4 NoSQL data architecture patterns

This chapter covers

- Key-value stores
- Graph stores
- Column family stores
- Document stores
- Variations of NoSQL architecture patterns

> *...no pattern is an isolated entity. Each pattern can exist in the world only to the extent that is supported by other patterns: the larger patterns in which it is embedded, the patterns of the same size that surround it, and the smaller patterns which are embedded in it.*
>
> —Christopher Alexander, *A Timeless Way of Building*

One of the challenges for users of NoSQL systems is there are many different architectural patterns from which to choose. In this chapter, we'll introduce the most common high-level NoSQL data architecture patterns, show you how to use them, and give you some real-world examples of their use. We'll close out the chapter by looking at some NoSQL pattern variations such as RAM and distributed stores.

Table 4.1 NoSQL data architecture patterns—the most important patterns introduced by the NoSQL movement, brief descriptions, and examples of where these patterns are typically used

Pattern name	Description	Typical uses
Key-value store	A simple way to associate a large data file with a simple text string	Dictionary, image store, document/file store, query cache, lookup tables
Graph store	A way to store nodes and arcs of a graph	Social network queries, friend-of-friends queries, inference, rules system, and pattern matching
Column family (Bigtable) store	A way to store sparse matrix data using a row and a column as the key	Web crawling, large sparsely populated tables, highly-adaptable systems, systems that have high variance
Document store	A way to store tree-structured hierarchical information in a single unit	Any data that has a natural container structure including office documents, sales orders, invoices, product descriptions, forms, and web pages; popular in publishing, document exchange, and document search

Table 4.1 lists the significant data architecture patterns associated with the NoSQL movement.

After reading this chapter, you'll know the main NoSQL data architectural patterns, how to classify the associated NoSQL products and services with a pattern, and the types of applications that use each pattern. When confronted with a new business problem, you'll have a better understanding of which NoSQL pattern might help provide a solution.

We talked about data architecture patterns in chapter 3; to refresh your memory, a *data architecture pattern* is a consistent way of representing data in a structure. This is true for SQL as well as NoSQL patterns. In this chapter, we'll focus on the architectural patterns associated with NoSQL. We'll begin with the simplest NoSQL pattern, the key-value store, and then look at graph stores, column family stores, document stores, and some variations on the NoSQL theme.

4.1 Key-value stores

Let's begin with the *key-value store* and then move on to its variants, and how this pattern is used to cost-effectively solve a variety of business problems. We'll talk about

- What a key-value store is
- Benefits of using a key-value store
- How to use a key-value store in an application
- Key-value store use cases

We'll start by giving you a clear definition.

4.1.1 What is a key-value store?

A *key-value store* is a simple database that when presented with a simple string (the key) returns an arbitrary large BLOB of data (the value). Key-value stores have no query language; they provide a way to add and remove key-value pairs (a combination of key and value where the key is bound to the value until a new value is assigned) into/from a database.

A key-value store is like a dictionary. A dictionary has a list of words and each word has one or more definitions, as shown in figure 4.1.

The dictionary is a simple key-value store where word entries represent keys and definitions represent values. Inasmuch as dictionary entries are sorted alphabetically by word, retrieval is quick; it's not necessary to scan the entire dictionary to find what you're looking for. Like the dictionary, a key-value store is also indexed by the key; the key points directly to the value, resulting in rapid retrieval, regardless of the number of items in your store.

One of the benefits of not specifying a data type for the value of a key-value store is that you can store any data type that you want in the value. The system will store the information as a BLOB and return the same BLOB when a GET (retrieval) request is made. It's up to the application to determine what type of data is being used, such as a string, XML file, or binary image.

The key in a key-value store is flexible and can be represented by many formats:

- Logical path names to images or files
- Artificially generated strings created from a hash of the value
- REST web service calls
- SQL queries

Values, like keys, are also flexible and can be any BLOB of data, such as images, web pages, documents, or videos. See figure 4.2 for an example of a common key-value store.

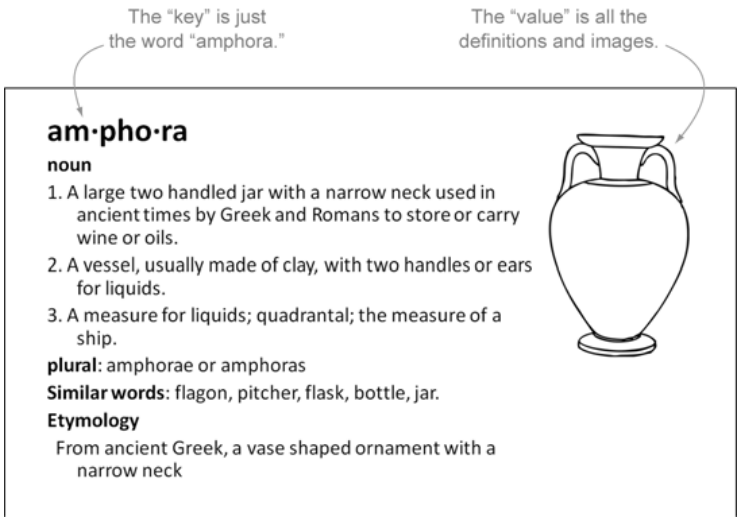

Figure 4.1 A sample dictionary entry showing how a dictionary is similar to a key-value store. In this case, the word you're looking up (amphora) is called the *key* and the definitions are the *values*.

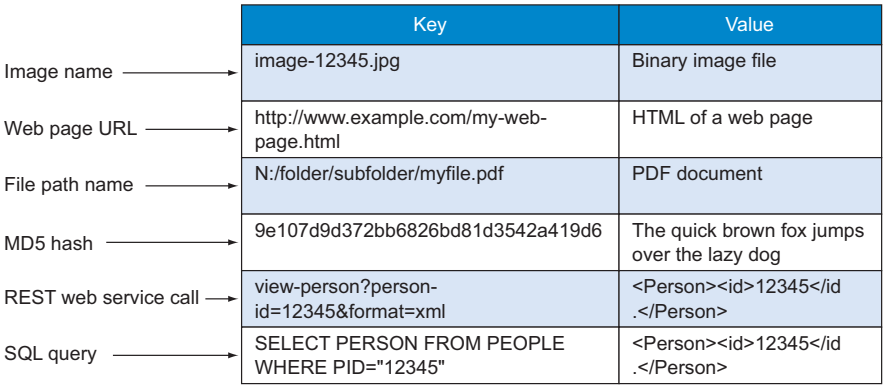

Figure 4.2 Sample items in a key-value store. A key-value store has a key that's associated with a value. Keys and values are flexible. Keys can be image names, web page URLs, or file path names that point to values like binary images, HTML web pages, and PDF documents.

The many names of a key-value store

A key-value store has different names depending on what system or programming language you're using. The process of looking up a stored value using an indexed key for data retrieval is a core-data access pattern that goes back to the earliest days of computing. A key-value store is used in many different computing systems but wasn't formalized as a data architecture pattern until the early 1990s. Popularity increased in 1992, when the open source Berkley DB libraries popularized it by including the key-value store pattern in the free UNIX distribution. Key-value store systems are sometimes referred to as *key-data stores*, since any type of byte-oriented data can be stored as the value. For the application programmer, a structure of an array with two columns is generally called an *associative array* or *map*, and each programming language may call it something slightly different—a hash, a dictionary, or even an object. The current convention, and this text, uses the term key-value store. For more on the history of Berkeley DB, see http://mng.bz/kG9c.

4.1.2 Benefits of using a key-value store

Why are key-value stores so powerful, and why are they used for so many different purposes? To sum it up: their simplicity and generality save you time and money by moving your focus from architectural design to reducing your data services costs through

- Precision service levels
- Precision service monitoring and notification
- Scalability and reliability
- Portability and lower operational costs

PRECISION SERVICE LEVELS

When you have a simple data service interface that's used across multiple applications, you can focus on things like creating precise service levels for data services. A service level doesn't change the API; it only puts precise specifications on how quickly or reliably the service will perform under various load conditions. For example, for any data service you might specify

- The maximum read time to return a value
- The maximum write time to store a new key-value pair
- How many reads per second the service must support
- How many writes per second the service must support
- How many duplicate copies of the data should be created for enhanced reliability
- Whether the data should be duplicated across multiple geographic regions if some data centers experience failures
- Whether to use transaction guarantees for consistency of data or whether eventual consistency is adequate

One of the best ways to visualize how developers control this is to think of a series of knobs and controls similar to what you'd see on a radio tuner, as shown in figure 4.3.

Each input knob can be adjusted to tune the service level that your business needs. Note that as the knobs are adjusted, the estimated monthly cost of providing this data service will change. It can be difficult to precisely estimate the total cost, since the actual cost of running the service is driven by market conditions and other factors, such as the cost for moving data into and out of the service.

You can configure your system to use a simple input form for setting up and allocating resources to new data services. By changing the information using the form, you can quickly change the number of resources allocated to the service. This simple interface allows you to set up new data services and reconfigure data services quickly without the additional overhead of operations staff. Because service levels can be tuned to an application requirement, you can rapidly allocate the appropriate reliability and performance resources to the system.

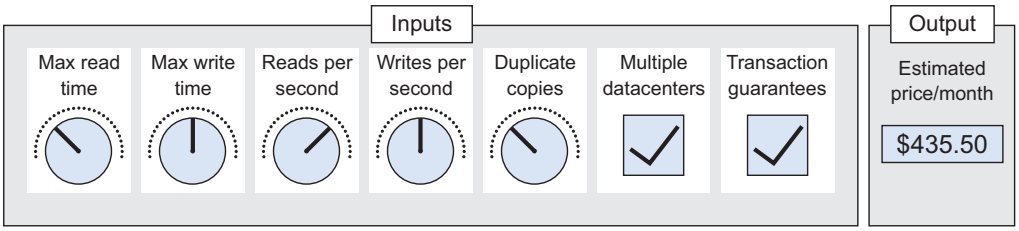

Figure 4.3 NoSQL data services can be adjusted like the tuning knobs on a radio. Each knob can individually be adjusted to control how many resources are used to provide service guarantees. The more resources you use, the higher the cost will be.

PRECISION SERVICE MONITORING AND NOTIFICATION

In addition to specifying service levels, you can also invest in tools to monitor your service level. When you configure the number of reads per second a service performs, setting the parameter too low may mean the user would experience a delay during peak times. By using a simple API, detailed reports showing the expected versus actual loads can point you to system bottlenecks that may need additional resource adjustments.

Automatic notification systems can also trigger email messages when the volume of reads or writes exceeds a threshold within a specified period of time. For example, you may want to send an email notification if the number of reads per second exceeds 80% of some predefined number within a 30-minute period. The email message could contain a link to the monitoring tools as well as links that would allow you to add more servers if the service level was critical to your users.

SCALABILITY AND RELIABILITY

When a database interface is simple, the resulting systems can have higher scalability and reliability. This means you can tune any solution to the desired requirements. Keeping an interface simple allows novice as well as advanced data modelers to build systems that utilize this power. Your only responsibility is to understand how to put this power to work solving business problems.

A simple interface allows you to focus on load and stress testing and monitoring of service levels. Because a key-value store is simple to set up, you can spend more time looking at how long it takes to put or get 10,000 items. It also allows you to share these load- and stress-testing tools with other members of your development team.

PORTABILITY AND LOWER OPERATIONAL COSTS

One of the challenges for information systems managers is to continually look for ways to lower their operational costs of deploying systems. It's unlikely that a single vendor or solution will have the lowest cost for all of your business problems. Ideally, information systems managers would like to annually request data service bids from their database vendors. In the traditional relational database world, this is impractical since porting applications between systems is too expensive compared to the relative savings of hosting your data on a new vendor's system. The more complicated and nonstandardized they are, the less portable they can be and the more difficult moving them to the lowest cost operator is (see figure 4.4).

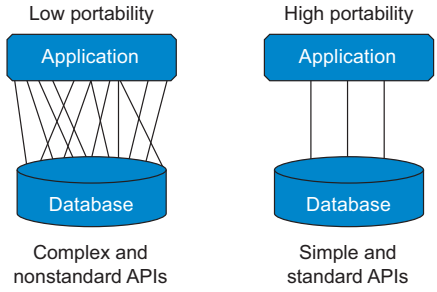

Figure 4.4 Portability of any application depends on database interface complexity. The low-portability system on the left has many complex interfaces between the application and the database, and porting the application between two databases might be a complex process requiring a large testing effort. In contrast, the high-portability application on the right only uses a few standardized interfaces, such as put, get, and delete, and could be quickly ported to a new database with lower testing cost.

Standards watch: complex APIs can still be portable if they're standardized and have portability tests

It's important to note that complex interfaces can still permit high portability. For example, XQuery, a query language used in XML systems, has hundreds of functions, which can be considered a complex application-database interface. But these functions have been standardized by the World Wide Web (W3C) and are still considered to be a low-cost and highly portable application-database interface layer. The W3C provides a comprehensive XQuery test suite to verify if the XQuery interfaces are consistent between implementations. Any XQuery implementation that has over a 99% pass rate allows applications to be ported without significant change.

4.1.3 Using a key-value store

Let's take a look at how an application developer might use a key-value store within an application. The best way to think about using a key-value store is to visualize a single table with two columns. The first column is called the *key* and the second column is called the *value*. There are three operations performed on a key-value store: put, get, and delete. These three operations form the basis of how programmers interface with the key-value store. We call this set of programmer interfaces the *application program interface* or *API*. The key-value interface is summarized in figure 4.5.

Instead of using a query language, application developers access and manipulate a key-value store with the `put`, `get`, and `delete` functions, as shown here:

1. `put($key as xs:string, $value as item())` adds a new key-value pair to the table and will update a value if this key is already present.
2. `get($key as xs:string) as item()` returns the value for any given key, or it may return an error message if there's no key in the key-value store.
3. `delete($key as xs:string)` removes a key and its value from the table, or it many return an error message if there's no key in the key-value store.

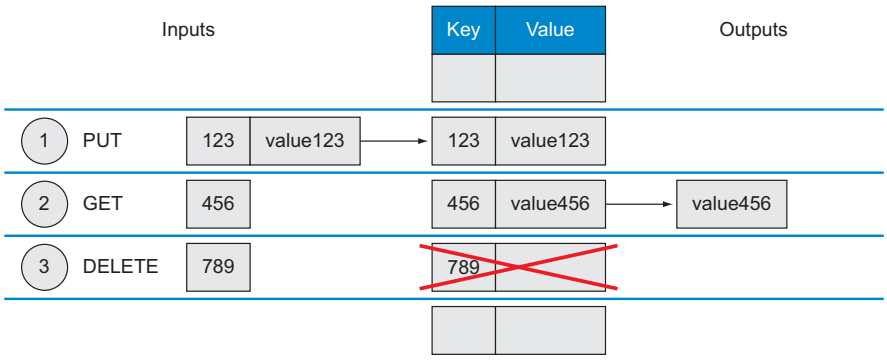

Figure 4.5 The key-value store API has three simple commands: `put`, `get`, and `delete`. This diagram shows how the `put` command inserts the input key `"123"` and value `"value123"` into a new key-value pair; the `get` command presents the key `"456"` and retrieves the value `"value456"`; and the `delete` command presents the key `"789"` and removes the key-value pair.

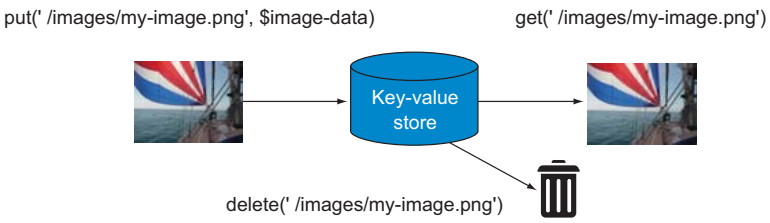

Figure 4.6 The code and result of using the commands associated with a key-value store. To add a new key, you use the `put` command, as shown on the left; to remove, you use the `delete` command, as shown in the middle; and to retrieve, you use the `get` command, as shown on the right.

Figure 4.6 shows how an application would store (`put`), retrieve (`get`), and remove (`delete`) an image from a key-value store.

Standards watch: REST API

Note that we use the verb `put` instead of `add` in a key-value store to align with the standard Representational State Transfer (REST) protocol, a style of software architecture for distributed systems that uses clients to initiate requests and servers to process requests and return responses. The use of `as xs:string` indicates that the key can be any valid string of characters with the exception of binary structures. The `item()` references a single structure that may be a binary file. The `xs:` prefix indicates that the format follows the W3C definition of data types that's consistent with the XML Schema and the closely related XPath and XQuery standards.

In addition to the put, get, and delete API, a key-value store has two rules: distinct keys and no queries on values:

1. *Distinct keys*—You can never have two rows with the same key-value. This means that all the keys in any given key-value store are unique.
2. *No queries on values*—You can't perform queries on the values of the table.

The first rule, distinct keys, is straightforward: if you can't uniquely identify a key-value pair, you can't return a single result. The second rule requires some additional thought if your knowledge base is grounded in traditional relational databases. In a relational database, you can constrain a result set using the `where` clause, as shown in figure 4.7.

A key-value store prohibits this type of operation, as you can't select a key-value pair using the value. The key-value store resolves the issues of indexing and retrieval in large datasets by transferring the association of the key with the value to the application layer, allowing the key-value store to retain a simple and flexible structure. This is an example of the trade-offs between application and database layer complexity we discussed in chapter 2.

There are few restrictions about what you can use as a key as long as it's a reasonably short string of characters. There are also few restrictions about what types of data you can put in the value of a key-value store. As long as your storage system can hold it, you can store it in a key-value store, making this structure ideal for multimedia: images, sounds, and even full-length movies.

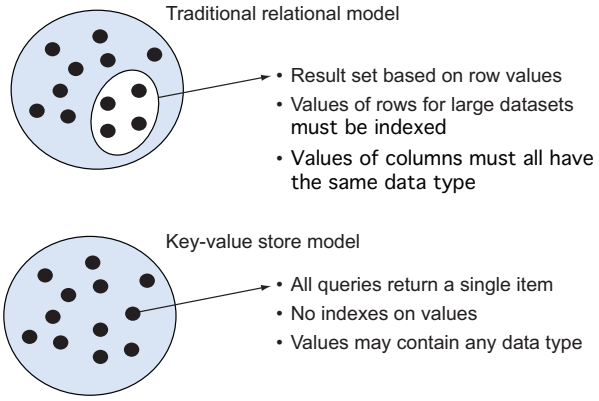

Figure 4.7 The trade-offs associated with traditional relational models (which focus on the database layer) versus key-value store models (which focus on the application layer).

GENERAL PURPOSE

In addition to being simple, a key-value store is a general-purpose tool for solving business problems. It's the Swiss Army knife of databases. What enables the generality is the ability for the application programmer to define what their key structures will be and what type of data they're going to store in the values.

Now that we've looked at the benefits and uses of a key-value store, we'll look at two use case examples. The first, storing web pages in a key-value store, shows how web search engines such as Google easily store entire websites in a key-value store. So if you want to store external websites in your own local database, this type of key-value store is for you.

The second use case, Amazon simple storage service (S3), shows how you can use a key-value store like S3 as a repository for your content in the cloud. If you have digital media assets such as images, music, or video, you should consider using a key-value store to increase the reliability and performance for a fraction of the cost.

4.1.4 Use case: storing web pages in a key-value store

We're all familiar with web search engines, but you may not realize how they work. Search engines like Google use a tool called a *web crawler* to automatically visit a website to extract and store the content of each web page. The words in each web page are then indexed for fast keyword search.

When you use your web browser, you usually enter a web address such as http://www.example.com/hello.html. This *uniform resource locator*, or *URL*, represents the *key* of a website or a web page. You can think of the web as a single large table with two columns, as depicted in figure 4.8.

Key	Value
http://www.example.com/index.html	<html>…
http://www.example.com/about.html	<html>…
http://www.example.com/products.html	<html>…
http://www.example.com/logo.png	Binary…

Figure 4.8 How you can use URLs as a key in a key-value store. Since each web page has a unique URL, you can be assured that no two web pages have the same URL.

The URL is the key, and the value is the web page or resource located at that key. If

all the web pages in only part of the web were stored in a single key-value store system, there might be billions or trillions of key-value pairs. But each key would be unique, like a URL to a web page is unique.

The ability to use a URL as a key allows you to store all of the static or unchanging components of your website in a key-value store. This includes images, static HTML pages, CSS, and JavaScript code. Many websites use this approach, and only the dynamic portions of a site where pages are generated by scripts are *not* stored in the key-value store.

4.1.5 Use case: Amazon simple storage service (S3)

Many organizations have thousands or millions of digital assets they want to store. These assets can include images, sound files, and videos. By using Amazon Simple Storage Service, which is really a key-value store, a new customer can quickly set up a secure web service accessible to anyone as long as they have a credit card.

Amazon S3, launched in the U.S. in March 2006, is an online storage web service that uses a simple REST API interface for storing and retrieving your data, at any time, from anywhere on the web.

At its core, S3 is a simple key-value store with some enhanced features:

- It allows an owner to attach metadata tags to an object, which provides additional information about the object; for example, content type, content length, cache control, and object expiration.
- It has an access control module to allow a bucket/object owner to grant rights to individuals, groups, or everyone to perform put, get, and delete operations on an object, group of objects, or bucket.

At the heart of S3 is the *bucket*. All objects you store in S3 will be in buckets. Buckets store key/object pairs, where the key is a string and the object is whatever type of data you have (like images, XML files, digital music). Keys are unique within a bucket, meaning no two objects will have the same key-value pair. S3 uses the same HTTP REST verbs (PUT, GET, and DELETE) discussed earlier in this section to manipulate objects:

- New objects are added to a bucket using the HTTP PUT message.
- Objects are retrieved from a bucket using the HTTP GET message.
- Objects are removed from a bucket using the HTTP DELETE message.

To access an object, you can generate a URL from the bucket/key combination; for example, to retrieve an object with a key of `gray-bucket` in a bucket called *testbucket*, the URL would be http://testbucket.s3.amazonws.com/gray-bucket .png.

The result on your screen would be the image shown in figure 4.9.

Figure 4.9 This image is the result of performing the http://testbucket.s3.amazonws.com/gray-bucket.png GET request from an Amazon S3 bucket.

In this section, we looked at key-value store systems and how they can benefit an organization, saving them time and money by moving the focus from archi-

tectural design to reducing data services costs. We've demonstrated how these simple and versatile structures can and are used to solve a broad range of business problems for organizations having similar as well as different business requirements. As you attack your next business problem, you'll be able to determine whether a key-value store is the right solution.

Now that you understand the key-value store, let's move to a similar and more complex data architecture pattern: the graph store. As you move through the graph store section, you'll see some similarities to the key-value store as well as different business situations where a using a graph store is the more appropriate solution.

4.2 Graph stores

Graph stores are important in applications that need to analyze relationships between objects or visit all nodes in a graph in a particular manner (graph traversal). Graph stores are highly optimized to efficiently store graph nodes and links, and allow you to query these graphs. Graph databases are useful for any business problem that has complex relationships between objects such as social networking, rules-based engines, creating mashups, and graph systems that can quickly analyze complex network structures and find patterns within these structures.

By the end of this section, you'll be able to identify the key features of a graph store and understand how graph stores are used to solve specific business problems. You'll become familiar with graph terms such as nodes, relationships, and properties, and you'll know about the published W3C standards for graph data. You'll also see how graph stores have been effectively implemented by companies to perform link analysis, use with rules and inference engines, and integrate linked data.

4.2.1 Overview of a graph store

A *graph store* is a system that contains a sequence of nodes and relationships that, when combined, create a graph. You know that in a key-value store there two data fields: the key and the value. In contrast, a graph store has three data fields: *nodes*, *relationships*, and *properties*. Some types of graph stores are referred to as *triple stores* because of their node-relationship-node structure (see figure 4.10).

In the last section, you saw how the structure of a key-value store is general and can be applied to many different situations. This is also true of the basic node-relationship-node structure of a graph store. Graph stores are ideal when you have many items that are related to each other in complex ways and these relationships have properties (like a sister/brother of). Graph stores allow you to do simple queries that show you the nearest neighboring nodes as well as queries that look deep into networks and quickly find patterns. For example, if you use a

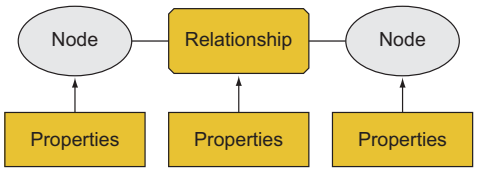

Figure 4.10 A graph store consists of many node-relationship-node structures. Properties are used to describe both the nodes and relationships.

relational database to store your list of friends, you can produce a list of your friends sorted by their last name. But if you use a graph store, you can not only get a list of your friends by their last name, you can also get a list of which friends are most likely to buy you a beer! Graph stores don't just tell you there's a relationship—they can give you detailed reports about each of your relationships.

Graph nodes are usually representations of real-world objects like nouns. Nodes can be people, organizations, telephone numbers, web pages, computers on a network, or even biological cells in a living organism. The relationships can be thought of as connections between these objects and are typically represented as arcs (lines that connect) between circles in diagrams.

Graph queries are similar to traversing nodes in a graph. You can query to ask things like these:

- What's the shortest path between two nodes in a graph?
- What nodes have neighboring nodes that have specific properties?
- Given any two nodes in a graph, how similar are their neighboring nodes?
- What's the average connectedness of various points on a graph with each other?

As you saw in chapter 2, RDBMSs use artificial numbers as primary and foreign keys to relate rows in tables that are stored on different sections of a single hard drive. Performing a join operation in RDBMSs is expensive in terms of latency as well as disk input and output. Graph stores relate nodes together, understanding that two nodes with the same identifiers are the same node. Graph stores assign internal identifiers to nodes and use those identifiers to join networks together. But unlike RDBMSs, graph store joins are computationally lightweight and fast. This speed is attributed to the small nature of each node and the ability to keep graph data in RAM, which means once the graph is loaded into memory, retrieving the data doesn't require disk input and output operations.

Unlike other NoSQL patterns we'll discuss in this chapter, graph stores are difficult to scale out on multiple servers due to the close connectedness of each node in a graph. Data can be replicated on multiple servers to enhance read and query performance, but writes to multiple servers and graph queries that span multiple nodes can be complex to implement.

Although graph stores are built around the simple and general-purpose node-relationship-node data structure, graph stores come with their own complex and inconsistent jargon when they're used in different ways. You'll find that you interact with graph stores in much the same way you do other types of databases. For example, you'll load, query, update, and delete data. The difference is found in the types of queries you use. A graph query will return a set of nodes that are used to create a graph image on the screen to show you the relationship between your data.

Let's take a look and see the variations on terms used to describe different types of graphs.

As you use the web, you'll often see links on a page that take you to another page; these links can be represented by a graph or triple. The current web page is the first or source node, the link is the arc that "points to" the second page, and the second or destination page is the second node. In this example, the first node is represented by the URL of the source page and the second node or destination is the URL of the destination page. This linking process can be found in many places on the web, from page links to wiki sites, where each source and destination node is a page URL. Figure 4.11 is an example of a graph store that has a web page that links to other web pages.

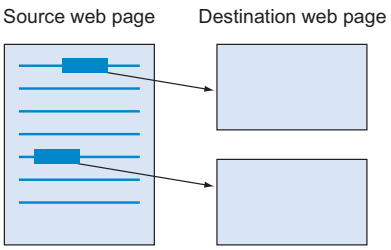

Figure 4.11 An example of using a graph store to represent a web page that contains links to two other web pages. The URL of the source web page is stored as a URL property and each link is a relationship that has a "points to" property. Each link is represented as another node with a property that contains the destination page's URL.

The concept of using URLs to identify nodes is appealing since it's human readable and provides a structure within the URL. The W3C generalized this structure to store the information about the links between pages as well as the links between objects into a standard called *Resource Description Format*, more commonly known as *RDF*.

4.2.2 Linking external data with the RDF standard

In a general-purpose graph store, you can create your own method to determine whether two nodes reference the same point in a graph. Most graph stores will assign internal IDs to each node as they load these nodes into RAM. The W3C has focused on a process of using URL-like identifiers called *uniform resource identifiers (URIs)* to create explicit node identifiers for each node. This standard is called the W3C *Resource Description Format (RDF)*.

RDF was specifically created to join together external datasets created by different organizations. Conceptually, you can load two external datasets into one graph store and then perform graph queries on this joined database. The trick is knowing when two nodes reference the same object. RDF uses directed graphs, where the relationship specifically points from a source node to a destination node. The terminology for the source, link, and destination may vary based on your situation, but in general the terms *subject*, *predicate*, and *object* are used, as shown in figure 4.12.

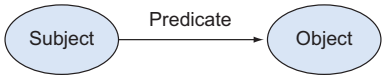

Figure 4.12 How RDF uses specific names for the general node-relationship-node structure. The source node is the subject, and the destination node is the object. The relationship that connects them together is the predicate. The entire structure is called an assertion.

These terms come from formal logic systems and language. This terminology for describing how nodes are identified has been standardized by the W3C in their RDF standard. In RDF each node-arc-node relationship is called a *triple* and is associated

with an assertion of fact. In figure 4.13 the first assertion is (book, has-author, Person123), and the second assertion is (Person123, has-name, “Dan”).

When stored in a graph store, the two statements are independent and may even be stored on different systems around the world. But if the URI of the Person123 structure is the same in both assertions, your application can figure out that the author of the book has a name of "Dan", as shown in figure 4.14.

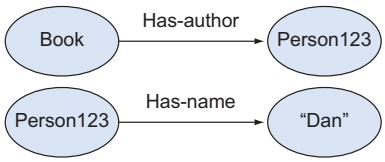

Figure 4.13 Two distinct RDF assertions. The first assertion states that a book has a person as its author. The second assertion shows that this person has a name of Dan. Since the object of the first and the subject of the second have the same URI, they can be joined together.

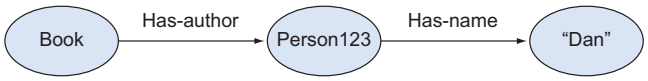

Figure 4.14 How two distinct RDF assertions can be joined together to create a new assertion. From this graph you can answer yes to the question, “Does this book have any author that has the name "Dan"?”

The ability to traverse a graph relies on the fact that two nodes in different groups reference the same physical object. In this example, the Person123 node needs to globally refer to the same item. Once you determine they’re the same, you can join the graphs together. This process is useful in areas like logic inference and complex pattern matching.

As you can imagine, the W3C, who created the RDF standard, is highly motivated to be consistent across all of their standards. Since they already have a method for identifying an HTML page anywhere in the world using a uniform resource locator structure, it makes sense to repurpose these structures whenever possible. The major difference is that, unlike a URL, a URI doesn’t have to point to any actual website or web page. The only criteria is that you must have a way to make them globally consistent across the entire web and match exactly when you compare two nodes.

While a pure triple store is the ideal, in the real world triple stores frequently associate other information with each triple. For example, they might include what group ID the graph belongs to, the date and time the node was created or last updated, or what security groups are associated with the graph. These attributes are frequently called *link metadata* because they describe information about the link itself. Storing this metadata with every node does take more disk space, but it makes the data much easier to audit and manage.

4.2.3 Use cases for graph stores

In this section, we’ll look at situations where a graph store can be used to effectively solve a particular business problem:

- *Link analysis* is used when you want to perform searches and look for patterns and relationships in situations such as social networking, telephone, or email records.
- *Rules and inference* are used when you want to run queries on complex structures such as class libraries, taxonomies, and rule-based systems.
- *Integrating linked data* is used with large amounts of open linked data to do real-time integration and build mashups without storing data.

LINK ANALYSIS

Sometimes the best way to solve a business problem is to traverse graph data—a good example of this is social networking. An example of a social network graph is shown in figure 4.15.

As you add new contacts to your friends list, you might want to know if you have any mutual friends. To get this information, you'd first need to get a list of your friends, and for each one of them get a list of their friends (friends-of-friends). Though you can do this type of search against a relational database, after the initial pass of listing out your friends, the system performance drops dramatically.

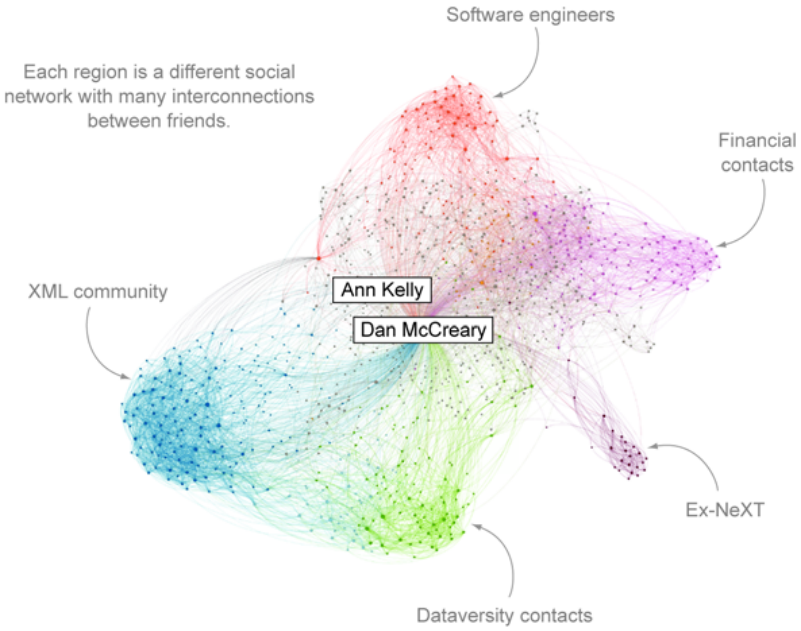

Figure 4.15 A social network graph generated by the LinkedIn InMap system. Each person is represented by a circle, and a line is drawn between two people that have a relationship. People are placed on the graph based on the number of connections they have with all the other people in the graph. People and relationships are shaded the same when there's a high degree of connectivity between the people. Calculating the placement of each person in a social network map is best performed by an in-memory graph traversal program.

Doing this type of analysis using an RDBMS would be slow. In the social networking scenario, you can create a "friends" table for each person with three columns: the ID of the first person, the ID of the second person, and the relationship type (family, close friend, or acquaintance). You can then index that table on both the first and second person, and an RDBMS will quickly return a list of your friends and your friends-of-friends. But in order to determine the next level of relationships, another SQL query is required. As you continue to build out the relationships, the size of each query grows quickly. If you have 100 friends who each have 100 friends, the friends-of-friends query or the second-level friends returns 10,000 (100 x 100) rows. As you might guess, doing this type of query in SQL could be complex.

Graph stores can perform these operations much faster by using techniques that consolidate and remove unwanted nodes from memory. Though graph stores would clearly be much faster for link analysis tasks, they usually require enough RAM to store all the links during analysis.

Graph stores are used for things beyond social networking—they're appropriate for identifying distinct patterns of connections between nodes. For example, creating a graph of all incoming and outgoing phone calls between people in a prison might show a concentration of calls (patterns) associated with organized crime. Analyzing the movement of funds between bank accounts might show patterns of money laundering or credit card fraud. Companies that are under criminal investigation might have all of their email messages analyzed using graph software to see who sent who what information and when. Law firms, law enforcement agencies, intelligence agencies, and banks are the most frequent users of graph store systems to detect legitimate activities as well as for fraud detection.

Graph stores are also useful for linking together data and searching for patterns within large collections of text documents. *Entity extraction* is the process of identifying the most important items (entities) in a document. Entities are usually the nouns in a document like people, dates, places, and products. Once the key entities have been identified, they're used to perform advanced search functions. For example, if you know all the dates and people mentioned in a document, you can create a report that shows which documents mention what people and when.

This entity extraction process (a type of *natural language processing* or *NLP*) can be combined with other tools to extract simple facts or assertions made within a document. For example, the sentence "John Adams was born on October 19, 1735" can be broken into the following assertions:

1. A person record was found with the name of John Adams and is a subject.
2. The born-on relationship links the subject to the object.
3. A date object record was found that has the value of October 19, 1735.

Although simple assertions can be easy to find using simple NLP processing, the process of fully understanding every sentence can be complex and dependant on the context of the situation. Our key takeaway is that if assertions are found in text, they can best be represented in graph structures.

GRAPHS, RULES, AND INFERENCE

The term *rules* can have multiple meanings that depend on where you're coming from and the context of the situation. Here, we use the term to define abstract rules that relate to an understanding of objects in a system, and how the object properties allow you to gain insight into and better use large datasets.

RDF was designed to be a standard way to represent many types of problems in the structure of a graph. A primary use for RDF is to store logic and rules. Once you've set these rules up, you can use an inference or rules engine to discover other facts about a system.

In our section on link analysis, we looked at how text can be encoded with entities such as people and dates to help you find facts. We can now take things a step further to get additional information from the facts that will help you solve business problems.

Let's start with trust, since it's an important aspect for businesses who want to attract and retain customers. Suppose you have a website that allows anyone to post restaurant reviews. Would there be value in allowing you to indicate which reviewers you trust? You're going out to dinner and you're considering two restaurants. Each restaurant has positive and negative reviews. Can you use simple inference to help you decide which restaurant to visit?

As a first test, you could see if your friends reviewed the restaurants. But a more powerful test would be to see if any of your friends-of-friends also reviewed the restaurants. If you trust John and John trusts Sue, what can you infer about your ability to trust Sue's restaurant recommendations? Chances are that your social network will help you use inference to calculate what reviews should have more weight. This is a simple example of using networks, graphs, and inference to gain additional knowledge on a topic. The use of RDF and inference isn't limited to social networks and product reviews. RDF is a general-purpose structure that can be used to store many forms of business logic.

The W3C does more than define RDF; it has an entire framework of standards for using RDF to solve business problems. This framework is frequently referred to as the *Semantic Web Stack*. Some of these are described in figure 4.16.

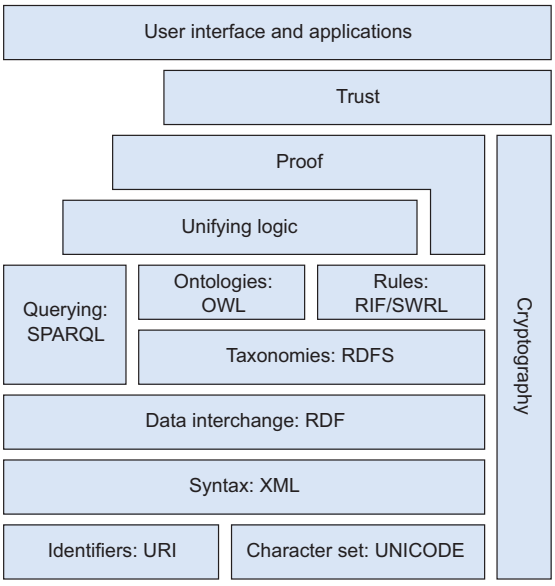

Figure 4.16 A typical semantic web stack with common low-level standards like URI, XML, and RDF at the bottom of the stack. The middle layer includes standards for querying (SPARQL) and standards for rules (RIF/SWRL). At the top of the stack are user interface and application layers above abstract layers of logic, proof, and trust building.

At the bottom of the stack, you see standards that are used in many areas, such as standardized character sets (Unicode) and standards that represent identifiers to objects in a URI-like format. Above that, you see that RDF is stored in XML files, a good example of using the XML tree-like document structure to contain graphs. Above the XML layer you see ways that items are classified using a taxonomy (RDFS) and above this you see the standards for ontologies (OWL) and rules (RIF/SWRL). The SPARQL query language also sits above the RDF layer. Above these areas, you see some of the areas that are still not standardized: logic, proof, and trust. This is where much of the research and development in the Semantic Web is focused. At the top, the user interface layer is similar to the application layers we talked about in chapter 2. Finally, along the side and to the right are cryptography standards that are used to securely exchange data over the public internet.

Many of the tools and languages associated with the upper layers of the Semantic Web Stack are still in research and development, and the number of investment case studies showing a significant ROI remain few and far between. A more practical step is to store original source documents with their extracted entities (annotations) directly in a document store that supports mixed content. We'll discuss these concepts and techniques later in the next chapter when we look at XML data stores.

In the next section, we'll look at how organizations are combining publicly available datasets (linked open data) from domain areas such as media, medical and environmental science, and publications to perform real-time extract, transform, and display operations.

USING GRAPHS TO PROCESS PUBLIC DATASETS

Graph stores are also useful for doing analysis on data that hasn't been created by your organization. What if you need to do analysis with three different datasets that were created by three different organizations? These organizations may not even know each other exists! So how can you automatically join their datasets together to get the information you need? How do you create mashups or recombinations of this data in an efficient way? One answer is by using a set of tools referred to as *linked open data* or *LOD*. You can think of it as an integration technique for doing joins between disparate datasets to create new applications and new insights.

LOD strategies are important for anyone doing research or analysis using publicly available datasets. This research includes topics such as customer targeting, trend analysis, sentiment analysis (the application of NLP, computational linguistics, and text analytics to identify and extract subjective information in source materials), or the creation of new information services. Recombining data into new forms provides opportunities for new businesses. As the amount of LOD grows, there are often new opportunities for new business ventures that combine and enrich this information.

LOD integration creates new datasets by combining information from two or more publicly available datasets that conform to the LOD structures such as RDF and URIs. A figure of some of the popular LOD sites called an *LOD cloud* diagram is shown in figure 4.17.

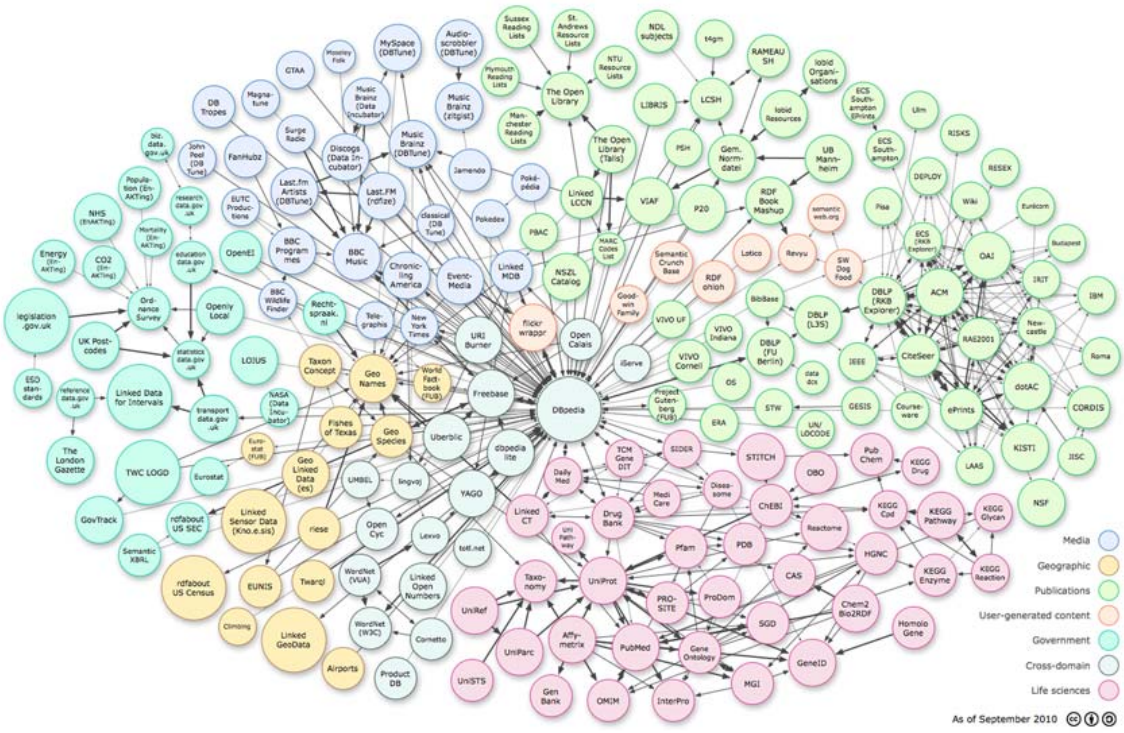

Figure 4.17 **The linked open data cloud is a series of shaded circles that are connected by lines. The shades indicate the domain—for example, darker for geographic datasets, lighter for life sciences. (Diagram by Richard Cyganiak and Anja Jentzsch: http://lod-cloud.net)**

At the center of LOD cloud diagrams you'll see sites that contain a large number of general-purpose datasets. These sites include LOD hub sites such as DBPedia or Freebase. DBPedia is a website that attempts to harvest facts from Wikipedia and convert them into RDF assertions. The data in the info boxes in Wikipedia is a good example of a source of consistent data in wiki format. Due to the diversity of data in DBPedia, it's frequently used as a hub to connect different datasets together.

Once you find a site that has the RDF information you're looking for, you can proceed in two ways. The first is to download *all* the RDF data on the site and load it into your graph store. For large RDF collections like DBPedia that have billions of triples, this can be impracticable. The second and more efficient method is to find a web service for the RDF site called a *SPARQL endpoint*. This service allows you to submit SPARQL queries to extract the data from each of the websites you need in an RDF form that can then be joined with other RDF datasets. By combining the data from SPARQL queries, you can create new data mashups that join data together in the same way joins combine data from two different tables in an RDBMS.

The key difference between a SPARQL query and an RDBMS is the process that creates the primary/foreign keys. In the RDBMS, all of the keys are in the same domain,

but in the LOD the data was created by different organizations, so the only way to join the data is to use consistent URIs to identify nodes.

The number of datasets that participate in the LOD community is large and growing, but as you might guess, there are few ways to guarantee the quality and consistency of public data. If you find inconsistencies and missing data, there's no easy way to create bulk updates to correct the source data. This means you may need to manually edit hundreds of Wiki pages in order to add or correct data. After this is done, you may need to wait till the next time the pages get indexed by the RDF extraction tools. These are challenges that have led to the concept of curated datasets that are based on public data but then undergo a postprocessing cleanup and normalization phase to make the data more usable by organizations.

In this section, we've covered graph representations and shown how organizations are using graph stores to solve business problems. We now move on to our third NoSQL data architecture pattern.

4.3 Column family (Bigtable) stores

As you've seen, key-value stores and graph stores have simple structures that are useful for solving a variety of business problems. Now let's look at how you can combine a row and column from a table to use as the key.

Column family systems are important NoSQL data architecture patterns because they can scale to manage large volumes of data. They're also known to be closely tied with many MapReduce systems. As you may recall from our discussion of MapReduce in chapter 2, MapReduce is a framework for performing parallel processing on large datasets across multiple computers (nodes). In the MapReduce framework, the *map* operation has a master node which breaks up an operation into subparts and distributes each operation to another node for processing, and *reduce* is the process where the master node collects the results from the other nodes and combines them into the answer to the original problem.

Column family stores use row and column identifiers as general purposes keys for data lookup. They're sometimes referred to as *data stores* rather than *databases*, since they lack features you may expect to find in traditional databases. For example, they lack typed columns, secondary indexes, triggers, and query languages. Almost all column family stores have been heavily influenced by the original Google Bigtable paper. HBase, Hypertable, and Cassandra are good examples of systems that have Bigtable-like interfaces, although how they're implemented varies.

We should note that the term *column family* is distinct from a *column store*. A column-store database stores all information within a column of a table at the same location on disk in the same way a row-store keeps row data together. Column stores are used in many OLAP systems because their strength is rapid column aggregate calculation. *MonetDB*, *SybaseIQ*, and *Vertica* are examples of column-store systems. Column-store databases provide a SQL interface to access their data.

	A	B	C
1			
2			
3		Hello World!	
4			
5			
6			

Figure 4.18 Using a row and column to address a cell. The cell has an address of 3B and can be thought of as the lookup key in a sparse matrix system.

4.3.1 Column family basics

Our first example of using rows and columns as a key is the spreadsheet. Though most of us don't think of spreadsheets as a NoSQL technology, they serve as an ideal way to visualize how keys can be built up from more than one value. Figure 4.18 shows a spreadsheet with a single cell at row 3 and column 2 (the B column) that contains the text "Hello World!"

In a spreadsheet, you use the combination of a row number and a column letter as an address to "look up" the value of any cell. For example, the third column in the second row in the figure is identified by the key C2. In contrast to the key-value store, which has a single key that identifies the value, a spreadsheet has row and column identifiers that make up the key. But like the key-value store, you can put many different items in a cell. A cell can contain data, a formula, or even an image. The model for this is shown in figure 4.19.

This is roughly the same concept in column family systems. Each item of data can only be found by knowing information about the row and column identifiers. And, like a spreadsheet, you can insert data into any cell at any time. Unlike an RDBMS, you don't have to insert all the column's data for each row.

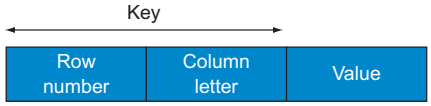

Figure 4.19 Spreadsheets use a row-column pair as a key to look up the value of a cell. This is similar to using a key-value system where the key has two parts. Like a key-value store, the value in a cell may take on many types such as strings, numbers, or formulas.

4.3.2 Understanding column family keys

Now that you're comfortable with slightly more complex keys, we'll add two additional fields to the keys from the spreadsheet example. In figure 4.20 you can see we've added a column family and timestamp to the key.

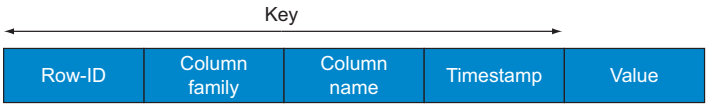

Figure 4.20 The key structure in column family stores is similar to a spreadsheet but has two additional attributes. In addition to the column name, a column family is used to group similar column names together. The addition of a timestamp in the key also allows each cell in the table to store multiple versions of a value over time.

The key in the figure is typical of column stores. Unlike the typical spreadsheet, which might have 100 rows and 100 columns, column family stores are designed to be...well...very big. How big? Systems with billions of rows and hundreds or thousands of columns are not unheard of. For example, a Geographic Information System (GIS) like Google Earth might have a row ID for the longitude portion of a map and use the column name for the latitude of the map. If you have one map for each square mile on Earth, you could have 15,000 distinct row IDs and 15,000 distinct column IDs.

What's unusual about these large implementations is that if you viewed them in a spreadsheet, you'd see that few cells contain data. This *sparse matrix* implementation is a grid of values where only a small percent of cells contain values. Unfortunately, relational databases aren't efficient at storing sparse data, but column stores are designed exactly for this purpose.

With a traditional relational database, you can use a simple SQL query to find all the columns in any table; when querying sparse matrix systems, you must look for every element in the database to get a full listing of all column names. One problem that may occur with many columns is that running reports that list columns and related columns can be tricky unless you use a column family (a high-level category of data also known as an *upper level ontology*). For example, you may have groups of columns that describe a website, a person, a geographical location, and products for sale. In order to view these columns together, you'd group them in the same column family to make retrieval easier.

Not all column family stores use a column family as part of their key. If they do, you'll need to take this into account when storing an item key, since the column family is part of the key, and retrieval of data can't occur without it. In as much as the API is simple, NoSQL products can scale to manage large volumes of data, adding new rows and columns without needing to modify a data definition language.

4.3.3 Benefits of column family systems

The column family approach of using a row ID and column name as a lookup key is a flexible way to store data, gives you benefits of higher scalability and availability, and saves you time and hassles when adding new data to your system. As you read through these benefits, think about the data your organization collects to see if a column family store would help you gain a competitive advantage in your market.

Since column family systems don't rely on joins, they tend to scale well on distributed systems. Although you can start your development on a single laptop, in production column family systems are usually configured to store data in three distinct nodes in possibly different geographic regions (geographically distinct data centers) to ensure high availability. Column family systems have automatic failover built in to detect failing nodes and algorithms to identify corrupt data. They leverage advanced hashing and indexing tools such as Bloom filters to perform probabilistic analysis on large data sets. The larger the dataset, the better these tools perform. Finally, column family implementations are designed to work with distributed filesystems (such as the

Hadoop distributed filesystem) and MapReduce transforms for getting data into or out of the systems. So be sure to consider these factors before you select a column family implementation.

HIGHER SCALABILITY

The world *Big* in the title of the original Google paper tells us that Bigtable-inspired column family systems are designed to scale beyond a single processor. At the core, column family systems are noted for their scalable nature, which means that as you add more data to your system, your investment will be in the new nodes added to the computing cluster. With careful design, you can achieve a linear relationship between the way data grows and the number of processors you require.

The principal reason for this relationship is the simple way that row IDs and column names are used to identify a cell. By keeping the interface simple, the back-end system can distribute queries over a large number of processing nodes without performing any join operations. With careful design of row IDs and columns, you give the system enough hints to tell it where to get related data and avoid unnecessary network traffic crucial to system performance.

HIGHER AVAILABILITY

By building a system that scales on distributed networks, you gain the ability to replicate data on multiple nodes in a network. Because column family systems use efficient communication, the cost of replication is lower. In addition, the lack of join operations allows you to store any portion of a column family matrix on remote computers. This means that if the server that holds part of the sparse matrix crashes, other computers are standing by to provide the data service for those cells.

EASY TO ADD NEW DATA

Like the key-value and graph stores, a key feature of the column family store is that you don't need to fully design your data model before you begin inserting data. But there are a couple constraints that you should know before you begin. Your groupings of column families should be known in advance, but row IDs and column names can be created at any time.

For all the good things that you can do with column family systems, be warned that they're designed to work on distributed clusters of computers and may not be appropriate for small datasets. You usually need at least five processors to justify a column family cluster, since many systems are designed to store data on three different nodes for replication. Column family systems also don't support standard SQL queries for real-time data access. They may have higher-level query languages, but these systems often are used to generate batch MapReduce jobs. For fast data access, you'll use a custom API written in a procedural language like Java or Python.

In the next three sections, we'll look at how column family implementations have been efficiently used by companies like Google to manage analytics, maps, and user preferences.

4.3.4 Case study: storing analytical information in Bigtable

In Google's Bigtable paper, they described how Bigtable is used to store website usage information in Google Analytics. The Google Analytics service allows you to track who's visiting your website. Every time a user clicks on a web page, the hit is stored in a single row-column entry that has the URL and a timestamp as the row ID. The row IDs are constructed so that all page hits for a specific user session are together.

As you can guess, viewing a detailed log of all the individual hits on your site would be a long process. Google Analytics makes it simple by summarizing the data at regular intervals (such as once a day) and creating reports that allow you to see the total number of visits and most popular pages that were requested on any given day.

Google Analytics is a good example of a large database that scales in a linear fashion as the number of users increases. As each transaction occurs, new hit data is immediately added to the tables even if a report is running. The data in Google Analytics, like other logging-type applications, is generally written once and never updated. This means that once the data is extracted and summarized, the original data is compressed and put into an intermediate store until archived.

This pattern of storing write-once data is the same pattern we discussed in the data warehouse and business intelligence section in chapter 3. In that section, we looked at sales fact tables and how business intelligence/data warehouse (BI/DW) problems can be cost-effectively solved by Bigtable implementations. Once the data from event logs is summarized, tools like pivot tables can use the aggregated data. The events can be web hits, sales transactions, or any type of event-monitoring system. The last step will be to use an external tool to generate the summary reports.

In the case of using HBase as a Bigtable store, you'll need to store the results in the Hadoop distributed filesystem (HDFS) and use a reporting tool such as Hadoop Hive to generate the summary reports. Hadoop Hive has a query language that looks similar to SQL in many ways, but it also requires you to write a MapReduce function to move data into and out of HBase.

4.3.5 Case study: Google Maps stores geographic information in Bigtable

Another example of using Bigtable to store large amounts of information is in the area of geographic information systems (GIS). GIS systems, like Google Maps, store geographic points on Earth, the moon, or other planets by identifying each location using its longitude and latitude coordinates. The system allows users to travel around the globe and zoom into and out of places using a 3D-like graphical interface.

When viewing the satellite maps, you can then choose to display the map layers or points of interest within a specific region of a map. For example, if you post vacation photos from your trip to the Grand Canyon on the web, you can identify each photo's location. Later, when your neighbor, who heard about your awesome vacation, is searching for images of the Grand Canyon, they'll see your photo as well as other photos with the same general location.

GIS systems store items once and then provide multiple access paths (queries) to let you view the data. They're designed to cluster similar row IDs together and result in rapid retrieval of all images/points that are near each other on the map.

4.3.6 Case study: using a column family to store user preferences

Many websites allow users to store preference information as part of their profile. This account-specific information can store privacy settings, contact information, and how they want to be notified about key events. Typically the size of this user preference page for a social networking site is under 100 fields or about 1 KB in size, which is reasonable as long as there's not a photo associated with it.

There are some things about user preference files that make them unique. They have minimal transactional requirements, and only the individual associated with the account makes changes. As a result, ensuring an ACID transaction isn't as important as making sure the transaction occurs when a user attempts to save or update their preference information.

Other factors to consider are the number of user preferences you have and system reliability. It's important that these read-mostly events are fast and scalable so that when a user logs in, you can access the preferences and customize their screen regardless of the number of concurrent system users.

Column family systems can provide the ideal match for storing user preferences when combined with an external reporting system. These reporting systems can be set up to provide high availability through redundancy, and yet still allow reporting to be done on the user preference data. In addition, as the number of users increases, the size of the database can expand by the addition of new nodes to your system without changing the architecture. If you have large datasets, big data stores may provide an ideal way to create reliable yet scalable data services.

Column family systems are known for their ability to scale to large datasets but they're not alone in this regard; document stores with their general and flexible nature are also a good pattern to consider when scalability is a requirement.

4.4 Document stores

Our coverage of NoSQL data patterns wouldn't be complete without talking about the most general, flexible, powerful, and popular area of the NoSQL movement: the document store. After reading this section, you'll have a clear idea of what document stores are and how they're used to solve typical business problems. We'll also look at some case studies where document stores have been successfully implemented.

As you may recall, key-value and Bigtable stores, when presented with a key, return the value (a BLOB of data) associated with that key. The key-value store and Bigtable values lack a formal structure and aren't indexed or searchable. Document stores work in the opposite manner: the key may be a simple ID which is never used or seen. But you can get almost any item out of a document store by querying any value or content within the document. For example, if you queried 500 documents associated with

the Civil War, you could search for those documents that referenced General Robert E. Lee. The query would return a list of documents that contained his name.

A consequence of using a document store is everything inside a document is automatically indexed when a new document is added. Though the indexes are large, everything is searchable. This means that if you know any property of a document, you can quickly find all documents with the same property. Document stores can tell not only that your search item is in the document, but also the search item's exact location by using the *document path*, a type of key, to access the leaf values of a tree structure, as illustrated in figure 4.21.

Even if a document structure is complex, a document store search API can remain simple and provides an easy way to select a document or subset of a document. A key-value store can store an entire document in the value portion of the key-value store, but a document store can quickly extract subsections of a large number of documents without loading each document into memory. If you want to display a single paragraph of a book, you don't need to load the entire book into RAM.

We'll begin our document store learning by visualizing something familiar: a tree with roots, branches, and leaves. We'll then look at how document and application collections use the document store concept and the document store API. Finally, we'll look at some case studies and popular software implementations that use document stores.

4.4.1 Document store basics

Think of a document store as a tree-like structure, as shown in figure 4.21.

Document trees have a single root element (or sometimes multiple root elements). Beneath the root element there is a sequence of branches, sub-branches, and values. Each branch has a related path expression that shows you how to navigate from the root of the tree to any given branch, sub-branch, or value. Each branch may have a value associated with that branch. Sometimes the existence of a branch in the tree has specific meaning, and sometimes a branch must have a given value to be interpreted correctly.

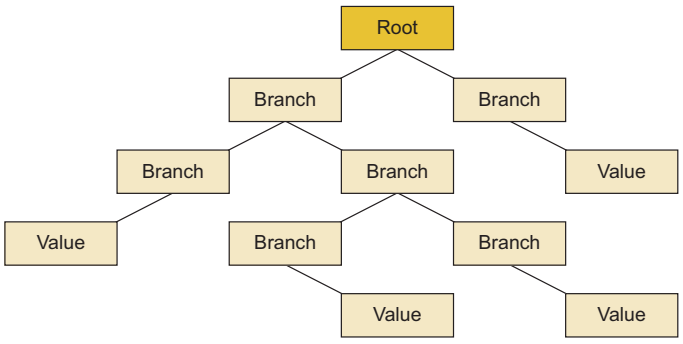

Figure 4.21 Document stores use a tree structure that begins with a root node, and have sub-branches that can also contain sub-branches. The actual data values are usually stored at the leaf levels of a tree.

4.4.2 Document collections

Most document stores group documents together in *collections*. These collections look like a directory structure you might find in a Windows or UNIX filesystem. Document collections can be used in many ways to manage large document stores. They can serve as ways to navigate document hierarchies, logically group similar documents, and store business rules such as permissions, indexes, and triggers. Collections can contain other collections and trees can contain subtrees.

If you're familiar with RDBMSs, you might think it natural to visualize document collections as an RDBMS. It might seem natural because you've used an XML column data type within your system. In this example, the RDBMS is a single table that contains XML documents. The problem with this view is that in the relational world, RDBMSs don't contain other tables, and you'd be missing the power and flexibility that comes with using a document store: allowing collections to have collections.

Document collections can also be used as application collections, which are containers for the data, scripts, views, and transforms of a software application. Let's see how an application collection (package) is used to load software applications into a native XML database like eXist.

4.4.3 Application collections

In some situations, the collection in a document store is used as a container for a web application package, as shown in figure 4.22.

This packaging format, called a *xar* file, is similar to a Java JAR file or a WAR file on Java web servers. Packaged applications can contain scripts as well as data. They're loaded into the document store and use packaging tools (scripts) to load the data if it's not already there. These packaging features make document stores more versatile, expanding their functionality to become application servers as well as document stores.

The use of collection structures to store application packages shows that a document store can be used as a container of high-level reusable components that can run on multiple NoSQL systems. If these developments continue, a market for reusable applications that are easy to install by nonprogrammers and can run on multiple NoSQL systems will soon be a reality.

Figure 4.22 Document store collections can contain many objects, including other collections and application packages. This is an example of a package repository that's used to load application packages into the eXist native XML database.

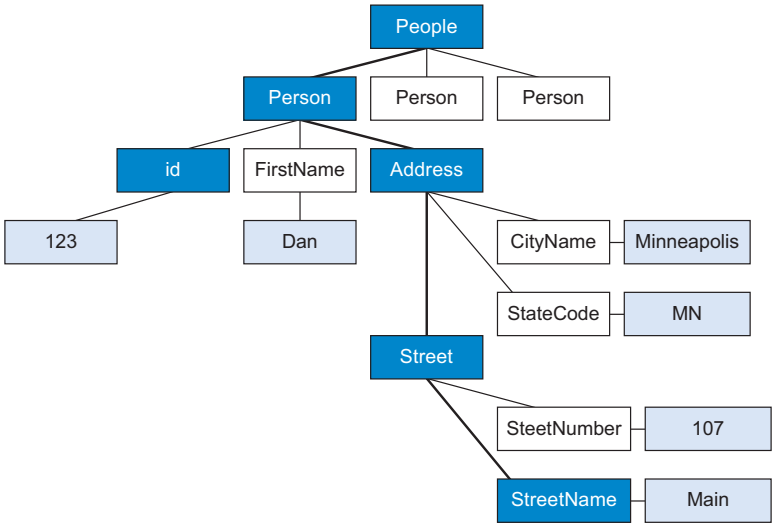

Figure 4.23 How a document path is used like a key to get the value out of a specific cell in a document. In this example, the path to the street name is `People/Person[id='123']/Address/Street/StreetName/text()`.

4.4.4 Document store APIs

Each document store has an API or query language that specifies the path or path expression to any node or group of nodes. Generally, nodes don't need to have distinct names; instead, a position number can be used to specify any given node in the tree. For example, to select the seventh person in a list of people, you might specify this query: `Person[7]`. Figure 4.23 is a more complex example of a complete path expression.

In figure 4.23 you begin by selecting a subset of all people records that have the identifier 123. Often this points to a single person. Next you look in the Address section of the record and select the text from the Address street name. The full path name to the street name is the following: `People/Person[id='123']/Address/Street/StreetName/text()`. If you think this seems complicated, know that path expressions are simple and easy to learn. When looking for something, you specify the correct child element down a path of the tree structure, or you can use a where clause, called a *predicate*, at any point in the path expression to narrow down the items selected.

We'll discuss more on using the World Wide Web standard for selecting a path using the XPath language in the next chapter.

4.4.5 Document store implementations

A document store can come in many varieties. Some are based on simple serialized object trees and some are more complex, containing content that might be found in web pages with text markup. Simpler document structures are often associated with serialized objects and may use the *JavaScript Object Notation (JSON)* format. JSON allows

arbitrary deep nesting of tree structures, but doesn't support the ability to store and query document attributes such as bold, italics, or hyperlinks within the text. We call this *complex content*. In our context we refer to JSON data stores as *serialized object stores* and to document stores that support complex content as *true document stores* with XML as the native format.

4.4.6 Case study: ad server with MongoDB

Do you ever wonder how those banner ads show up on the web pages you browse or how they really seem to target the things you like or are interested in? It's not a coincidence that they match your interests: they're tailored to you. It's done with ad serving. The original reason for MongoDB, a popular NoSQL product, was to create a service that would quickly send a banner ad to an area on a web page for millions of users at the same time.

The primary purpose behind ad serving is to quickly select the most appropriate ad for a particular user and place it on the page in the time it takes a web page to load. Ad servers should be highly available and run 24/7 with no downtime. They use complex business rules to find the most appropriate ad to send to a web page. Ads are selected from a database of ad promotions of paid advertisers that best match the person's interest. There are millions of potential ads that could be matched to any one user. Ad servers can't send the same ad repeatedly; they must be able to send ads of a specific type (page area, animation, and so on) in a specific order. Finally, ad systems need accurate reporting that shows what ads were sent to which user and which ads the user found interesting enough to click on.

The business problem 10gen (creators of MongoDB) was presented with was that no RDBMSs could support the complex real-time needs of the ad service market. MongoDB proved that it could meet all of the requirements at its inception. It has built-in autopartitioning, replication, load balancing, file storage, and data aggregation. It uses a document store structure that avoids the performance problems associated with most object-relational systems. In short, it was custom designed to meet the needs and demands of the ever-growing ad serving business and in the process turned out to be a good strategy for other problems that don't have real-time requirements, but want the ability to avoid the complex and slow object-relational mapping problems of traditional systems.

As well as being used as a basis for banner ad serving, MongoDB can be used in some of the following use cases:

- *Content management*—Store web content and photos and use tools such as geolocation indexes to find items.
- *Real-time operational intelligence*—Ad targeting, real-time sentiment analysis, customized customer-facing dashboards, and social media monitoring.
- *Product data management*—Store and query complex and highly variable product data.

- *User data management*—Store and query user-specific data on highly scalable web applications. Used by video games and social network applications.
- *High-volume data feeds*—Store large amounts of real-time data into a central database for analysis characterized by asynchronous writes to RAM.

4.4.7 Case study: CouchDB, a large-scale object database

In 2005, Damien Katz was looking for a better way to store a large number of complex objects using only commodity hardware. A veteran Lotus Notes user, he was familiar with its strengths and weaknesses, but he wanted to do things differently and created a system called *CouchDB* (cluster of unreliable commodity hardware), which was released as an open source document store with many of the same features of distributed computing as part of its core architecture.

CouchDB has document-oriented data synchronization built in at a low level. This allows multiple remote nodes to each have different versions of documents that are automatically synchronized if communication between the two nodes is interrupted. CouchDB uses MVCC to archive document-oriented ACID transactions, and also has support for document versioning. Written in *Erlang*, a functional programming language, CouchDB has the ability to rapidly and reliably send messages between nodes without a high overhead. This feature makes CouchDB remarkably reliable even when using a large number of processors over unreliable networks.

Like MongoDB, CouchDB stores documents in a JSON-style format and uses a JavaScript-like language to perform queries on the documents. Because of its powerful synchronization abilities, CouchDB is also used to synchronize mobile phone data.

Although CouchDB remains an active Apache project, many of the original developers of CouchDB, including Katz, are now working on another document store through the company *Couchbase*. Couchbase provides a distinct version of the product with an open source license.

The four main patterns—key-value store, graph store, Bigtable store, and document store—are the major architecture patterns associated with NoSQL. As with most things in life, there are always variations on a theme. Next, we'll take a look at a representative sample of the types of pattern variations and how they can be combined to build NoSQL solutions in organizations.

4.5 Variations of NoSQL architectural patterns

The key-value store, graph store, Bigtable store, and document store patterns can be modified by focusing on a different aspect of system implementation. We'll look at variations on the architectures that use RAM or solid state drives (SSDs), and then talk about how the patterns can be used on distributed systems or modified to create enhanced availability. Finally, we'll look at how database items can be grouped together in different ways to make navigation over many items easier.

4.5.1 Customization for RAM or SSD stores

In chapter 2 we reviewed how access times change based on the type of memory media used. Some NoSQL products are designed to specifically work with one type of memory; for example, Memcache, a key-value store, was specifically designed to see if items are in RAM on multiple servers. A key-value store that only uses RAM is called a RAM cache; it's flexible and has general tools that application developers can use to store global variables, configuration files, or intermediate results of document transformations. A RAM cache is fast and reliable, and can be thought of as another programming construct like an array, a map, or a lookup system. There are several things about them you should consider:

- Simple RAM resident key-value stores are generally empty when the server starts up and can only be populated with values on demand.
- You need to define the rules about how memory is partitioned between the RAM cache and the rest of your application.
- RAM resident information must be saved to another storage system if you want it to persist between server restarts.

The key is to understand that RAM caches must be re-created from scratch each time a server restarts. A RAM cache that has no data in it is called a *cold cache* and is why some systems get faster the more they're used after a reboot.

SSD systems provide permanent storage and are almost as fast as RAM for read operations. The Amazon DynamoDB key-value store service uses SSDs for all its storage, resulting in high-performance read operations. Write operations to SSDs can often be buffered in large RAM caches, resulting in fast write times until the RAM becomes full.

As you'll see, using RAM and SSDs efficiently is critical when using distributed systems that provide for higher volume and availability.

4.5.2 Distributed stores

Now let's see how NoSQL data architecture patterns vary as you move from a single processor to multiple processors that are distributed over data centers in different geographic regions. The ability to elegantly and transparently scale to a large number of processors is a core property of most NoSQL systems. Ideally, the process of data distribution is transparent to the user, meaning that the API doesn't require you to know how or where your data is stored. But knowing that your NoSQL software can scale and how it does this is critical in the software selection process.

If your application uses many web servers, each caching the result of a long-running query, it's most efficient to have a method that allows the servers to work together to avoid duplication. This mechanism, known as *memcache*, was introduced in the LiveJournal case study in chapter 1. Whether you're using NoSQL or traditional SQL systems, RAM continues to be the most expensive and precious resource in an

application server's configuration. If you don't have enough RAM, your application won't scale.

The solution used in a distributed key-value store is to create a simple, lightweight protocol that checks whether any other server has an item in its cache. If it does, this item is quickly returned to the requester and no additional searching is required. The protocol is simple: each memcache server has a list of the other memcache servers it's working with. Whenever a memcache server receives a request that's not in its own cache, it checks with the other peer servers by sending them the key.

The memcache protocol shows that you can create simple communication protocols between distributed systems to make them work efficiently as a group. This type of information sharing can be extended to other NoSQL data architectures such as Bigtable stores and document stores. You can generalize the key-value pair to other patterns by referring to them as *cached items*.

Cached items can also be used to enhance the overall reliability of a data service by replicating the same items in multiple caches. If one server goes down, other servers quickly fill in so that the application gives the user the feeling of service without interruption.

To provide a seamless data service without interruption, the cached items need to be replicated automatically on multiple servers. If the cached items are stored on two servers and the first one becomes unavailable, the second server can quickly return the value; there's no need to wait for the first server to be rebooted or restored from backup.

In practice, almost all distributed NoSQL systems can be configured to store cached items on two or three different servers. The decision of which server stores which key can be determined by implementing a simple round-robin or random distribution system. There are many trade-offs relating to loads distributed over large clusters of key-value store systems and how the cached items in unavailable systems can be quickly replicated onto new nodes.

NoSQL systems dominate organizations that have large collections of data items, and it becomes cumbersome to deal with these items if they can only be accessed in a single linear listing. You can group items together in different ways to make them easier to manage and navigate, as you'll next see.

4.5.3 Grouping items

In the key-value store section, we looked at how web pages can be stored in a key-value store using a website URL as the key and the web page as the value. You can extend this construct to filesystems as well. In a filesystem, the key is the directory or folder path, and the value is the file content. But unlike web pages, filesystems have the ability to list all the files in a directory without having to open the files. If the file content is large, it would be inefficient to load all of the files into memory each time you want a listing of the files.

To make this easier and more efficient, a key-value store can be modified to include additional information in the structure of the key to indicate that the key-value pair is associated with another key-value pair, creating a *collection*, or general-purpose structures used to group resources. Though each key-value store system might call it something different (such as folders, directories, or buckets), the concept is the same.

The implementation of a collection system can also vary dramatically based on what NoSQL data pattern you use. Key-value stores have several methods to group similar items based on attributes in their keys. Graph stores associate one or more group identifiers with each triple. Big data systems use column families to group similar columns. Document stores use a concept of a document collection. Let's take a look at some examples used by key-value stores.

One approach to grouping items is to have two key-value data types, the first called *resource keys* and the second *collection keys*. You can use collection keys to store a list of keys that are in a collection. This structure allows you to store a resource in multiple collections and also to store collections within collections. Using this design poses some complex issues that require careful thought and planning about what should be done with a resource if it's in more than one collection and one of the collections is deleted. Should all resources in a collection be automatically deleted?

To simplify this process and subsequent design decisions, key-value systems can include the concept of creating collection hierarchies and require that a resource be in one and only one collection. The result is that the path to a resource is essentially a distinct key for retrieval. Also known as a *simple document hierarchy*, the familiar concept of folders and documents resonates well with end users.

Once you've established the concept of a collection hierarchy in a key, you can use it to perform many functions on groups of key-value pairs; for example:

- Associate metadata with a collection (who created the collection, when it was created, the last time it was modified, and who last modified the collection).
- Give the collection an owner and group, and associate access rights with the owner group and other users in the same way UNIX filesystems use permissions.
- Create an access control permission structure on a collection, allowing only users with specific privileges the ability to read or modify the items within the collection.
- Create tools to upload and/or download a group of items into a collection.
- Set up systems that compress and archive collections if they haven't been accessed for a specific period of time.

If you're thinking, "That sounds a lot like a filesystem," you're right. The concept of associating metadata with collections is universal, and many file and document management systems use concepts similar to key-value stores as part of their core infrastructure.

4.6 Summary

In this chapter, we reviewed some of the basic NoSQL data architectural patterns and data structures. We looked at how simple data structures like key-value stores can lead to a simple API, which make the structures easy to implement and highly portable across systems. We progressed from a simple interface such as key-value stores to the more complex document store, where each branch or leaf in a document can be selected to create query results.

By looking at the fundamental data structures being used by each data architecture pattern, you can understand the strengths and weaknesses of each pattern for various business problems. These patterns are useful for classifying many commercial and open source products and understanding their core strengths and weaknesses. The challenge is that many real-world systems rarely fit into a single category. They may start with a single pattern, but then so many features and plug-ins are added, they become difficult to put neatly into a single classification system. Many products that began as simple key-value stores have many features common to Bigtable stores. So it's best to treat these patterns as guidelines rather than rigid classification rules.

In our next chapter, we'll take an in-depth look at the richest data architecture pattern: the native XML database with complex content. We'll see some case studies where native XML database systems are used for enterprise integration and content publishing.

4.7 Further reading

- Berners-Lee, Tim. "What the Semantic Web can represent." September 1998. http://mng.bz/L9a2.
- "Cypher query language." From *The Neo4j Manual*. http://mng.bz/hC3g.
- DeCandia, Giuseppe, et al. "Dynamo: Amazon's Highly Available Key-value Store." Amazon.com. 2007. http://mng.bz/YY5A.
- MongoDB. Glossary: collection. http://mng.bz/Jl5M.
- Stardog. An RDF triple store that uses SPARQL. http://stardog.com
- "The Linking Open Data cloud diagram." September 2011. http://richard.cyganiak.de/2007/10/lod/.
- W3C. "Packaging System: EXPath Candidate Module 9." May 2012. http://expath.org/spec/pkg.