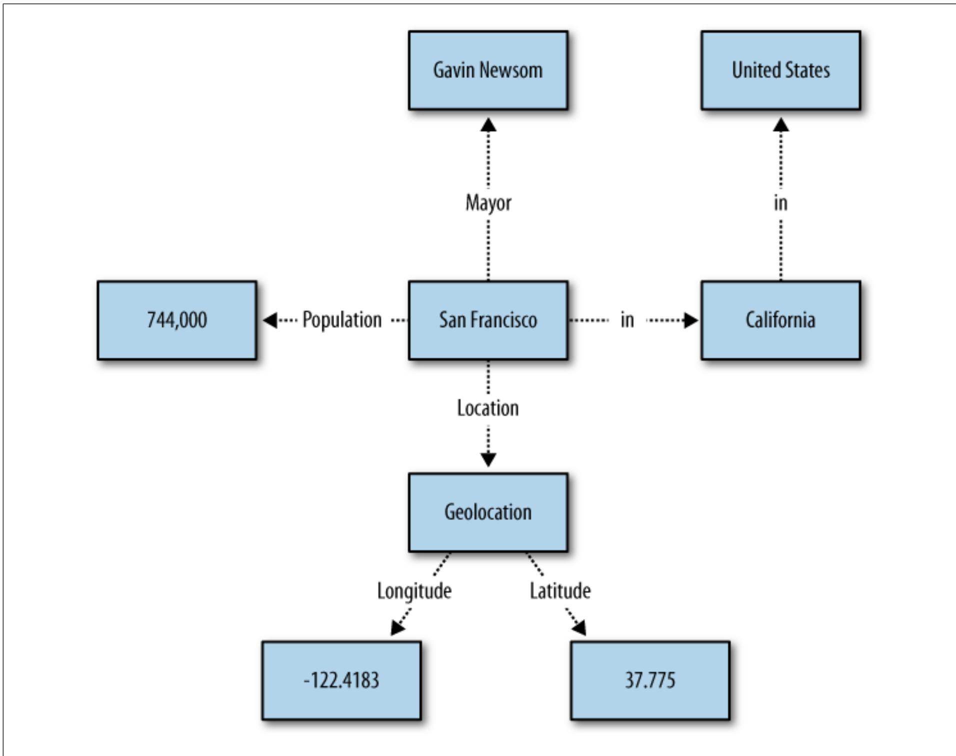


Figure 2-7. An example of location data expressed as a graph

You can download a file containing triples about places from *http://semprog.com/psw/chapter2/place_triples.txt*. In a Python session, load it up and try some simple queries:

```
>>> from simplegraph import SimpleGraph
>>> placegraph=SimpleGraph()
>>> placegraph.loadfile("place_triples.txt")
```

This pattern returns everything we know about San Francisco:

```
>>> for t in placegraph.triples((None,"name","San Francisco")):
...     print t
...
(u'San_Francisco_California', 'name', 'San Francisco')
>>> for t in placegraph.triples(("San_Francisco_California",None,None)):
...     print t
...
('San_Francisco_California', u'name', u'San Francisco')
('San_Francisco_California', u'inside', u'California')
('San_Francisco_California', u'longitude', u'-122.4183')
('San_Francisco_California', u'latitude', u'37.775')
('San_Francisco_California', u'mayor', u'Gavin Newsom')
('San_Francisco_California', u'population', u'744042')
```


Download at Boykma.Com

This pattern shows all the mayors in the graph:

```
>>> for t in placegraph.triples((None,'mayor',None)):
...     print t
...
(u'Aliso_Viejo_California', 'mayor', u'Donald Garcia')
(u'San_Francisco_California', 'mayor', u'Gavin Newsom')
(u'Hillsdale_Michigan', 'mayor', u'Michael Sessions')
(u'San_Francisco_California', 'mayor', u'John Shelley')
(u'Alameda_California', 'mayor', u'Lena Tam')
(u'Stuttgart_Germany', 'mayor', u'Manfred Rommel')
(u'Athens_Greece', 'mayor', u'Dora Bakoyannis')
(u'Portsmouth_New_Hampshire', 'mayor', u'John Blalock')
(u'Cleveland_Ohio', 'mayor', u'Newton D. Baker')
(u'Anaheim_California', 'mayor', u'Curt Pringle')
(u'San_Jose_California', 'mayor', u'Norman Mineta')
(u'Chicago_Illinois', 'mayor', u'Richard M. Daley')

...
```

We can also try something a little bit more sophisticated by using a loop to get all the cities in California and then getting their mayors:

```
>>> cal_cities=[p[0] for p in placegraph.triples((None,'inside','California'))]
>>> for city in cal_cities:
...     for t in placegraph.triples((city,'mayor',None)):
...       print t
...
(u'Aliso_Viejo_California', 'mayor', u'William Phillips')
(u'Chula_Vista_California', 'mayor', u'Cheryl Cox')
(u'San_Jose_California', 'mayor', u'Norman Mineta')
(u'Fontana_California', 'mayor', u'Mark Nuaimi')
(u'Half_Moon_Bay_California', 'mayor', u'John Muller')
(u'Banning_California', 'mayor', u'Brenda Salas')
(u'Bakersfield_California', 'mayor', u'Harvey Hall')
(u'Adelanto_California', 'mayor', u'Charley B. Glasper')
(u'Fresno_California', 'mayor', u'Alan Autry')
(etc...)
```

This is a simple example of joining data in multiple steps. As mentioned previously, the next chapter will show you how to build a simple graph-query language to do all of this in one step.

Celebrities

Our next example is a fun one: celebrities. The wonderful thing about famous people is that other people are always talking about what they're doing, particularly when what they're doing is unexpected. For example, take a look at the graph around the ever-controversial Britney Spears, shown in Figure 2-8.

Download at Boykma.Com

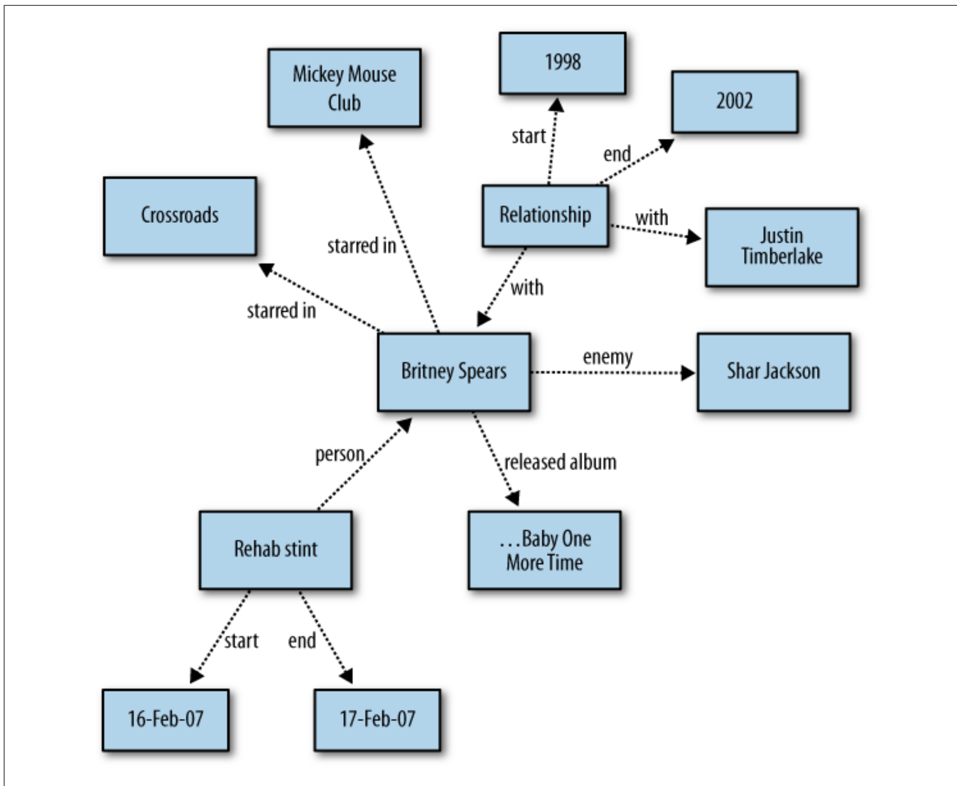


Figure 2-8. An example of celebrity data expressed as a graph

Even from this very small section of Ms. Spears's life, it's clear that there are lots of different things and, more importantly, lots of different *types* of things we say about celebrities. It's almost comical to think that one could frontload the schema design of everything that a famous musician or actress might do in the future that would be of interest to people. This graph has already failed to include such things as favorite nightclubs, estranged children, angry head-shavings, and cosmetic surgery controversies.

We've created a sample file of triples about celebrities at *http://semprog.com/psw/chapter2/celeb_triples.txt*. Feel free to download this, load it into a graph, and try some fun examples:

```
>>> from simplegraph import SimpleGraph
>>> cg=SimpleGraph()
>>> cg.load('celeb_triples.csv')
>>> jt_relations=[t[0] for t in cg.triples((None,'with','Justin Timberlake'))]
>>> jt_relations # Justin Timberlake's relationships
[u'rel373', u'rel372', u'rel371', u'rel323', u'rel16', u'rel15',
 u'rel14', u'rel13', u'rel12', u'rel11']
```


Download at Boykma.Com

```
>>> for rel in jt_relations:
...     print [t[2] for t in cg.triples((rel,'with',None))]
...
[u'Justin Timberlake', u'Jessica Biel']
[u'Justin Timberlake', u'Jenna Dewan']
[u'Justin Timberlake', u'Alyssa Milano']
[u'Justin Timberlake', u'Cameron Diaz']
[u'Justin Timberlake', u'Britney Spears']
[u'Justin Timberlake', u'Jessica Biel']
[u'Justin Timberlake', u'Jenna Dewan']
[u'Justin Timberlake', u'Alyssa Milano']
[u'Justin Timberlake', u'Cameron Diaz']
[u'Justin Timberlake', u'Britney Spears']
>>> bs_movies=[t[2] for t in cg.triples(('Britney Spears','starred_in',None))]
>>> bs_movies # Britney Spears' movies
[u'Longshot', u'Crossroads', u"Darrin's Dance Grooves", u'Austin Powers: Goldmember']
>> movie_stars=set()
>>> for t in cg.triples((None,'starred_in',None)):
...     movie_stars.add(t[0])
...
>>> movie_stars # Anyone with a 'starred_in' assertion
set([u'Jenna Dewan', u'Cameron Diaz', u'Helena Bonham Carter', u'Stephan Jenkins',
u'Pen\xe9lope Cruz', u'Julie Christie', u'Adam Duritz', u'Keira Knightley',
(etc...)
```

As an exercise, see if you can write Python code to answer some of these questions:

- Which celebrities have dated more than one movie star?
- Which musicians have spent time in rehab? (Use the `person` predicate from rehab nodes.)
- Think of a new predicate to represent fans. Add a few assertions about stars of whom you are a fan. Now find out who your favorite stars have dated.

Hopefully you're starting to get a sense of not only how easy it is to add new assertion types to a triplestore, but also how easy it is to try things with assertions created by someone else. You can start asking questions about a dataset from a single file of triples. The essence of semantic data is ease of extensibility and ease of sharing.

Business

Lest you think that semantic data modeling is all about movies, entertainment, and celebrity rivalries, Figure 2-9 shows an example with a little more gravitas: data from the business world. This graph shows several different types of relationships, such as company locations, revenue, employees, directors, and political contributions.

Download at Boykma.Com

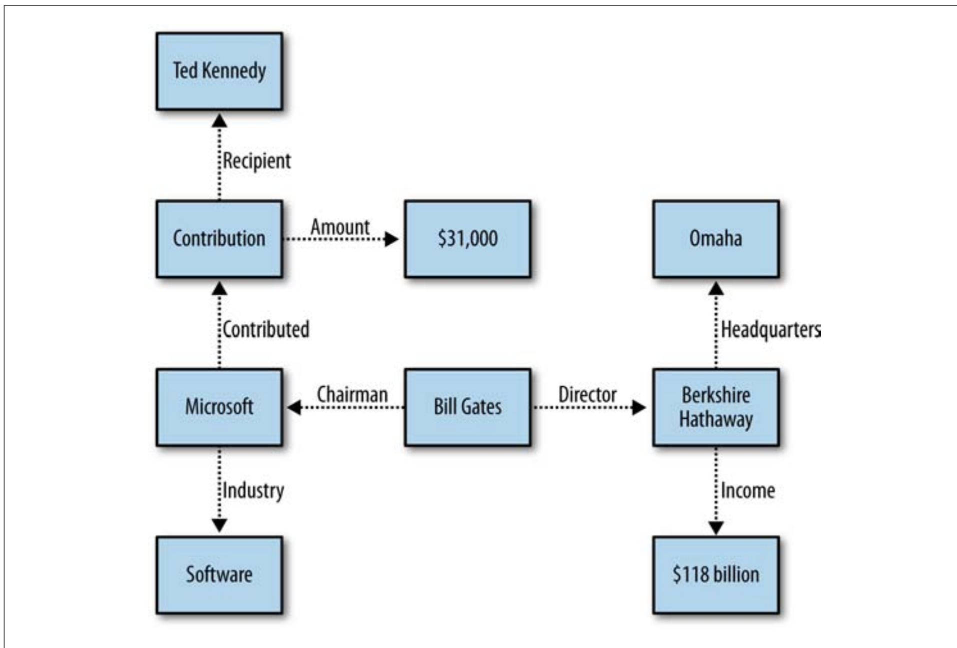


Figure 2-9. An example of business data expressed as a graph

Obviously, a lot of information that is particular to these businesses could be added to this graph. The relationships shown here are actually quite generic and apply to most companies, and since companies can do so many different things, it's easy to imagine more specific relationships that could be represented. We might, for example, want to know what investments Berkshire Hathaway has made, or what software products Microsoft has released. This just serves to highlight the importance of a flexible schema when dealing with complex domains such as business.

Again, we've provided a file of triples for you to download, at *http://semprog.com/psw/chapter2/business_triples.csv*. This is a big graph, with 36,000 assertions about 3,000 companies.

Here's an example session:

```
>>> from simplegraph import SimpleGraph
>>> bg=SimpleGraph()
>>> bg.load('business_triples.csv')

>>> # Find all the investment banks
>>> ibanks=[t[0] for t in bg.triples((None,'industry','Investment Banking'))]
>>> ibanks
[u'COWN', u'GBL', u'CLMS', u'WDR', u'SCHW', u'LM', u'TWPG', u'PNSN', u'BSC', u'GS',
 u'NITE', u'DHIL', u'JEF', u'BLK', u'TRAD', u'LEH', u'ITG', u'MKTX', u'LAB', u'MS',
 u'MER', u'OXPS', u'SF']

>>> bank_contrib={}  # Contribution nodes from Investment banks
```


Download at Boykma.Com

```
>>> for b in ibanks:
...     bank_contrib[b]=[t[0] for t in bg.triples((None,'contributor',b))]

>>> # Contributions from investment banks to politicians
>>> for b,contribs in bank_contrib.items():
...     for contrib in contribs:
...             print [t[2] for t in bg.triples((contrib,None,None))]

[u'BSC', u'30700.0', u'Orrin Hatch']
[u'BSC', u'168335.0', u'Hillary Rodham Clinton']
[u'BSC', u'5600.0', u'Christopher Shays']
[u'BSC', u'5000.0', u'Barney Frank']
(etc...)

>>> sw=[t[0] for t in bg.triples((None,'industry','Computer software'))]
>>> sw
[u'GOOG', u'ADBE', u'IBM',   # Google, Adobe, IBM

>>> # Count locations
>>> locations={}
>>> for company in sw:
...     for t in bg.triples((company,'headquarters',None)):
...             locations[t[2]]=locations.setdefault(t[2],0)+1

>>> # Locations with 3 or more software companies
>>> [loc for loc,c in locations.items() if c>=3]
[u'Austin_Texas', u'San_Jose_California', u'Cupertino_California',
u'Seattle_Washington']
```

Notice that we've used ticker symbols as IDs, since they are guaranteed to be unique and short.

We hope this gives you a good idea of the many different types of data that can be expressed in triples, the different things you might ask of the data, and how easy it is to extend it. Now let's move on to creating a better method of querying.

Download at Boykma.Com

Download at Boykma.Com

CHAPTER 3

Using Semantic Data

So far, you've seen how using explicit semantics can make it easier to share your data and extend your existing system as you get new data. In this chapter we'll show that semantics also makes it easier to develop reusable techniques for querying, exploring, and using data. Capturing semantics in the data itself means that reconfiguring an algorithm to work on a new dataset is often just a matter of changing a few keywords.

We'll extend the simple triplestore we built in Chapter 2 to support constraint-based querying, simple feed-forward reasoning, and graph searching. In addition, we'll look at integrating two graphs with different kinds of data but create separate visualizations of the data using tools designed to work with semantic data.

A Simple Query Language

Up to this point, our query methods have looked for patterns within a single triple by setting the subject, predicate, or object to a wildcard. This is useful, but by treating each triple independently, we aren't able to easily query across a graph of relationships. It is these graph relationships, spanning multiple triples, that we are most interested in working with.

For instance, in Chapter 2 when we wanted to discover which mayors served cities in California, we were forced to run one query to find "cities" (subject) that were "inside" (predicate) "California" (object) and then independently loop through all the cities returned, searching for triples that matched the "city" (subject) and "mayor" (predicate).

To simplify making queries like this, we will abstract this basic query process and develop a simple language for expressing these types of graph relationships. This graph pattern language will form the basis for building more sophisticated queries and applications in later chapters.

Download at Boykma.Com

Variable Binding

Let's consider a fragment of the graph represented by the `places_triples` used in the section "Other Examples" on page 29. We can visualize this graph, representing three cities and the predicates "inside" and "mayor", as shown in Figure 3-1.

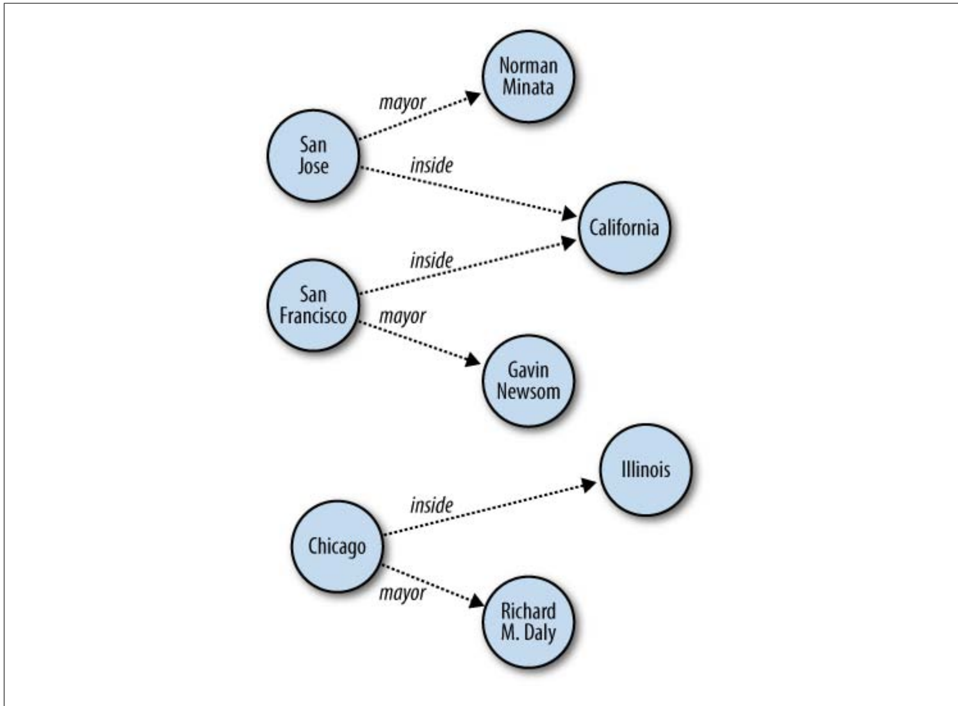


Figure 3-1. A graph of city mayors and locations

Consider the two statements about San Francisco: it is "inside" California, and Gavin Newsom is the mayor. When we express this piece of the graph in triples, we use a common subject identifier `San_Francisco_California` to indicate that the statements describe a single entity:

```
("San_Francisco_California", "inside", "California")
("San_Francisco_California", "mayor", "Gavin Newsom")
```

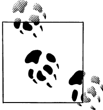
The shared subject identifier `San_Francisco_California` indicates that the two statements are about the same entity, the city of San Francisco. When an identifier is used multiple times in a set of triples, it indicates that a node in the graph is shared by all the assertions. And just as you are free to select any name for a variable in a program, the choice of identifiers used in triples is arbitrary, too. As long as you are consistent in your use of an identifier, the facts that you can learn from the assertions will be clear.

Download at Boykma.Com

For instance, we could have said Gavin Newsom is the mayor of a location within California with the triples:

```
("foo", "inside", "California")
("foo", "mayor", "Gavin Newsom")
```

While `San_Francisco_California` is a useful moniker to help humans understand that "San Francisco" is the location within California that has Newsom as a mayor, the relationships described in the triples using `San_Francisco_California` and `foo` are identical.



It is important not to think of the identifier as the "name" of an entity. Identifiers are simply arbitrary symbols that indicate multiple assertions are related. As in the original `places_triples` dataset in Chapter 2, if we wanted to name the entity where Newsom is mayor, we would need to make the name relationship explicit with another triple:

```
("foo", "name", "San Francisco")
```

With an understanding of how shared identifiers are used to indicate shared nodes in the graph, we can return to our question of which mayors serve in California. As before, we start by asking, "Which nodes participate in an assertion with the predicate of 'inside' and object of 'California'?" Using our current query method, we would write this constraint as this triple query:

```
(None, "inside", "California")
```

and our query method would return the set of triples that matched the pattern.

Instead of using a wildcard, let's introduce a variable named `?city` to collect the identifiers for the nodes in the graph that satisfy our constraints. For the graph fragment pictured in Figure 3-1 and the triple query pattern (`"?city", "inside", "California"`), there are two triples that satisfy the constraints, which gives us two possible values for the variable `?city`: the identifiers `San_Francisco_California` and `San_Jose_California`.

We can express these results as a list of dictionaries mapping the variable name to each matching identifier. We refer to these various possible mappings as "bindings" of the variable `?city` to different values. In our example, the results would be expressed as:

```
[{"?city": "San_Francisco_California"}, {"?city": "San_Jose_California"}]
```

We now have a way to assign results from a triple query to a variable. We can use this to combine multiple triple queries and take the intersection of the individual result sets. For instance, this query specifies the intersection of two triple queries in order to find all cities in California that have a mayor named Norman Mineta:

```
("?city", "inside", "California")
("?city", "mayor", "Norman Mineta")
```


Download at Boykma.Com

The result `[{"?city": "San_Jose_California"}]` is the solution to this graph query because it is the only value for the variable `?city` that is in all of the individual triple query results. We call the individual triple queries in the graph query "constraints" because each triple query constrains and limits the possible results for the whole graph query.

We can also use variables to ask for more results. By adding an additional variable, we can construct a graph query that tells us which mayors serve cities within California:

```
("?city", "inside", "California")
("?city", "mayor", "?name_of_mayor")
```

There are two solutions to this query. In the first solution, the variable `?city` is bound to the identifier `San_Jose_California`, which causes the variable `?name_of_mayor` to be bound to the identifier "Norman Mineta". In the second solution, `?city` will be bound to `San_Francisco_California`, resulting in the variable `?name_of_mayor` being bound to "Gavin Newsom". We can write this solution set as:

```
[{"?city": "San_Francisco_California", "?name_of_mayor": "Gavin Newsom"},
{"?city": "San_Jose_California", "?name_of_mayor": "Norman Mineta"}]
```

It is important to note that all variables in a solution are bound simultaneously—each dictionary represents an alternative and valid set of bindings given the constraints in the graph query.

Implementing a Query Language

In this section, we'll show you how to add variable binding to your existing triplestore. This will allow you to ask the complex questions in the previous chapter with a single method call, instead of doing separate loops and lookups for each step. For example, the following query returns all of the investment banks in New York that have given money to Utah Senator Orrin Hatch:

```
>>> bg.query([('?company','headquarters','New_York_NY'),
              ('?company','industry','Investment Banking'),
              ('?company','contributor','?contribution'),
              ('?contribution','recipient','Orrin Hatch'),
              ('?contribution','amount','?dollars')])
```

The variables are preceded with a question mark, and everything else is considered a constant. This call to the query method tries to find possible values for `company`, `contribution`, and `dollars` that fulfill the following criteria:

1. `company` is headquartered in New York
2. `company` is in the industry Investment Banking
3. `company` made a contribution called `contribution`
4. `contribution` had a recipient called Orrin Hatch
5. `contribution` had an amount equal to `dollars`

Download at Boykma.Com

From the session at the end of Chapter 2, we know that one possible answer is:

```
{'?company':'BSC',
 '?contribution':'contXXX',
 '?dollars':'30700'}
```

If BSC has made multiple contributions to Orrin Hatch, we'll get a separate solution for each contribution.

Before we get to the implementation, just to make sure you're clear, here's another example:

```
>>> cg.query([('?rel1','with','?person'),
              ('?rel1','with','Britney Spears'),
              ('?rel1','end','?year1'),
              ('?rel2','with','?person'),
              ('?rel2','start','?year1')])
```

This asks, "Which person started a new relationship in the same year that their relationship with Britney Spears ended?" The question is a little convoluted, but hopefully it shows the power of querying with variable binding. In this case, we're looking for sets that fulfill the following criteria:

1. `rel1` (a relationship) involved `person`
2. `rel1` also involved Britney Spears
3. `rel1` ended in `year1`
4. `rel2` involved `person`
5. `rel2` started in `year1`

Since there's nothing saying that `person` can't be Britney Spears, it's possible that she could be one of the answers, if she started a new relationship the same year she ended one. As we look at more sophisticated querying languages, we'll see ways to impose negative constraints.

The implementation of `query` is a very simple method for variable binding. It's not super efficient and doesn't do any query optimization, but it will work well on the sets we've been working with and should help you understand how variable binding works. You can add the following code to your existing `simplegraph` class, or you can download *http://semprog.com/psw/chapter3/simplegraph.py*:

```
    def query(self,clauses):
        bindings = None
        for clause in clauses:
            bpos = {}
            qc = []
            for pos, x in enumerate(clause):
                if x.startswith('?'):
                    qc.append(None)
                    bpos[x] = pos
                else:
                    qc.append(x)
            rows = list(self.triples((qc[0], qc[1], qc[2])))
```


Download at Boykma.Com

```
            if bindings == None:
                # This is the first pass, everything matches
                bindings = []
                for row in rows:
                    binding = {}
                    for var, pos in bpos.items():
                        binding[var] = row[pos]
                    bindings.append(binding)
            else:
                # In subsequent passes, eliminate bindings that don't work
                newb = []
                for binding in bindings:
                    for row in rows:
                        validmatch = True
                        tempbinding = binding.copy()
                        for var, pos in bpos.items():
                            if var in tempbinding:
                                if tempbinding[var] != row[pos]:
                                    validmatch = False
                            else:
                                tempbinding[var] = row[pos]
                        if validmatch: newb.append(tempbinding)
                bindings = newb
        return bindings
```

This method loops over each clause, keeping track of the positions of the variables (any string that starts with `?`). It replaces all the variables with `None` so that it can use the `triples` method already defined in `simplegraph`. It then gets all the rows matching the pattern with the variables removed.

For every row in the set, it looks at the positions of the variables in the clause and tries to fit the values to one of the existing bindings. The first time through there are no existing bindings, so every row that comes back becomes a potential binding. After the first time, each row is compared to the existing bindings—if it matches, more variables are added and the binding is added to the current set. If there are no rows that match an existing binding, that binding is removed.

Try using this new query method in a Python session for the two aforementioned queries:

```
>>> from simplegraph import SimpleGraph()
>>> bg = SimpleGraph()
>>> bg.load('business_triples.csv')
>>> bg.query([('?company','headquarters','New_York_New_York'),
              ('?company','industry','Investment Banking'),
              ('?cont','contributor','?company'),
              ('?cont','recipient','Orrin Hatch'),
              ('?cont','amount','?dollars')])
[{'company': u'BSC', 'cont': u'contrib285', 'dollars': u'30700.0'}]
>>> cg = SimpleGraph()
>>> cg.load('celeb_triples.csv')
>>> cg.query([('?rel1','with','?person'),
              ('?rel1','with','Britney Spears'),
              ('?rel1','end','?year1'),
```


Download at Boykma.Com

```
                ('?rel2','with','?person'),
                ('?rel2','start','?year1')])
[{'person': u'Justin Timberlake', 'rel1': u'rel16', 'year1': u'2002',
    'rel2': u'rel372'} ...
```

Now see if you can formulate some queries of your own on the movie graph. For example, which actors have starred in movies directed by Ridley Scott as well as movies directed by George Lucas?

Feed-Forward Inference

Inference is the process of deriving new information from information you already have. This can be used in a number of different ways. Here are a few examples of types of inference:

Simple and deterministic
: If I know a rock weighs 1 kg, I can infer that the same rock weighs 2.2 lbs.

Rule-based
: If I know a person is under 16 and in California, I can infer that they are not allowed to drive.

Classifications
: If I know a company is in San Francisco or Seattle, I can classify it as a "west coast company."

Judgments
: If I know a person's height is 6 feet or more, I refer to them as tall.

Online services
: If I know a restaurant's address, I can use a geocoder to find its coordinates on a map.

Obviously, the definition of what counts as "information" and which rules are appropriate will vary depending on the context, but the idea is that by using rules along with some knowledge and outside services, we can generate new assertions from our existing set of assertions.

The last example is particularly interesting in the context of web development. The Web has given us countless online services that can be queried programmatically. This means it is possible to take the assertions in a triplestore, formulate a request to a web service, and use the results from the query to create new assertions. In this section, we'll show examples of basic rule-based inference and of using online services to infer new triples from existing ones.

Inferring New Triples

The basic pattern of inference is simply to query for information (in the form of bindings) relevant to a rule, then apply a transformation that turns these bindings into a

Download at Boykma.Com

new set of triples that get added back to the triplestore. We're going to create a basic class that defines inference rules, but first we'll add a new method for applying rules to the `SimpleGraph` class. If you downloaded *http://semprog.com/psw/chapter3/simplegraph.py*, you should already have this method. If not, add it to your class:

```
    def applyinference(self,rule):
        queries = rule.getqueries()
        bindings=[]
        for query in queries:
            bindings += self.query(query)
        for b in bindings:
            new_triples = rule.maketriples(b)
            for triple in new_triples:
                self.add(triple)
```

This method takes a rule (usually an instance of `InferenceRule`) and runs its query to get a set of bindings. It then calls `rule.maketriples` on each set of bindings, and adds the returned triples to the store.

The `InferenceRule` class itself doesn't do much, but it serves as a base from which other rules can inherit. Child classes will override the `getquery` method and define a new method called `_maketriples`, which will take each binding as a parameter. You can download *http://semprog.com/psw/chapter3/inferencerule.py*, which contains the class definition of `InferenceRule` and the rest of the code from this section. If you prefer, you can create a new file called *inferencerule.py* and add the following code:

```
class InferenceRule:
    def getqueries(self):
        return []

    def maketriples(self,binding):
        return self._maketriples(**binding)
```

Now we are ready to define a new rule. Our first rule will identify companies headquartered in cities on the west coast of the United States and identify them as such:

```
class WestCoastRule(InferenceRule):
    def getqueries(self):
        sfoquery = [('?company', 'headquarters', 'San_Francisco_California')]
        seaquery = [('?company', 'headquarters', 'Seattle_Washington')]
        laxquery = [('?company', 'headquarters', 'Los_Angelese_California')]
        porquery = [('?company', 'headquarters', 'Portland_Oregon')]
        return [sfoquery, seaquery, laxquery, porquery]

    def _maketriples(self, company):
        return [(company, 'on_coast', 'west_coast')]
```

The rule class lets you define several queries: in this case, there's a separate query for each of the major west coast cities. The variables used in the queries themselves (in this case, just `company`) become the parameters for the call to `_maketriples`. This rule asserts that all the companies found by the first set of queries are on the west coast.

Download at Boykma.Com

You can try this in a session:

```
>>> wcr = WestCoastRule()
>>> bg.applyinference(wcr)
>>> list(bg.triples((None, 'on_coast', None)))
[(u'PCL', 'on_coast', 'west_coast'), (u'PCP', 'on_coast', 'west_coast') ...
```

Although we have four different queries for our west coast rule, each of them has only one clause, which makes the rules themselves very simple. Here's a rule with a more sophisticated query that can be applied to the celebrities graph:

```
class EnemyRule(InferenceRule):
    def getqueries(self):
        partner_enemy = [('?person', 'enemy', '?enemy'),
                         ('?rel', 'with', '?person'),
                         ('?rel', 'with', '?partner')]
        return [partner_enemy]

    def _maketriples(self, person, enemy, rel, partner):
        return (partner, 'enemy', enemy)
```

Just from looking at the code, can you figure out what this rule does? It asserts that if a `person` has a relationship `partner` and also an `enemy`, then their partner has the same enemy. That may be a little presumptuous, but nonetheless it demonstrates a simple logical rule that you can make with this pattern. Again, try it in a session:

```
>>> from simplegraphq import *
>>> cg = SimpleGraph()
>>> cg.load('celeb_triples.csv')
>>> er = EnemyRule()
>>> list(cg.triples((None, 'enemy', None)))
[(u'Jennifer Aniston', 'enemy', u'Angelina Jolie')...
>>> cg.applyinference(er)
>>> list(cg.triples((None, 'enemy', None)))
[(u'Jennifer Aniston', 'enemy', u'Angelina Jolie'), (u'Vince Vaughn', 'enemy', \
    u'Angelina Jolie')...
```

These queries are all a little self-directed, since they operate directly on the data and have limited utility. We'll now see how we can use this reasoning framework to add more information to the triplestore by retrieving it from outside sources.

Geocoding

Geocoding is the process of taking an address and getting the geocoordinates (the longitude and latitude) of that address. When you use a service that shows the location of an address on a map, the address is always geocoded first. Besides displaying on a map, geocoding is useful for figuring out the distance between two places and determining automatically if an address is inside or outside particular boundaries. In this section, we'll show you how to use an external geocoder to infer triples from addresses of businesses.

Download at Boykma.Com

Using a free online geocoder

There are a number of geocoders that are accessible through an API, many of which are free for limited use. We're going to use the site *http://geocoder.us/*, which provides free geocoding for noncommercial use and very cheap access for commercial use for addresses within the United States. The site is based on a Perl script called *Geo::Coder::US*, which can be downloaded from *http://search.cpan.org/~sderle/Geo-Coder-US/* if you're interested in hosting your own geocoding server.

There are several other commercial APIs, such as Yahoo!'s, that offer free geocoding with various limitations and can be used to geocode addresses in many different countries. We've provided alternative implementations of the geocoding rule on our site *http://semprog.com/*.

The free service of geocoder.us is accessed using a REST API through a URL that looks like this:

> *http://rpc.geocoder.us/service/csv?address=1600+Pennsylvania+Avenue,+Washington+DC*

If you go to this URL in your browser, you should get a single line response:

```
38.898748,-77.037684,1600 Pennsylvania Ave NW,Washington,DC,20502
```

This is a comma-delimited list of values showing the latitude, longitude, street address, city, state, and ZIP code. Notice how, in addition to providing coordinates, the geocoder has also changed the address to a more official "Pennsylvania Ave NW" so that you can easily compare addresses that were written in different ways. Try a few other addresses and see how the results change.

Adding a geocoding rule

To create a geocoding rule, just make a simple class that extends `InferenceRule`. The query for this rule finds all triples that have "address" as their predicate; then for each address it contacts the geocoder and tries to extract the latitude and longitude:

```
from urllib import urlopen, quote_plus

class GeocodeRule(InferenceRule):
    def getquery(self):
        return [('?place', 'address', '?address')]

    def _maketriples(self, place, address):
        url = 'http://rpc.geocoder.us/service/csv?address=%s' % quote_plus(address)
        con = urlopen(url)
        data = con.read()
        con.close()
        parts = data.split(',')
        if len(parts) >= 5:
            return [(place, 'longitude', parts[0]),
                    (place, 'latitude', parts[1])]
        else:
```


Download at Boykma.Com

```
            # Couldn't geocode this address
            return []
```

You can then run the geocoder by creating a new graph and putting an address in it:

```
>>> from simplegraph import *
>>> from inferencerule import *
>>> geograph = SimpleGraph()
>>> georule = GeocodeRule()
>>> geograph.add(('White House', 'address', '1600 Pennsylvania Ave, Washington, DC'))
>>> list(geograph.triples((None, None, None))
[('White House', 'address', '1600 Pennsylvania Ave, Washington, DC')]
>>> geograph.applyinference(georule)
>>> list(geograph.triples((None, None, None))
[('White House', 'latitude', '-77.037684'),
 ('White House', 'longitude', '38.898748'),
 ('White House', 'address', '1600 Pennsylvania Ave, Washington, DC')]
```

We've provided a file containing a couple of restaurants in Washington, DC, at *http://semprog.com/psw/chapter3/DC_addresses.csv*. Try downloading the data and running it through the geocoding rule. This may take a minute or two, as access to the geocoder is sometimes throttled. If you're worried it's taking too long, you can add print statements to the geocode rule to monitor the progress.

Chains of Rules

The fact that these inferences create new assertions in the triplestore is incredibly useful to us. It means that we can write inference rules that operate on the results of other rules without explicitly coordinating all of the rules together. This allows the creation of completely decoupled, modular systems that are very robust to change and failure.

To understand how this works, take a look at `CloseToRule`. This takes a place name and a graph in its constructor. It queries for every geocoded item in the graph, calculates how far away they are, and then asserts `close_to` for all of those places that are close by:

```
class CloseToRule(InferenceRule):

    def __init__(self, place, graph):
        self.place = place
        laq = list(graph.triples((place, 'latitude', None)))
        loq = list(graph.triples((place, 'longitude', None)))

        if len(laq) == 0 or len(loq) == 0:
            raise "Exception","%s is not geocoded in the graph" % place

        self.lat = float(laq[0][2])
        self.long = float(loq[0][2])

    def getqueries(self):
        geoq=[('?place', 'latitude', '?lat'), ('?place', 'longitude', '?long')]
        return [geoq]

    def _maketriples(self, place, lat, long):
```


Download at Boykma.Com

```
        # Formula for distance in miles from geocoordinates
        distance=((69.1*(self.lat - float(lat)))**2 + \
            (53*(self.lat - float(lat)))**2)**.5

        # Are they less than a mile apart
        if distance < 1:
            return [(self.place, 'close_to', place)]
        else:
            return [(self.place, 'far_from', place)]
```

Now you can use these two rules together to first geocode the addresses, then create assertions about which places are close to the White House:

```
>>> from simplegraph import *
>>> from inferencerule import *
>>> pg = SimpleGraph()
>>> pg.load('DC_addresses.csv')
>>> georule = GeocodeRule()
>>> pg.applyinference(georule)
>>> whrule = CloseToRule('White House',pg)
>>> pg.applyinference(whrule)
>>> list(pg.triples((None, 'close_to', None)))
[('White House', 'close_to', u'White House'),
 ('White House', 'close_to', u'Pot Belly'),
 ('White House', 'close_to', u'Equinox')]
```

Now we've chained together two rules. Keep this session open while we take a look at another rule, which identifies restaurants that are likely to be touristy. In the `DC_addresses` file, there are assertions about what kind of place each venue is ("tourist attraction" or "restaurant") and also information about prices ("cheap" or "expensive"). The `TouristyRule` combines all this information along with the `close_to` assertions to determine which restaurants are touristy:

```
class TouristyRule(InferenceRule):
    def getqueries(self):
        tr = [('?ta', 'is_a', 'Tourist Attraction'),
              ('?ta', 'close_to', '?restaurant'),
              ('?restaurant', 'is_a', 'restaurant'),
              ('?restaurant', 'cost', 'cheap')]

    def _maketriples(self, ta, restaurant):
        return [(restaurant, 'is_a', 'touristy restaurant')]
```

That is, if a restaurant is cheap and close to a tourist attraction, then it's probably a pretty touristy restaurant:

```
>>> tr = TouristyRule()
>>> pg.applyinference(tr)
>>> list(pg.triples((None, 'is_a', 'touristy restaurant')))
[(u'Pot Belly', 'is_a', 'touristy restaurant')]
```

So we've gone from a bunch of addresses and restaurant prices to predictions about restaurants where you might find a lot of tourists. You can think of this as a group of

Download at Boykma.Com

dependent functions, similar to Figure 3-2, which represents the way we normally think about data processing.

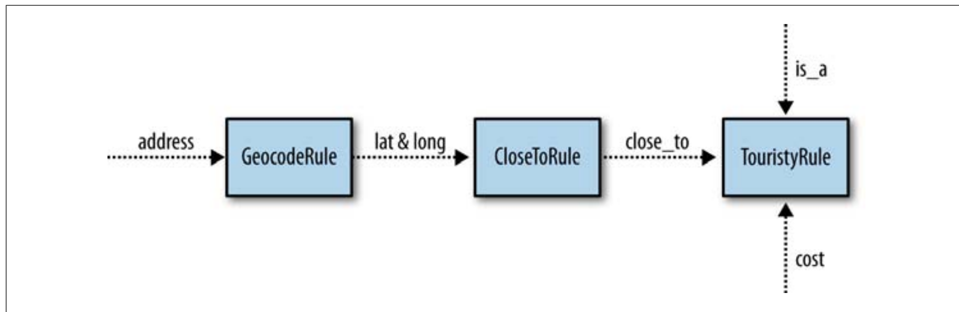


Figure 3-2. A "chain view" of the rules

What's important to realize here is that the rules exist totally independently. Although we ran the three rules in sequence, they weren't aware of each other—they just looked to see if there were any triples that they knew how to deal with and then created new ones based on those. These rules can be run continuously—even from different machines that have access to the same triplestore—and still work properly, and new rules can be added at any time. To help understand what this implies, consider a few examples:

1. Geographical oddities may have a latitude and longitude but no address. They can be put right into the triplestore, and `CloseToRule` will find them without them ever being noticed by `GeocodeRule`.
2. We may invent new rules that add addresses to the database, which will run through the whole chain.
3. We may initially know about a restaurant's existence but not know its cost. In this case, `GeocodeRule` can geocode it, `CloseToRule` can assert that it is close to things, but `TouristyRule` won't be able to do anything with it. However, if we later learn the cost, `TouristyRule` can be activated *without activating the other rules*.
4. We may know that some place is close to another place but not know its exact location. Perhaps someone told us that they walked from the White House to a restaurant. This information can be used by `TouristyRule` without requiring the other rules to be activated.

So a better way to think about this is Figure 3-3.

The rules all share an information store and look for information they can use to generate new information. This is sometimes referred to as a *multi-agent blackboard*. It's a different way of thinking about programming that sacrifices in efficiency but that has the advantages of being very decoupled, easily distributable, fault tolerant, and flexible.

Download at Boykma.Com

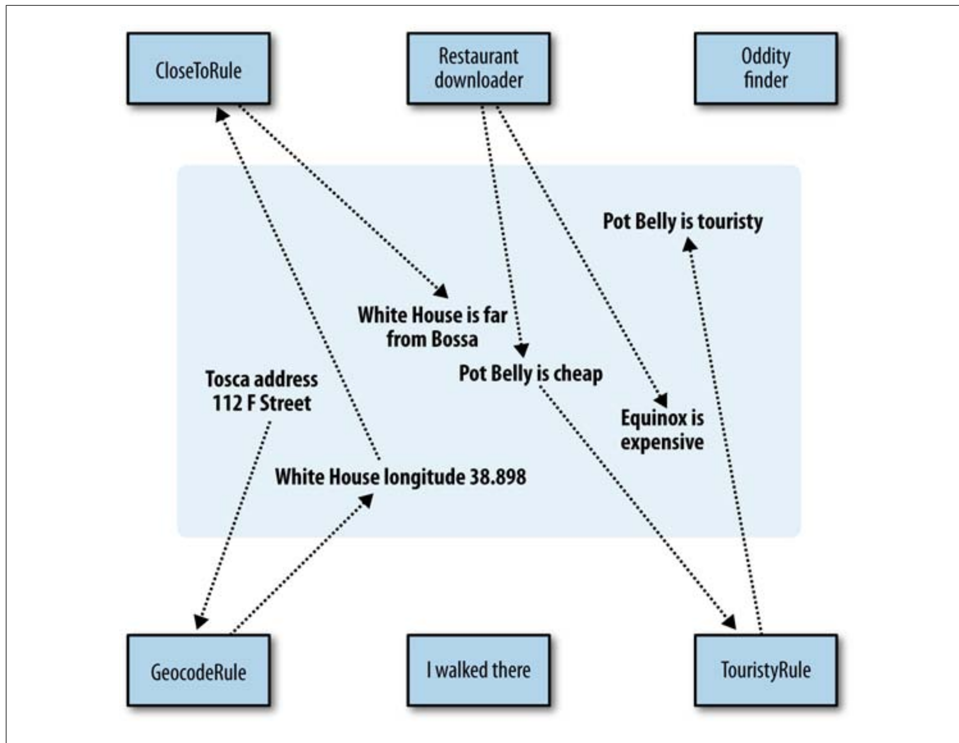


Figure 3-3. Inference rules reading from and writing to a "blackboard"

A Word About "Artificial Intelligence"

It's important to realize, of course, that "intelligence" doesn't come from chains of symbolic logic like this. In the past, many people have made the mistake of trying to generate intelligent behavior this way, attempting to model large portions of human knowledge and reasoning processes using only symbolic logic. These approaches always reveal that there are pretty severe limitations on what can be modeled. Predicates are imprecise, so many inferences are impossible or are only correct in the context of a particular application.

The examples here show triggers for making new assertions entirely from existing ones, and also show ways to query other sources to create new assertions based on existing ones.

Searching for Connections

A common question when working with a graph of data is how two entities are connected. The most common algorithm for finding the shortest path between two points

Download at Boykma.Com

in a graph is an algorithm called *breadth-first search*. The breadth-first search algorithm finds the shortest path between two nodes in a graph by taking the first node, looking at all of its neighbors, looking at all of the neighbors of its neighbors, and so on until the second node is found or until there are no new nodes to look at. The algorithm is guaranteed to find the shortest path if one exists, but it may have to look at all of the edges in the graph in order to find it.

Six Degrees of Kevin Bacon

A good example of the breadth-first search algorithm is the trivia game "Six Degrees of Kevin Bacon," where one player names a film actor, and the other player tries to connect that actor to the actor Kevin Bacon through the shortest path of movies and co-stars. For instance, one answer for the actor Val Kilmer would be that Val Kilmer starred in *Top Gun* with Tom Cruise, and Tom Cruise starred in *A Few Good Men* with Kevin Bacon, giving a path of length 2 because two movies had to be traversed. See Figure 3-4.

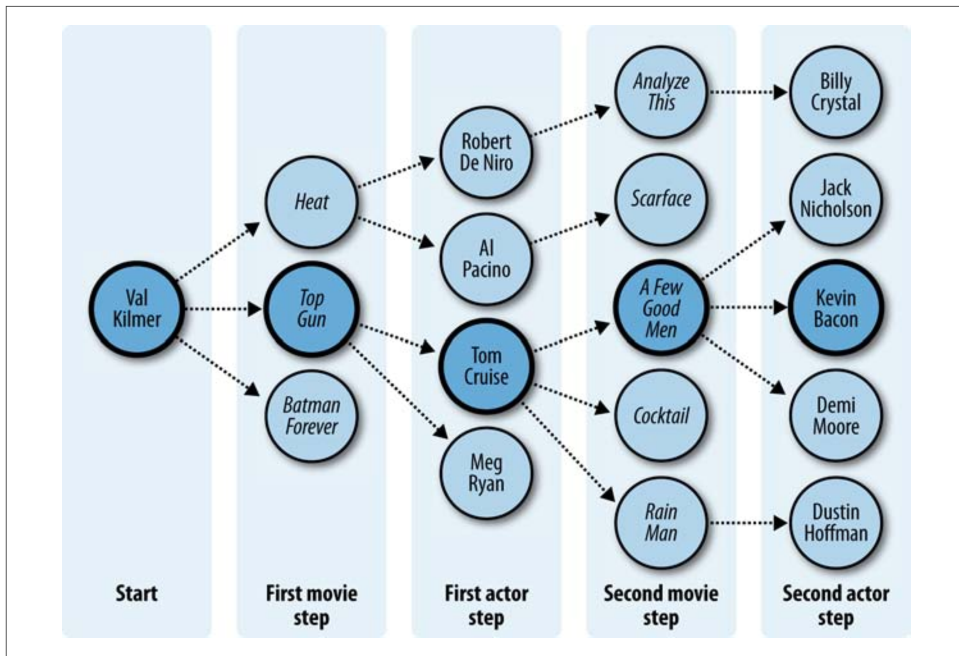


Figure 3-4. Breadth-first search from Val Kilmer to Kevin Bacon

Here's an implementation of breadth-first search over the movie data introduced in Chapter 2. On each iteration of the `while` loop, the algorithm processes a successive group of nodes one edge further than the last. So on the first iteration, the starting actor node is examined, and all of its adjacent movies that have not yet been seen are added to the `movieIds` list. Then all of the actors in those movies are found and are added to

Download at Boykma.Com

the actorIds list if they haven't yet been seen. If one of the actors is the one we are looking for, the algorithm finishes.

The shortest path back to the starting node from each examined node is stored at each step as well. This is done in the "parent" variable, which at each step points to the node that was used to find the current node. This path of "parent" nodes can be followed back to the starting node:

```
def moviebfs(startId, endId, graph):
    actorIds = [(startId, None)]
    # keep track of actors and movies that we've seen:
    foundIds = set()
    iterations = 0
    while len(actorIds) > 0:
        iterations += 1
        print "Iteration " + str(iterations)
        # get all adjacent movies:
        movieIds = []
        for actorId, parent in actorIds:
            for movieId, _, _ in graph.triples((None, "starring", actorId)):
                if movieId not in foundIds:
                    foundIds.add(movieId)
                    movieIds.append((movieId, (actorId, parent)))
        # get all adjacent actors:
        nextActorIds = []
        for movieId, parent in movieIds:
            for _, _, actorId in graph.triples((movieId, "starring", None)):
                if actorId not in foundIds:
                    foundIds.add(actorId)
                    # we found what we're looking for:
                    if actorId == endId: return (iterations, (actorId, \
                        (movieId, parent)))
                    else: nextActorIds.append((actorId, (movieId, parent)))
        actorIds = nextActorIds
    # we've run out of actors to follow, so there is no path:
    return (None, None)
```

Now we can define a function that runs moviebfs and recovers the shortest path:

```
def findpath(start, end, graph):
    # find the ids for the actors and compute bfs:
    startId = graph.value(None, "name", start)
    endId = graph.value(None, "name", end)
    distance, path = moviebfs(startId, endId, graph)
    print "Distance: " + str(distance)
    # walk the parent path back to the starting node:
    names = []
    while path is not None:
        id, nextpath = path
        names.append(graph.value(id, "name", None))
        path = nextpath
    print "Path: " + ", ".join(names)
```


Download at Boykma.Com