SAMPLE CHAPTER

Spark GraphX IN ACTION

Michael S. Malak
Robin East

MANNING

Spark GraphX in Action

by Michael S. Malak

Robin East

Chapter 1

brief contents

Part 1

Spark and graphs

Graphs—the things composed of vertices and edges, not graphs from Algebra class—carry a mystique about them. They seem to be very powerful, yet what can be done with them is a bit of a mystery. Part of the problem is that the answer "graphs can do anything" says precisely nothing. Right off in chapter 1, we suggest a broad categorization of different types of graphs found in the world. In the last third of chapter 3 we illustrate graph terminology.

Apache Spark is a distributed computing system growing in popularity due to its speed. GraphX is Spark applied to graphs, and chapter 1 describes how GraphX fits into a data processing workflow. In chapter 2, you'll actually get hands on with PageRank, the algorithm that launched Google.

Chapter 3 is a crash course in the three foundational technologies required for this book: Spark, Scala, and graphs.

1 Two important technologies: Spark and graphs

This chapter covers

- Why Spark has become the leading Big Data processing system
- What makes graphs a unique way of modeling connected data
- How GraphX makes Spark a leading platform for graph analytics

It's well-known that we are generating more data than ever. But it's not just the individual data points that are important—it's also the connections between them. Extracting information from such connected datasets can give insights into numerous areas such as detecting fraud, collecting bioinformatics, and ranking pages on the web.

Graphs provide a powerful way to represent and exploit these connections. Graphs represent networks of data points as vertices and encode connections through edges between pairs of vertices. Graphs can be used to model such diverse areas as computer vision, natural language processing, and recommender systems.

With such a representation of connected data comes a whole raft of tools and techniques that can be used to mine the information content of the network. Among the many tools covered in this book, you'll find PageRank (for finding the most influential members of the network), topic modeling with Latent Dirichlet Allocation (LDA), and clustering coefficient to discover highly connected communities.

Unfortunately, traditional tools used for the analysis of data, such as relational databases, are not well suited to this type of problem. Table-oriented frameworks such as SQL are cumbersome when it comes to representing typical graph notions such as following a trail of connections. Furthermore, traditional methods of data processing fail to scale as the size of the data to be analyzed increases.

A solution is at hand with graph processing systems. Such systems supply data models and programming interfaces that provide a more natural way to query and analyze graph structures. Graph processing systems provide the means to create graph structures from raw data sources and apply the processing necessary to mine the information content therein.

Apache Spark is the Big Data processing alternative that has all but supplanted Hadoop, the open source data processing platform that ushered in the era of Big Data. Easily scaling to clusters of hundreds of nodes, Spark's in-memory data processing can often outperform Hadoop many times over.

GraphX is the graph processing layer on top of Spark that brings the power of Big Data processing to graphs—graphs that would be too large to fit on a single machine. People started using Spark for graphs long ago, including with the predecessor Bagel module, but with GraphX we now have a standardized way to do so, and it also provides a library of useful algorithms.

Here are some of the many reasons why you may want to use Spark GraphX:

- You already have Spark data processing pipelines and want to incorporate graph processing.
- You're curious about the power of Spark and/or GraphX.
- You're among the many for whom graph data has become important.
- Your graph data is too large to fit on a single machine.
- Either you don't need multiple applications accessing the same data store or you plan to add a REST server to Spark; for example, with the add-on originally by Ooyala called Spark Job Server.
- Either you don't need database-type transactions or you plan on using a graph database such as Neo4j or Titan in conjunction with GraphX.
- You already have a Spark cluster available to your application.
- You would like to use the concise, expressive power of Scala.

1.1 Spark: the step beyond Hadoop MapReduce

This section discusses Big Data in relation to Spark and graphs. Big Data is a major challenge for data science teams, in part because a single machine is unlikely to have

the power and capacity to run processing at the scale required. Moreover, even systems designed for Big Data, such as Hadoop, can struggle to process graph data efficiently due to some of the properties of that data, as you'll see later in this chapter.

Apache Spark is similar to Apache Hadoop in that it stores data distributed across a cluster of servers, or *nodes*. The difference is that Apache Spark stores data in memory (RAM) whereas Hadoop stores data on disk (either a spinning hard disk drive or a solid-state drive (SSD)), as shown in figure 1.1.

DEFINITION The word *node* has two distinct uses when it comes to graphs and to cluster computing. Graph data is composed of vertices and edges, and in that context *node* is a synonym for vertex. In cluster computing, the physical machines that comprise the cluster are also known as nodes. To avoid confusion, we refer to graph nodes/vertices only as *vertices*, which is also the terminology adopted by Spark GraphX. When we use the word *node* in this book, we mean strictly one physical computer participating in cluster computing.

Besides differing in where data is processed during computation (RAM versus disk), Spark's API is much easier to work with than the Hadoop Map/Reduce API. Combined

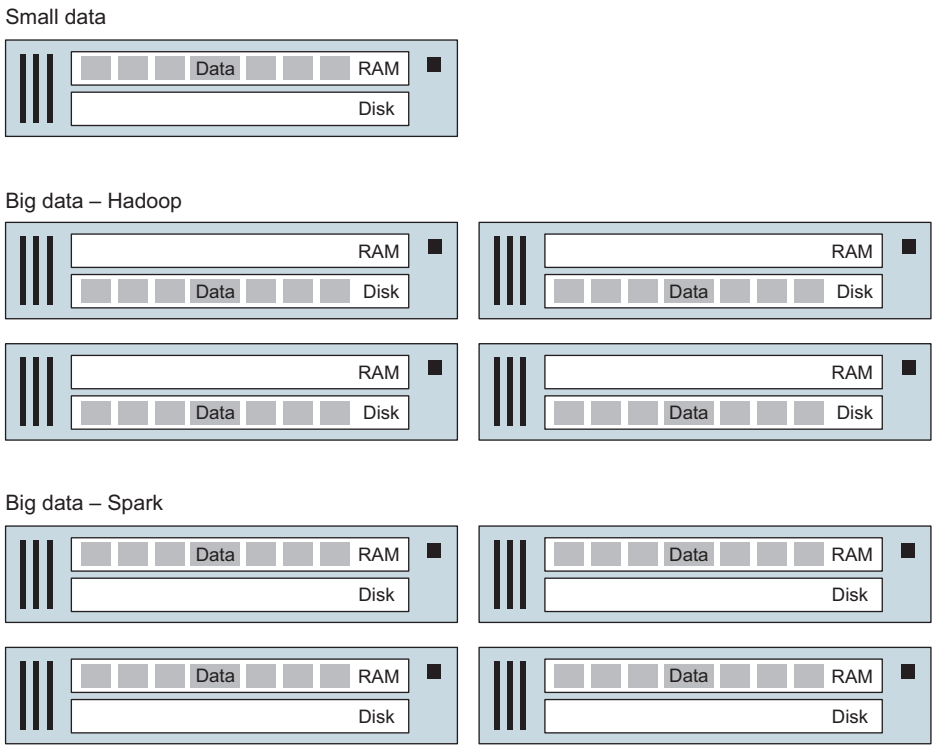


Figure 1.1 Big Data is data that is too big to fit on a single machine. Hadoop and Spark are technologies that distribute Big Data across a cluster of nodes. Spark is faster than Hadoop alone because it distributes data across the RAM in the cluster instead of the disks.

with the conciseness of Scala, the native programming language of Spark, a ratio of 100:1 for the number of Hadoop Map/Reduce Java lines of code to Spark Scala lines of code is common.

Although this book uses Scala primarily, don't worry if you don't know Scala yet. Chapter 3 provides a jumpstart into Scala, and all along the way we explain the tricks and terse, arcane syntax that are part and parcel of Scala. But deep familiarity with at least one programming language—such as Java, C++, C#, or Python—is assumed.

1.1.1 The elusive definition of Big Data

The idea of Big Data has gotten a lot of hype. The ideas trace back to the 2003 Google Paper on the Google File System and the 2004 Google paper on Map/Reduce, and these inspired the development of what is now Apache Hadoop.

The term *Big Data* has a lot of competing definitions, and some claim it has by now lost all meaning, but there is a simple core and crucial concept it still legitimately embodies: data that's too large to fit on a single machine.

Data sizes have exploded. Data is coming from website click streams, server logs, and sensors, to name a few sources. Some of this data is graph data, meaning it's comprised of edges and vertices, such as from collaborative websites (aka *Web 2.0* of which *social media* is a subset). Large sets of graph data are effectively crowdsourced, such as the body of interconnected knowledge contained in Wikipedia or the graph represented by Facebook friends, LinkedIn connections, or Twitter followers.

1.1.2 Hadoop: the world before Spark

Before we talk about Spark, let's recap how Hadoop solves the Big Data processing problem, because Spark builds on the core Hadoop concepts described in this section.

Hadoop provides a framework to implement fault-tolerant parallel processing on a cluster of machines. Hadoop provides two key capabilities:

- *HDFS*—Distributed storage
- *MapReduce*—Distributed compute

HDFS provides distributed, fault-tolerant storage. The NameNode partitions a single large file into smaller blocks. A typical block size is 64 MB or 128 MB. The blocks are scattered across the machines in the cluster. Fault-tolerance is provided by replicating each block of the file to a number of nodes (the default is three, but to make the diagram simpler, figure 1.2 shows a replication factor of two). Should a node fail, rendering all the file blocks on that machine unavailable, other nodes can transparently provide the missing blocks. This is a key idea in the architecture of Hadoop: the design accommodates machine failures as part of normal operations.

MapReduce (see figure 1.3) is the Hadoop parallel processing framework that provides parallel and distributed computation. MapReduce allows the programmer to write a single piece of code, encapsulated in *map* and *reduce* functions that are executed against the dataset residing on HDFS. To achieve *data locality*, the code is shipped (in .jar form) to the data nodes, and the Map is executed there. This avoids

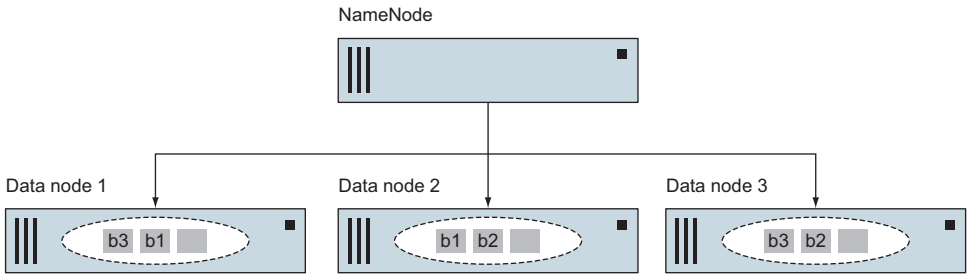


Figure 1.2 Three data blocks distributed with replication factor 2 across a Hadoop Distributed File System (HDFS)

consuming network bandwidth to ship the data around the cluster. For the Reduce summary, though, the results of the Maps are shipped to some Reduce node for the Reduce to take place there (this is called *shuffling*). Parallelism is achieved primarily during the Map, and Hadoop also provides resiliency in that if a machine or process fails, the computation can be restarted on another machine.

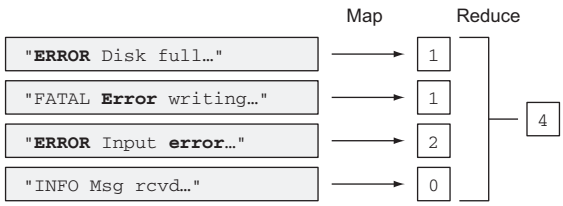


Figure 1.3 MapReduce is the processing paradigm used by both Hadoop and Spark. Shown is a MapReduce operation to count the number of times "error" appears in a server log. The Map is (normally) a one-to-one operation that produces one transformed data item for each source data item. The Reduce is a many-to-one operation that summarizes the Map outputs. Both Hadoop and Spark use the MapReduce paradigm.

The MapReduce programming framework abstracts the dataset as a stream of key-value pairs to be processed and the output written back to HDFS. It's a limited paradigm but it has been used to solve many data parallel problems by chaining together MapReduce read-process-write operations. Simple tasks, such as the word counting in figure 1.3, benefit from this approach. But iterative algorithms like machine learning suffer, which is where Spark comes in.

1.1.3 Spark: in-memory MapReduce processing

This section looks at an alternative distributed processing system, Spark, which builds on the foundations laid by Hadoop. In this section you'll learn about Resilient Distributed Datasets (RDDs), which have a large role to play in how Spark represents graph data.

Hadoop falls down on a couple of classes of problems:

- Interactive querying
- Iterative algorithms

Hadoop is good for running a single query on a large dataset, but in many cases, once we have an answer, we want to ask another question of the data. This is referred to as *interactive querying*. With Hadoop, this means waiting to reload the data from disk and process it again. It's not unusual to have to execute the same set of computations as a precursor to subsequent analysis.

Iterative algorithms are used in a wide array of machine learning tasks, such as Stochastic Gradient Descent, as well as graph-based algorithms like PageRank. An iterative algorithm applies a set of calculations to a dataset over and over until some criterion has been met. Implementing such algorithms in Hadoop typically requires a series of MapReduce jobs where data is loaded on each iteration. For large datasets, there could be hundreds or thousands of iterations, resulting in long runtimes.

Next you'll see how Spark solves these problems. Like Hadoop, Spark runs on a cluster of commodity hardware machines. The key abstraction in Spark is a Resilient Distributed Dataset (RDD). RDDs are created by the Spark application (residing in the Spark Driver) via a Cluster Manager, as shown in figure 1.4.

An RDD consists of distributed subsets of the data called *partitions* that can be loaded into memory on the machines across the cluster.

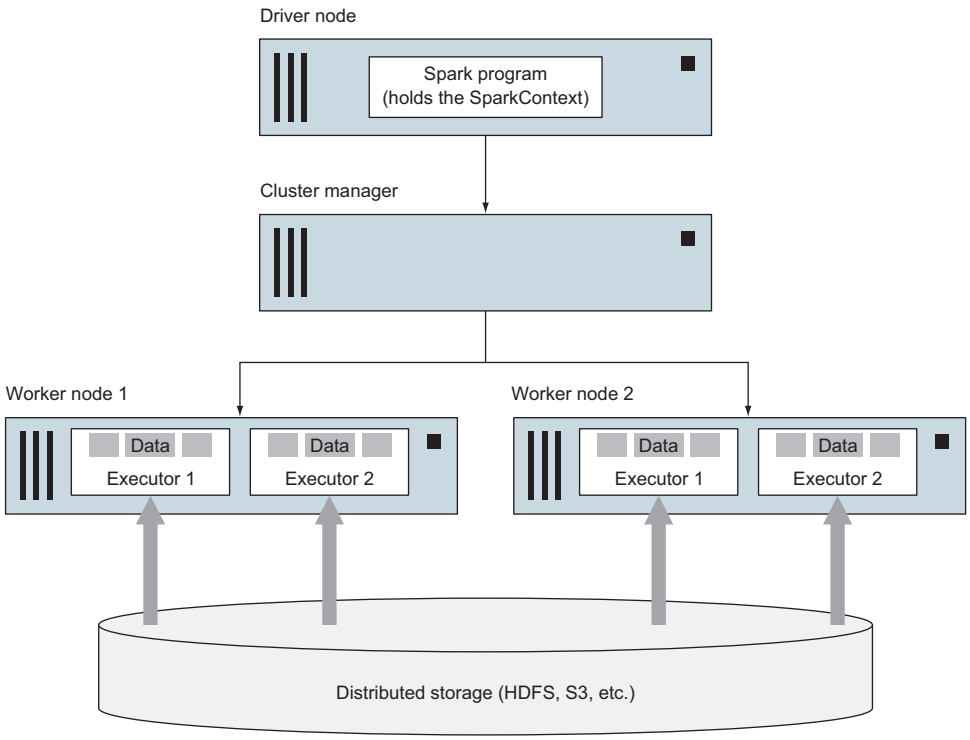


Figure 1.4 Spark provides RDDs that can be viewed as distributed in-memory arrays.

IN-MEMORY PROCESSING

Spark performs most of its operations in RAM. Because Spark is memory-based, it's more suited to processing graphs than Hadoop Map/Reduce because Map/Reduce processes data sequentially, whereas RAM is by nature random-access.

The key to Spark's usefulness in interactive querying and iterative processing is its ability to cache RDDs in memory. Caching an RDD avoids the need to reprocess the chain of parent RDDs each time a result is returned.

Naturally, this means that to take advantage of Spark's in-memory processing, the machines in the cluster must have a large amount of RAM. But if the available memory is insufficient, Spark will spill data back to disk gracefully and continue to work.

A Spark cluster needs a place to store data permanently. That place needs to be a distributed storage system, and options include HDFS, Cassandra, and Amazon's S3.

1.2 Graphs: finding meaning from relationships

Graphs can be used to represent naturally occurring connected data, such as the following:

- Social networks
- Mobile phone systems
- Web pages on the internet

Limited for decades to the realm of academia and research, graphs have over the past few years been adopted by organizations from Silicon Valley social media companies to governmental intelligence agencies seeking to find and use relationship patterns in their data. Graphs have now even entered the popular lexicon, with Facebook introducing its Graph Search, intelligence agencies publicly calling for the need to "connect the dots," and the old internet meme/game called the Six Degrees of Kevin Bacon. Even the now-universal and ubiquitous icon for *share* on social media and smartphone cameras is that of a miniature graph:

One of the most common uses for graphs today is to mine social media data, specifically to identify cliques, to recommend new connections, and to suggest products and ads. Such data can be big—more than can be stored on a single machine—which is where Spark comes in: it stores data across multiple machines participating in a cluster.

Spark is well-suited to handling graph data for another reason: it stores data in the memory (RAM) of each computer in the cluster, in contrast to Hadoop, which stores data on the disk of each computer in the cluster. Whereas Hadoop can handle sequential access of data, Spark can handle the arbitrary access order needed by a graph system, which has to traverse graphs from one vertex to the next.

GraphX is not a database. Instead, it's a graph processing system, which is useful, for example, for fielding web service queries or performing one-off, long-running standalone computations. Because GraphX isn't a database, it doesn't handle updates and deletes like Neo4j and Titan, which *are* graph databases. Apache Giraph is

another example of a graph processing system, but Giraph is limited to slow Hadoop Map/Reduce. GraphX, Giraph, and GraphLab are all separate implementations of the ideas expressed in the Google Pregel paper. Such graph processing systems are optimized for running algorithms on the entire graph in a massively parallel manner, as opposed to working with small pieces of graphs like graph databases. To draw a comparison to the world of standard relational databases, graph databases like Neo4j are like OLTP (Online Transaction Processing) whereas graph processing systems like GraphX are like OLAP (Online Analytical Processing).

Graphs can store various kinds of data: geospatial, social media, paper citation networks, and, of course, web page links. A tiny social media network graph is shown in figure 1.5. "Ann," "Bill," "Charles," "Diane," and "'Went to gym this morning'" are *vertices*, and "Is-friends-with," "Wrote-status," and "Likes-status" are *edges*.

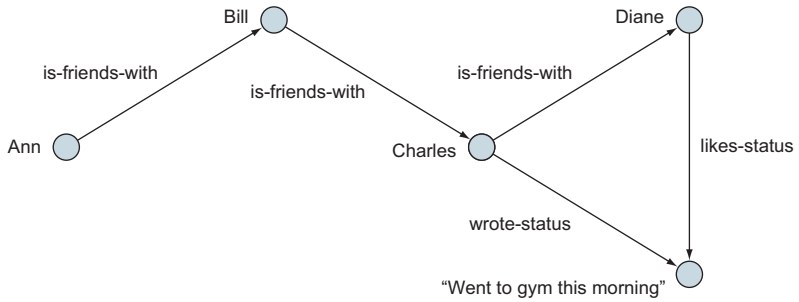


Figure 1.5 If Charles shares his status with friends of friends, determining the list of who could see his status would be cumbersome to figure out if you only had tables or arrays to work with.

1.2.1 Uses of graphs

It's well-known that we are now living in a world where we are generating more data than ever before. We are collecting more data points with richer content from an ever-expanding variety of sources.

To take advantage of this situation, organizations big and small are also putting data analysis and data mining at the heart of their operations—a move that some have dubbed the *data-driven business*. But data is not just getting bigger, it's more connected. This connectedness is what gives data its richness and provides ever greater opportunities to understand the world around us. Graphs offer a powerful way to represent and exploit these connections.

What forms does this connected data take? Start with one of the most well-known connected datasets: the World Wide Web. At a simplistic level, the web consists of billions of pages of metadata, text, images, and videos, and every page can point to one or more of the other pages using a link tag.

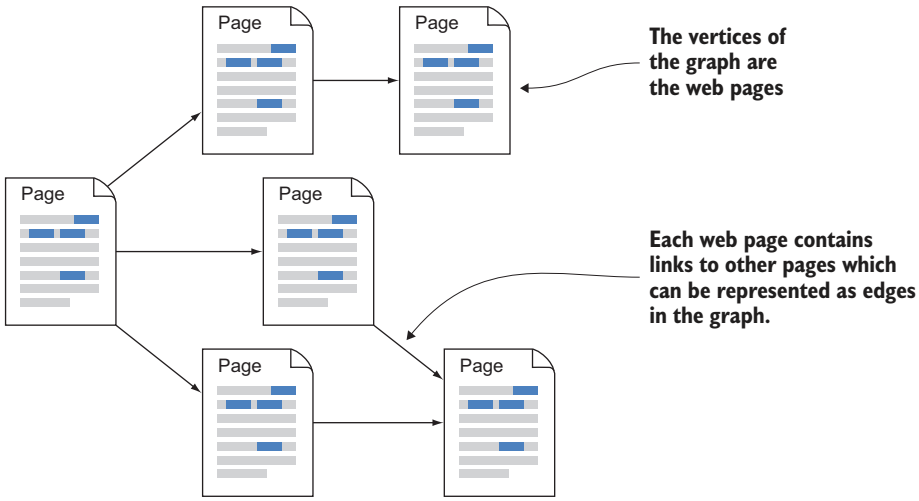


Figure 1.6 The links between web pages can be represented as a graph. The structure of the graph provides information about the relative authority, or ranking, of each page.

As figure 1.6 shows, you can represent these pages and links as a graph. You can then use the structure of the graph to provide information on the relative authority of each page. You can visualize this as each page providing a vote for each page it points to. But not all pages are equal; you might imagine that a page on a major news site has more importance than a posting by an unknown blogger. This is the problem that's solved by the PageRank algorithm, as you will see in chapter 5, and it has many more applications beyond ranking web pages.

The graphs we have looked at so far have captured links between pages; there is either a link or no link. We can make the graphs richer if we have more information about the connection. A typical example would be ratings information. When you give a 5-star rating to a movie on Netflix, not only do you create a connection between yourself and the movie, you also assign a value to that connection.

Movie ratings aren't the only value that can be applied to connections in graphs. Dollar values in the analysis of financial fraud, distances travelled between cities, and the traffic carried across a network of mobile phone stations are other examples of ways to enhance the richness of the connections represented in graphs.

Even if the connections between data points don't have a measurable value, there is still valuable information that can be captured in the graph. Take a social media site as an example. Each profile could store details of where a person went to school, and as before, this represents a connection between the person and the school. If we capture other information, such when they attended the school, that additional information can be represented in the graph. Now when we want to show friend recommendations to our user, we can make sure we don't show them the class of '96 when they are in the class of '83.

Graphs existed long before social networking. Other uses for graphs include

- Finding the shortest route in a geo-mapping app
- Recommending products, services, personal contacts, or media based on other people with similar-looking graphs
- Converting a tangle of interconnected topics into a hierarchy for organizational schemes that require a hierarchy (computer file system folders, a class syllabus, and so forth)
- Determining the most authoritative scholarly papers

1.2.2 Types of graph data

What kind of data can you put into a graph? The usual answer "anything" is not very helpful. Figure 1.7 shows some different types of data that can be represented by a graph:

- Network
- Tree
- RDBMS-like data
- Sparse matrix
- Kitchen sink

A *network* graph can be a road network as shown in figure 1.7, a social network, or a computer network. A tree graph has no cycles (loops). Any RDBMS can be converted into a graph format; an employee RDBMS is shown converted into a graph. But this would only be useful if some graph algorithms are needed, such as PageRank for community detection or minimum spanning tree for network planning.

As discussed in chapter 3, every graph has an associated *adjacency matrix*. This powerful concept has an important implication: that a graph is just an alternative data structure and not something magical. Some algorithms, which might otherwise have to deal with unwieldy matrices, can take advantage of the more compressed representation of a graph, especially if the alternative is a *sparse matrix*. SVD++, discussed in chapter 7, is an example of such an algorithm.

Attempts have been made to create *kitchen sink* graphs to encode all of human knowledge. The Cyc project is an example that attempts to encode all of human common sense into a graph. The YAGO (Yet Another Great Ontology) project has the slightly more modest goal of encoding an ontology (dictionary, hierarchy, and relationships) that represents everything in the world. Sometimes people think artificial intelligence will automatically result from such an ambitious graph. That doesn't happen, but such graphs are useful for assististing natural language processing projects of reasonable goals.

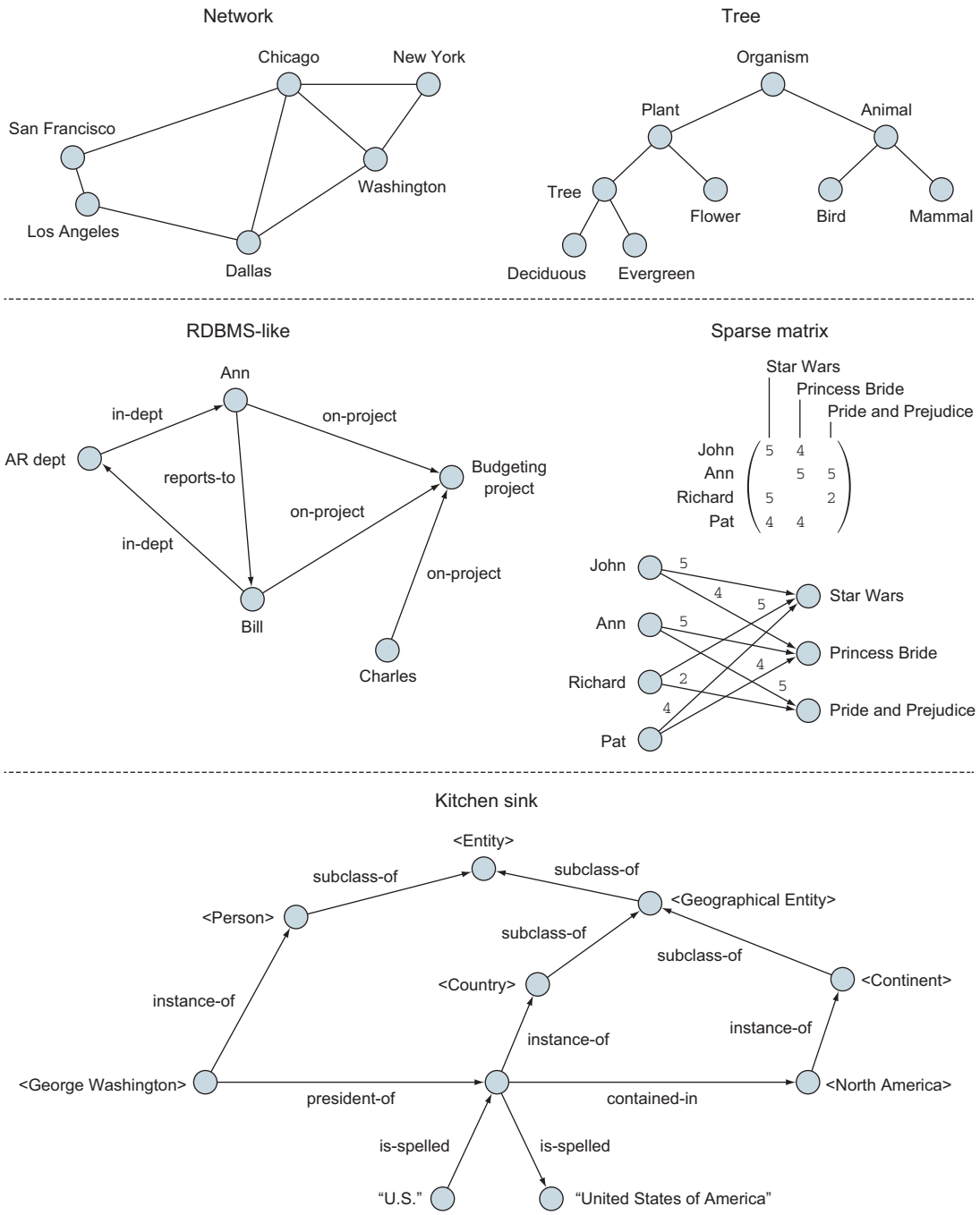


Figure 1.7 Different types of data that can be represented by graphs

1.2.3 Plain RDBMS inadequate for graphs

If you were to try to represent a graph in an RDBMS—or arrays of objects, if you're not familiar with SQL—you would probably have one table (or array) of vertices and another table of edges. The table of edges would have foreign keys (references) to the vertices table so that each edge would refer to the two vertices in connects. This is all well and good, provided you don't need to query deeply in the graph.

In the example graph in figure 1.5, suppose we want to find out who can see Charles's status "Went to gym this morning." If Charles shared it only with direct friends, then finding who can see it—who Charles' direct friends are—would be easy to do with a table structure. But suppose Charles shared his status with friends of friends; then to reach Ann would require hopping through the tables. In terms of SQL, we would have to join the edge table to itself. If we wanted the Six Degrees of Kevin Bacon, we would have to join the edge table to itself six times within the same SQL query.

What is common to problems that can be modeled as graphs is that we are focusing as much on the connections between entities as on the entities themselves. In many cases we want to traverse the connections to find things such as friends-of-friends-of-friends in social networks, cascades of retweets on Twitter, or the common component in a network of failed computers.

Furthermore, not all connections are created equal. Suppose we are analyzing surveillance data on a known criminal and his many associates and connections. We want to identify those people most likely to provide us with information, but it doesn't make sense to investigate everybody who has some connection; we want to prioritize by some sort of metric that measures the strength of the connection. One such metric could be the number of times a week that contact is made. Graphs allow us to assign a value or weight to each connection and then use that weighting in subsequent processing.

1.3 Putting them together for lightning fast graph processing: Spark GraphX

GraphX is a layer on top of Spark that provides a graph data structure composed of Spark RDDs, and it provides an API to operate on those graph data structures. GraphX comes with the standard Spark distribution, and you use it through a combination of the GraphX-specific API and the regular Spark API.

Spark originated out of AMPLab at the University of California, Berkeley in 2011 and became a top-level Apache project in 2014. Not everything from AMPLab is part of the official Apache Spark distribution. And to operate, Spark requires two major pieces shown in the bottom two gray layers of figure 1.8: distributed storage and a cluster manager. In this book, we assume HDFS for the distributed storage and not having a cluster manager, which is running Spark on a single computer; this is sometimes called *pseudo-distributed* mode for test and development.

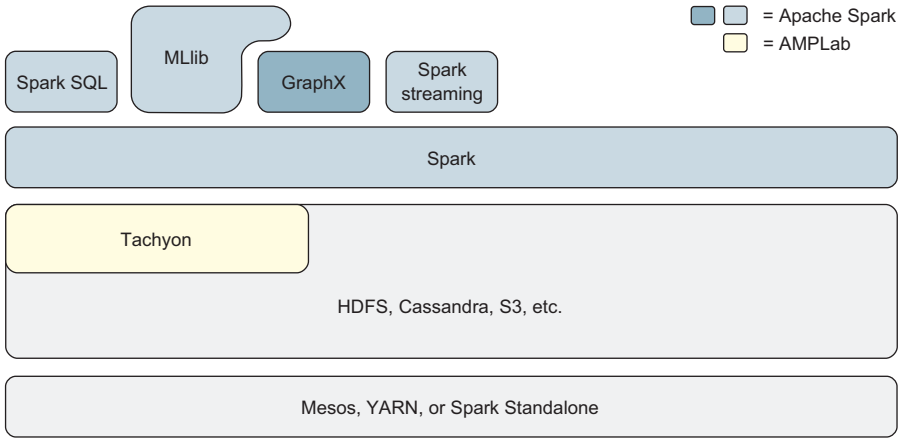


Figure 1.8 The Spark stack. Some components, including GraphX, come with Spark. Others, such as HDFS and YARN, are part of Apache Hadoop. In the last category, Tachyon comes from AMPLab at the University of California, Berkeley. Most of MLlib stands alone on top of Spark Core, but a couple of its algorithms make use of GraphX under the covers.

> **NOTE** Because GraphX is fully part of the base Spark package from Apache, version numbers for Spark Core and its base components, including GraphX, are synchronized.

1.3.1 Property graph: adding richness

We've seen that graphs in the real world contain valuable information beyond simply the connection between a vertex and an edge. Graphs are rich with data, and we need a way to represent this richness.

GraphX implements a notion called the *property graph*. As shown in figure 1.9, both vertices and edges can have arbitrary sets of attributes associated with them. The attribute could be something as simple as the age of a person or something as complex as an XML document, image, or video.

GraphX represents a graph using 2 RDDs, vertices and edges. Representing graphs in this way allows GraphX to deal with one of the major issues in processing large graphs: partitioning.

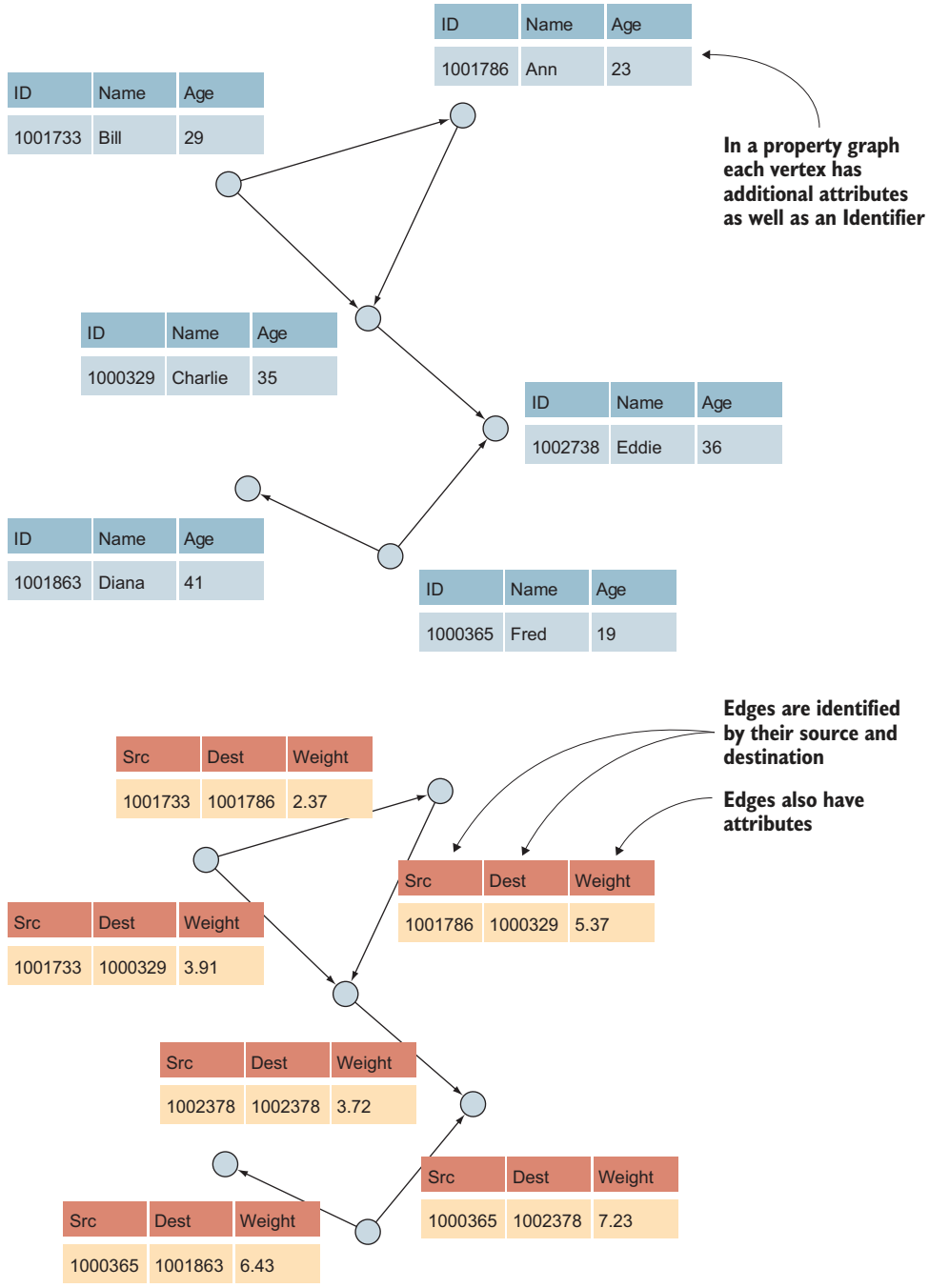


Figure 1.9 Both vertices and edges in a property graph contain additional attributes.

1.3.2 Graph partitioning: graphs meet Big Data

If we have a graph too large to fit in the memory of a single computer, Spark lets us divide it among multiple computers in a cluster of computers. But what's the best way to split up a graph?

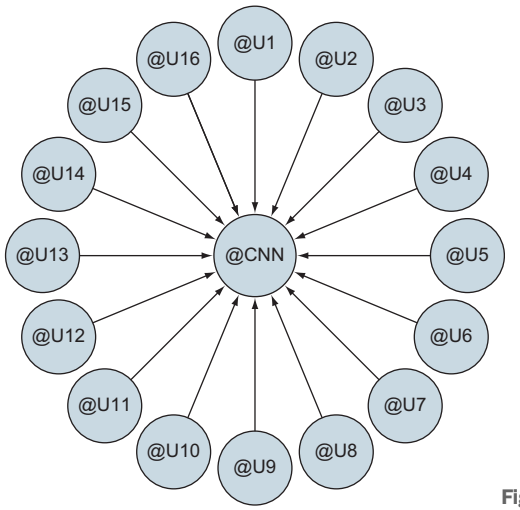


Figure 1.10 A graph with a high-degree vertex

The naïve way, and the way in which it was done for many years, was to assign different vertices to different computers in the cluster. But this led to computational bottlenecks because real-world graphs always seem to have some extremely high-degree vertices (see figure 1.10). The vertex degrees of real-world graphs tend to follow the *Power Law* (see the sidebar on the following page).

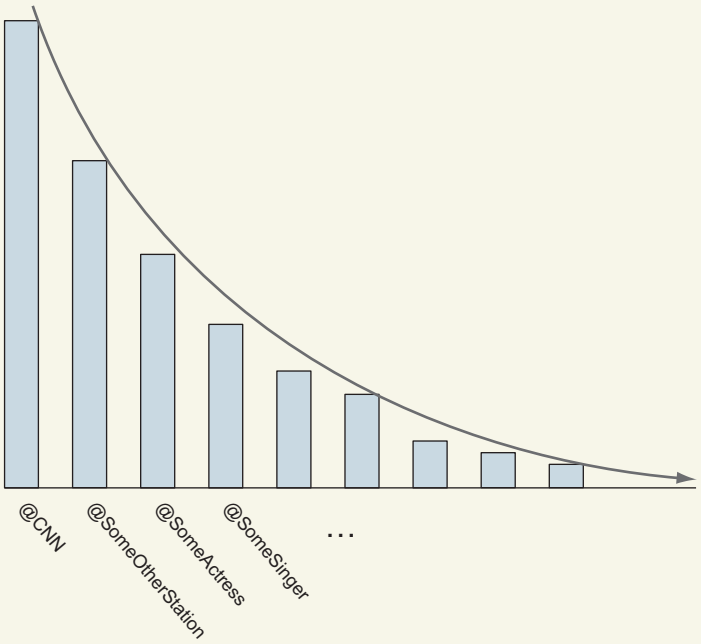
> **DEFINITION** The word *degree* also has two meanings when it comes to graphs. Earlier we used it in the context of the Six Degrees of Kevin Bacon, meaning the number of hops, or edges, from one actor to another, where an edge means the two actors appeared in the same film. But the degree of a vertex is completely different: it's the combined number of edges going out of or coming into a particular vertex. We won't be referring to Kevin Bacon anymore, so we'll use the word *hop* for those types of uses going forward, and *degree* only in the context of a vertex and the number of edges incident to it.

Partitioning a graph by vertices is called *edge-cut* because it's the edges that are getting cut. But a graph processing system that instead employs *vertex-cut*, which evenly distributes the edges among the machines/nodes, more evenly balances the data across the cluster. This idea came from research in 2005, was popularized by a graph processing system called GraphLab (now called PowerGraph), and was adopted by GraphX as the default partitioning scheme.

GraphX supports four different partitioning schemes for edges, described in section 9.4. GraphX partitions vertices independently of edges. By avoiding piling all the edges from a high-degree vertex onto a single machine, GraphX avoids the load imbalance suffered by earlier graph processing systems and graph databases.

Power Law of Graphs

Graphs in the real world have been found to obey the Power Law, which in the context of ranking the vertices by degree (intuitively, by popularity) means that the most popular vertex will be, say, 40% more popular than the second most popular vertex, which in turn will be 40% more popular than the third most popular vertex to it, and so on.



In this context of ranking, it is also known as Zipf's Law. These are the realities of graphs, and distributing graph data by the vertex-cut strategy balances graph data across a cluster. Spark GraphX employs the vertex-cut strategy by default.

1.3.3 GraphX lets you choose: graph parallel or data parallel

As we've seen, GraphX stores a graph's edges in one table and vertices in another. This allows graph algorithms implemented in GraphX to efficiently traverse graphs as graphs, along edges from one vertex to another, or as tables of edges or vertices (see figure 1.11). This latter mode of access permits efficient bulk transforms of edge or vertex data.

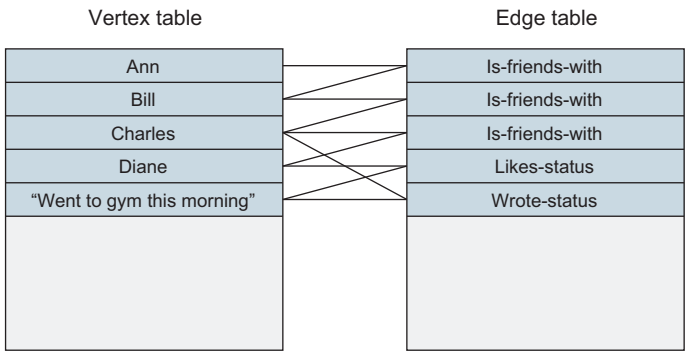
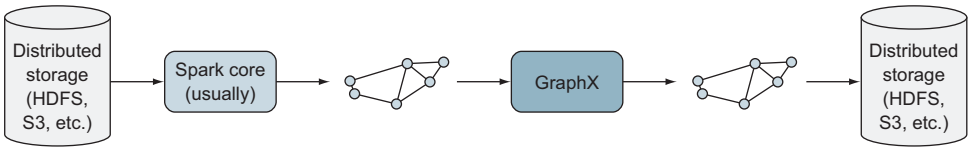


Figure 1.11 GraphX facilitates data access for either graph-parallel or data-parallel operations.

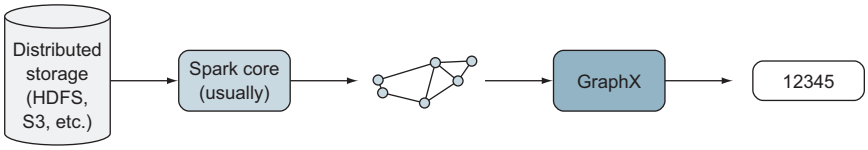
Although GraphX stores edges and vertices in separate tables as one might design an RDBMS schema to do, internally GraphX has special indexes to rapidly traverse the graph, and it exposes an API that makes graph querying and processing easier than trying to do the same in SQL.

1.3.4 Various ways GraphX fits into a processing flow

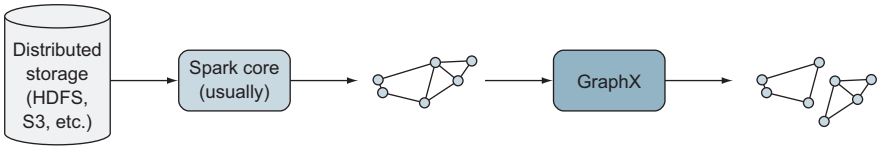
GraphX is inherently a batch-processing system. It doesn't integrate with Spark Streaming, for example (at least not in any straightforward way). There isn't one cookie-cutter way to use GraphX. There are many different batch processing data flows into which GraphX can fit, and the data flows in figures 1.12 and 1.13 cover some of these.



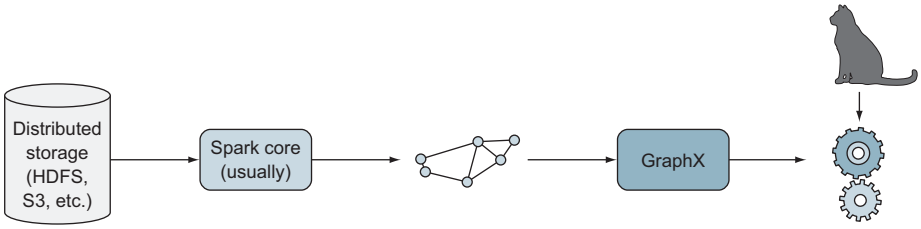
In this common workflow, a graph is transformed into a new graph (for example, vertices or edges may have new property values). An example of this is PageRank covered in sections 2.3 and 5.1.



Some graph algorithms, like the Global Clustering Coefficient from section 8.4, output only a global metric that describes the whole graph.



Other graph algorithms, like Connected Components from section 5.4, output subgraphs.



Machine learning algorithms implemented in GraphX , such as SVD++ (section 7.2), output a machine learning "model" which can then be used by themselves to make "predictions" – that is, when raw data is input into the model, the model itself outputs some data or label.

Figure 1.12 Various possible GraphX data flows. Because GraphX's capabilities for reading graph data files are so limited, data files usually have to be massaged and transformed using the Spark Core API into the graph format that GraphX uses. The output of a GraphX algorithm can be another graph, a number, some subgraphs, or a machine learning model.

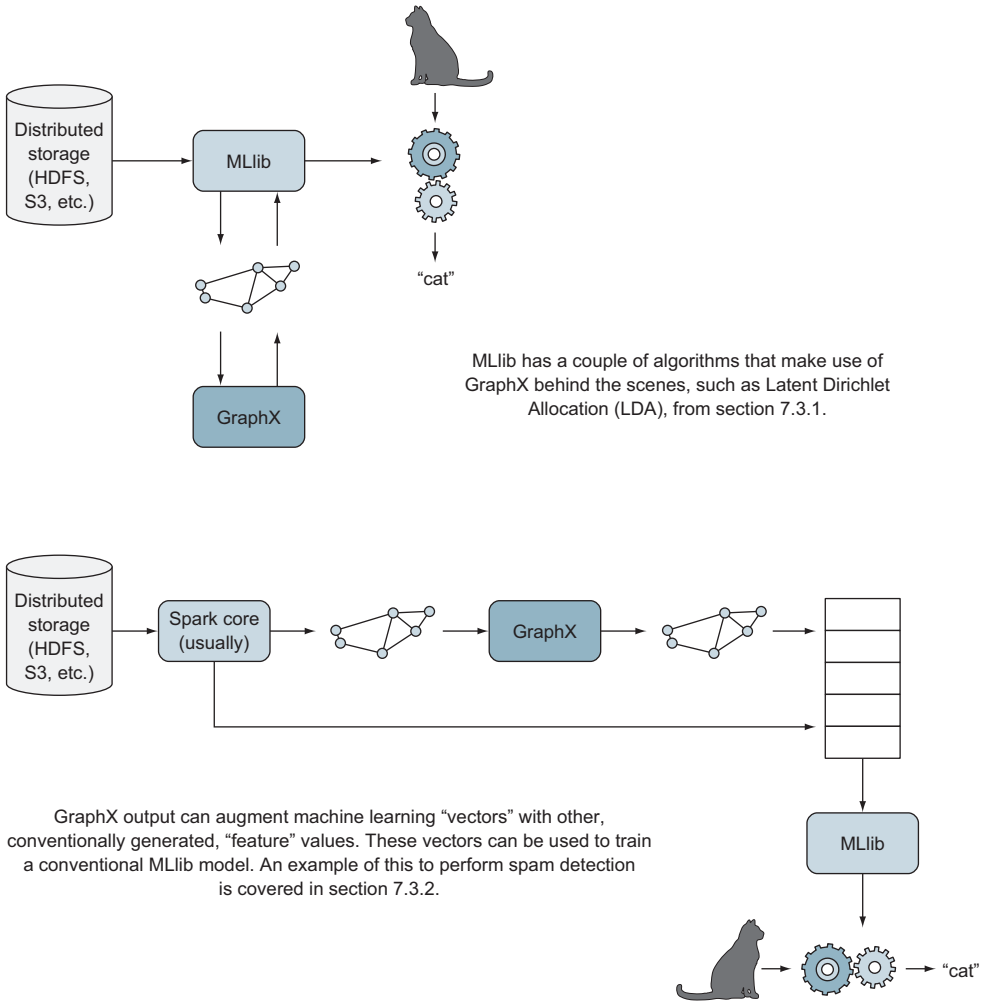


Figure 1.13 Data flows that involve using MLlib, the Spark machine learning component. A couple of algorithms use GraphX behind the scenes, but GraphX can also be used alongside any MLlib algorithm.

1.3.5 GraphX vs. other systems

Graph systems can be divided into two broad categories: graph processing systems and graph databases. Many are memory-based, and some even support cluster computing. Spark GraphX is a graph processing system rather than a graph database. A graph database has the great advantage of providing database transactions, a query language, and easy incremental updates and persistence, but if it's a disk-based graph database, it doesn't have the performance of a fully in-memory graph processing system like

GraphX. Graph processing systems are useful, for example, for fielding web service requests or performing one-off, long-running standalone computations.

Most graph analytics tasks require other types of processing as well. Graph analytics is usually one part of a larger processing pipeline. Often there's a need to generate a graph from raw data—say, from CSV or XML files. To generate the required property graph attributes, we may have to join data from another table. Once the graph processing task is completed, the resulting graph may need to be joined with other data. For example, we could use PageRank to find the most influential people in a social network. We could then use sales data from an RDBMS to find customers who are both influential and high-value to select the most promising recipients of a marketing promotion. Spark makes it easy to compose complex pipelines using both data-parallel and graph-parallel processing.

If you have a system that's already using Spark for other things and you also need to process graph data, Spark GraphX is a way to efficiently do that without having to learn and administer a completely different cluster technology, such as a separate distributed graph database. Because of GraphX's fast processing, you can even couple it to a graph database such as Neo4j and realize the best of both worlds: database transactions on the graph database and fast processing when you need it.

Apache Giraph is another example of a graph processing system, but again, Giraph is limited to slow Hadoop Map/Reduce. GraphX, Giraph, and GraphLab are all separate implementations of the ideas expressed in the Google Pregel paper. Neo4j, Titan, and Oracle Spatial and Graph are examples of graph databases. Graph databases have query languages that are convenient for finding information about a particular vertex or set of vertices. Pregel-based graph processing systems, in contrast, are bad at that and instead are good at executing massively parallel algorithms like PageRank. Now, GraphX does have the Spark REPL Shell, which provides an interactive command line interface into GraphX (as opposed to having to compile a program every time), and this speeds along development of GraphX applications and algorithms—but given the current syntax, it's still too cumbersome for querying, as shown in section 3.3.4.

GraphX is still young, and some of its limitations stem from the limitations of Spark. For example, GraphX datasets, like all Spark datasets, can't normally be shared by multiple Spark programs unless a REST server add-on like Spark JobServer is used. Until the IndexedRDD capability is added to Spark (Jira ticket SPARK-2365), which is effectively a mutable (that is, updatable) HashMap version of an RDD (Resilient Distributed Dataset, the foundation of Spark), GraphX is limited by the immutability of Spark RDDs, which is an issue for large graphs. Although faster for some uses, GraphX is often slower than systems written in C++, such as GraphLab/PowerGraph, due to GraphX's reliance on the JVM.

1.3.6 Storing the graphs: distributed file storage vs. graph database

Because GraphX is strictly an in-memory processing system, you need a place to store graph data. Spark expects distributed storage, such as HDFS, Cassandra, or S3, and storing graphs in distributed storage is the usual way to go.

But some use GraphX, a graph processing system, in conjunction with a graph database to get the best of both worlds (see figure 1.14). GraphX versus Neo4j is a frequent debate, but for some use cases, both are better than one or the other. The open source project Mazerunner is an extension to Neo4j that offloads graph analytics such as PageRank to GraphX.

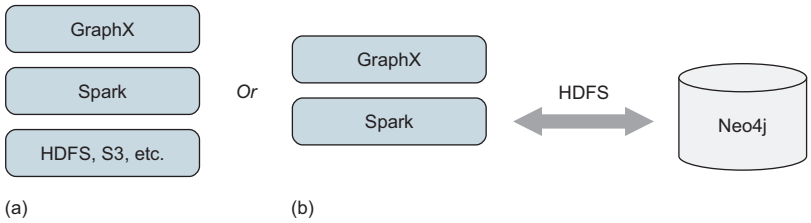


Figure 1.14 The conventional and by far most common way for GraphX to store its data is out to HDFS or to some other distributed storage system (a). Some, however, use the power of a full-fledged graph database, and realize the best of both worlds: transactions in a graph database and fast processing in GraphX (b).

1.4 Summary

- Graphs are a natural and powerful way to model connected data.
- Like Hadoop, Spark provides a Map/Reduce API (distributed computation) plus distributed storage. The main difference is that Spark stores data in RAM throughout the cluster, whereas Hadoop stores data on disk throughout the cluster.
- GraphX builds on the foundations of Spark to provide flexible and efficient graph-parallel processing.
- Spark also provides data-parallel processing that makes it ideal for real-world Big Data problems that often call for both graph-parallel and data-parallel processing.
- GraphX isn't a graph database and isn't suited to querying individual vertices or small groups of vertices. Rather, it's a graph processing system suited for massively parallel algorithms such as PageRank.
- Types of graph data include network, tree, relational, kitchen sink, and the graph equivalent to a sparse matrix.
- Graph algorithms include PageRank, recommender systems, shortest paths, community detection, and much more.

DATA SCIENCE/SCALA

Spark GraphX IN ACTION

Malak • East

GraphX is a powerful graph processing API for the Apache Spark analytics engine that lets you draw insights from large datasets. GraphX gives you unprecedented speed and capacity for running massively parallel and machine learning algorithms.

Spark GraphX in Action begins with the big picture of what graphs can be used for. This example-based tutorial teaches you how to use GraphX interactively. You'll start with a crystal-clear introduction to building big data graphs from regular data, and then explore the problems and possibilities of implementing graph algorithms and architecting graph processing pipelines. Along the way, you'll collect practical techniques for enhancing applications and applying machine learning algorithms to graph data.

What's Inside

- Understanding graph technology
- Using the GraphX API
- Developing algorithms for big graphs
- Machine learning with graphs
- Graph visualization

Readers should be comfortable writing code. Experience with Apache Spark and Scala is not required.

Michael Malak has worked on Spark applications for Fortune 500 companies since early 2013. **Robin East** has worked as a consultant to large organizations for over 15 years and is a data scientist at Worldpay.

"Learn complex graph processing from two experienced authors… A comprehensive guide."
—Gaurav Bhardwaj, 3Pillar Global

"The best resource to go from GraphX novice to expert in the least amount of time."
—Justin Fister, PaperRater

"A must-read for anyone serious about large-scale graph data mining!"
—Antonio Magnaghi
OpenMail

"Reveals the awesome and elegant capabilities of working with linked data for large-scale datasets."
—Sumit Pal
Independent consultant

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
www.manning.com/books/spark-graphx-in-action

ISBN 13: 978-1-61729-252-1
ISBN 10: 1-61729-252-4
54999
9 781617 292521

MANNING $49.99 / Can $57.99 [INCLUDING eBOOK]