CHAPTER 1

Three Database Revolutions

Fantasy. Lunacy.

All revolutions are, until they happen, then they are historical inevitabilities.

— David Mitchell, Cloud Atlas

We're still in the first minutes of the first day of the Internet revolution.

—Scott Cook

This book is about a third revolution in database technology. The first revolution was driven by the emergence of the electronic computer, and the second revolution by the emergence of the relational database. The third revolution has resulted in an explosion of nonrelational database alternatives driven by the demands of modern applications that require global scope and continuous availability. In this chapter we'll provide an overview of these three waves of database technologies and discuss the market and technology forces leading to today's next generation databases.

Figure 1-1 shows a simple timeline of major database releases.

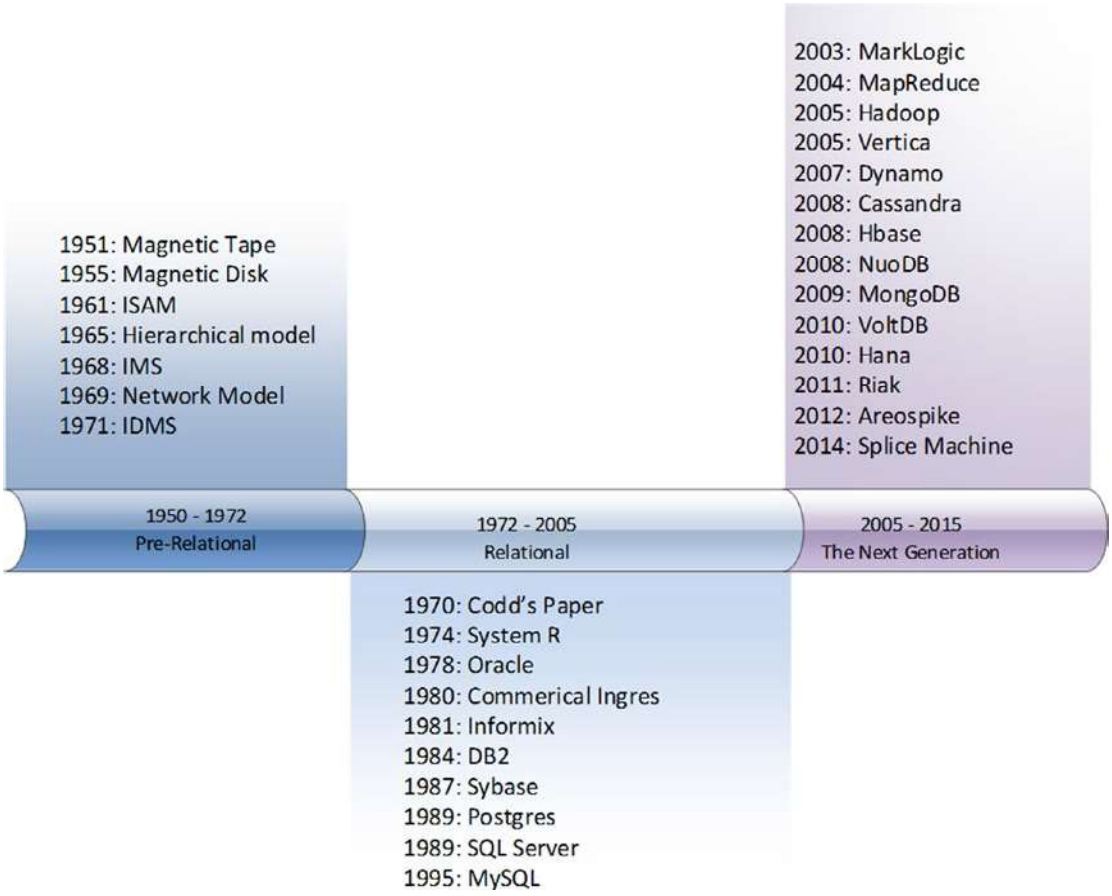

Figure 1-1. *Timeline of major database releases and innovations*

Figure 1-1 illustrates three major eras in database technology. In the 20 years following the widespread adoption of electronic computers, a range of increasingly sophisticated database systems emerged. Shortly after the definition of the relational model in 1970, almost every significant database system shared a common architecture. The three pillars of this architecture were the relational model, ACID transactions, and the SQL language.

However, starting around 2008, an explosion of new database systems occurred, and none of these adhered to the traditional relational implementations. These new database systems are the subject of this book, and this chapter will show how the preceding waves of database technologies led to this next generation of database systems.

Early Database Systems

Wikipedia defines a *database* as an "organized collection of data." Although the term *database* entered our vocabulary only in the late 1960s, collecting and organizing data has been an integral factor in the development of human civilization and technology. Books—especially those with a strongly enforced structure such as dictionaries and encyclopedias—represent datasets in physical form. Libraries and other indexed archives of information represent preindustrial equivalents of modern database systems.

We can also see the genesis of the digital database in the adoption of punched cards and other physical media that could store information in a form that could be processed mechanically. In the 19th century, loom cards were used to "program" fabric looms to generate complex fabric patterns, while tabulating machines used punched cards to produce census statistics, and player pianos used perforated paper strips that represented melodies. Figure 1-2 shows a Hollerith tabulating machine being used to process the U.S. census in 1890.

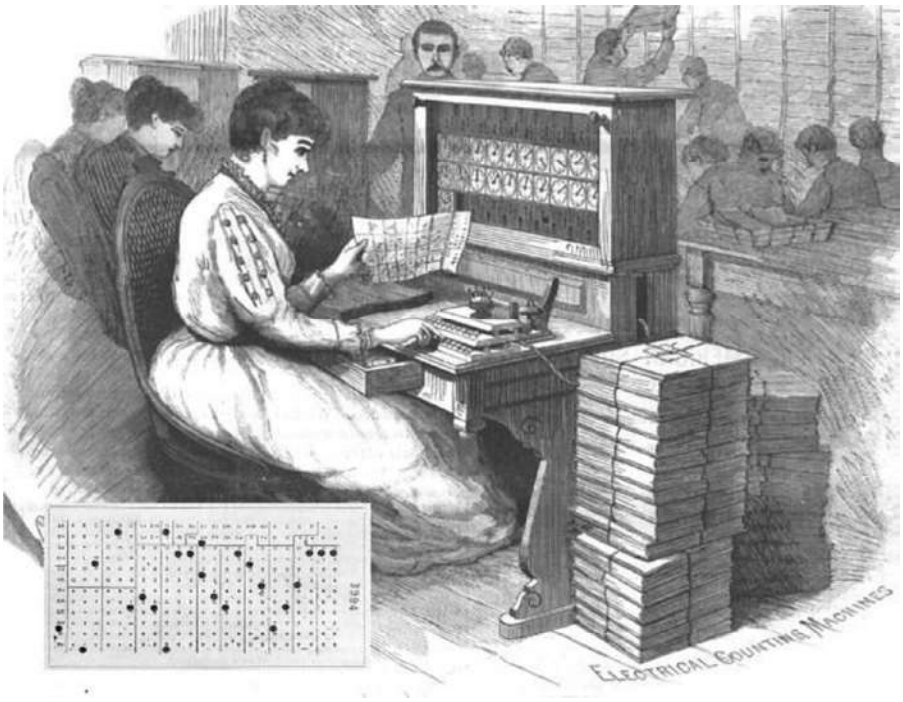

Figure 1-2. *Tabulating machines and punched cards used to process 1890 U.S. census*

The emergence of electronic computers following the Second World War represented the first revolution in databases. Some early digital computers were created to perform purely mathematical functions—calculating ballistic tables, for instance. But equally as often they were intended to operate on and manipulate data, such as processing encrypted Axis military communications.

Early "databases" used paper tape initially and eventually magnetic tape to store data sequentially. While it was possible to "fast forward" and "rewind" through these datasets, it was not until the emergence of the spinning magnetic disk in the mid-1950s that direct high-speed access to individual records became possible. Direct access allowed fast access to any item within a file of any size. The development of indexing methods such as ISAM (Index Sequential Access Method) made fast record-oriented access feasible and consequently allowed for the birth of the first OLTP (On-line Transaction Processing) computer systems.

ISAM and similar indexing structures powered the first electronic databases. However, these were completely under the control of the application—there were databases but no *Database Management Systems* (*DBMS*).

The First Database Revolution

Requiring every application to write its own data handling code was clearly a productivity issue: every application had to reinvent the database wheel. Furthermore, errors in application data handling code led inevitably to corrupted data. Allowing multiple users to concurrently access or change data without logically or physically corrupting the data requires sophisticated coding. Finally, optimization of data access through caching, pre-fetch, and other techniques required complicated and specialized algorithms that could not easily be duplicated in each application.

Therefore, it became desirable to externalize database handling logic from the application in a separate code base. This layer—the Database Management System, or DBMS—would minimize programmer overhead and ensure the performance and integrity of data access routines.

Early database systems enforced both a schema (a definition of the structure of the data within the database) and an access path (a fixed means of navigating from one record to another). For instance, the DBMS might have a definition of a CUSTOMER and an ORDER together with a defined access path that allowed you to retrieve the orders associated with a particular customer or the customer associated with a specific order.

These first-generation databases ran exclusively on the mainframe computer systems of the day — largely IBM mainframes. By the early 1970s, two major models of DBMS were competing for dominance. The *network model* was formalized by the CODASYL standard and implemented in databases such as IDMS, while the *hierarchical model* provided a somewhat simpler approach as was most notably found in IBM's IMS (Information Management System). Figure 1-3 provides a comparison of these databases' structural representation of data.

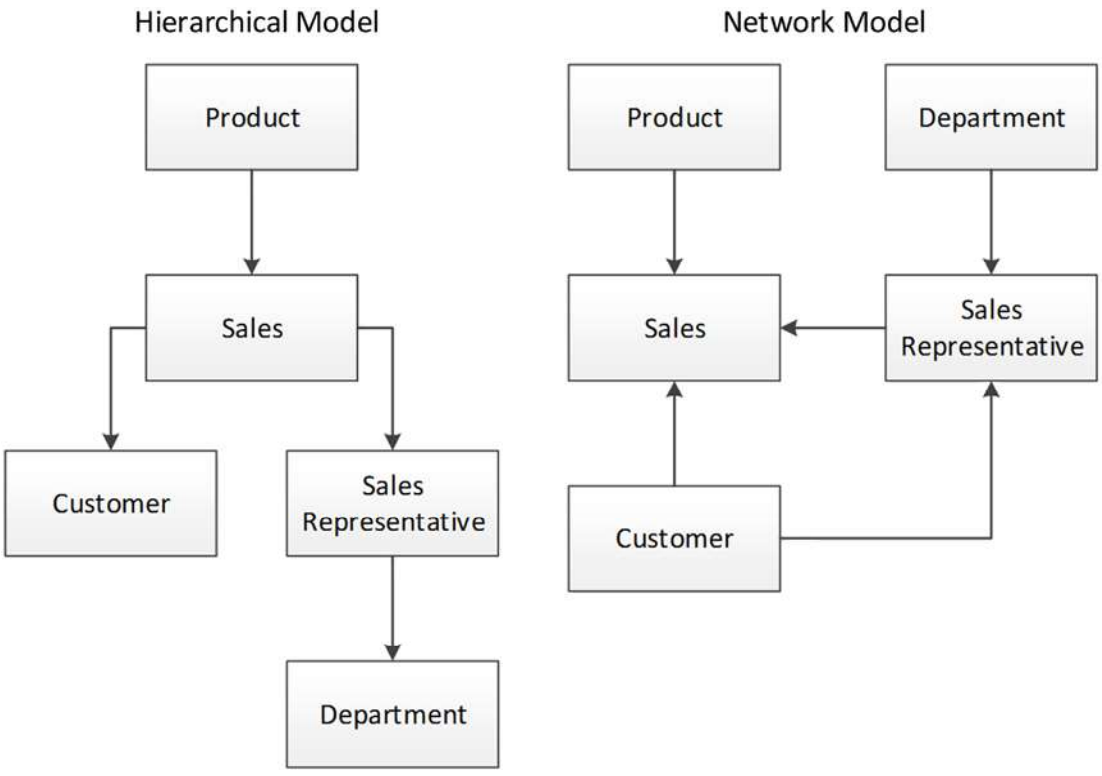

Figure 1-3. *Hierarchical and network database models*

■ **Note** These early systems are often described as "navigational" in nature because you must navigate from one object to another using pointers or links. For instance, to find an order in a hierarchical database, it may be necessary to first locate the customer, then follow the link to the customer's orders.

Hierarchical and network database systems dominated during the era of mainframe computing and powered the vast majority of computer applications up until the late 1970s. However, these systems had several notable drawbacks.

First, the navigational databases were extremely inflexible in terms of data structure and query capabilities. Generally only queries that could be anticipated during the initial design phase were possible, and it was extremely difficult to add new data elements to an existing system.

Second, the database systems were centered on record at a time transaction processing—what we today refer to as CRUD (Create, Read, Update, Delete). Query operations, especially the sort of complex analytic queries that we today associate with business intelligence, required complex coding. The business demands for analytic-style reports grew rapidly as computer systems became increasingly integrated with business processes. As a result, most IT departments found themselves with huge backlogs of report requests and a whole generation of computer programmers writing repetitive COBOL report code.

The Second Database Revolution

Arguably, no single person has had more influence over database technology than Edgar Codd. Codd received a mathematics degree from Oxford shortly after the Second World War and subsequently immigrated to the United States, where he worked for IBM on and off from 1949 onwards. Codd worked as a "programming mathematician" (ah, those were the days) and worked on some of IBM's very first commercial electronic computers.

In the late 1960s, Codd was working at an IBM laboratory in San Jose, California. Codd was very familiar with the databases of the day, and he harbored significant reservations about their design. In particular, he felt that:

- **Existing databases were too hard to use**. Databases of the day could only be accessed by people with specialized programming skills.
- **Existing databases lacked a theoretical foundation**. Codd's mathematical background encouraged him to think about data in terms of formal structures and logical operations; he regarded existing databases as using arbitrary representations that did not ensure logical consistency or provide the ability to deal with missing information.
- **Existing databases mixed logical and physical implementations**. The representation of data in existing databases matched the format of the physical storage in the database, rather than a logical representation of the data that could be comprehended by a nontechnical user.

Codd published an internal IBM paper outlining his ideas for a more formalized model for database systems, which then led to his 1970 paper "A Relational Model of Data for Large Shared Data Banks."[1] This classic paper contained the core ideas that defined the *relational database model* that became the most significant—almost universal—model for database systems for a generation.

Relational theory

The intricacies of relational database theory can be complex and are beyond the scope of this introduction. However, at its essence, the relational model describes how a given set of data should be presented to the user, rather than how it should be stored on disk or in memory. Key concepts of the relational model include:

- **Tuples**, an unordered set of **attribute** values. In an actual database system, a tuple corresponds to a row, and an attribute to a column value.
- A **relation**, which is a collection of distinct tuples and corresponds to a table in relational database implementations.
- **Constraints**, which enforce consistency of the database. Key constraints are used to identify tuples and relationships between tuples.
- **Operations** on relations such as joins, projections, unions, and so on. These operations always return relations. In practice, this means that a query on a table returns data in a tabular format.

A row in a table should be identifiable and efficiently accessed by a unique key value, and every column in that row must be dependent on that key value and no other identifier. Arrays and other structures that contain nested information are, therefore, not directly supported.

Levels of conformance to the relational model are described in the various "*normal forms*." *Third normal form* is the most common level. Database practitioners typically remember the definition of third normal form by remembering that all non-key attributes must be dependent on "the key, the whole key, and nothing but the key—So Help Me Codd"![2]

Figure 1-4 provides an example of normalization: the data on the left represents a fairly simple collection of data. However, it contains redundancy in student and test names, and the use of a repeating set of attributes for the test answers is dubious (while possibly within relational form, it implies that each test has the same number of questions and makes certain operations difficult). The five tables on the right represent a normalized representation of this data.

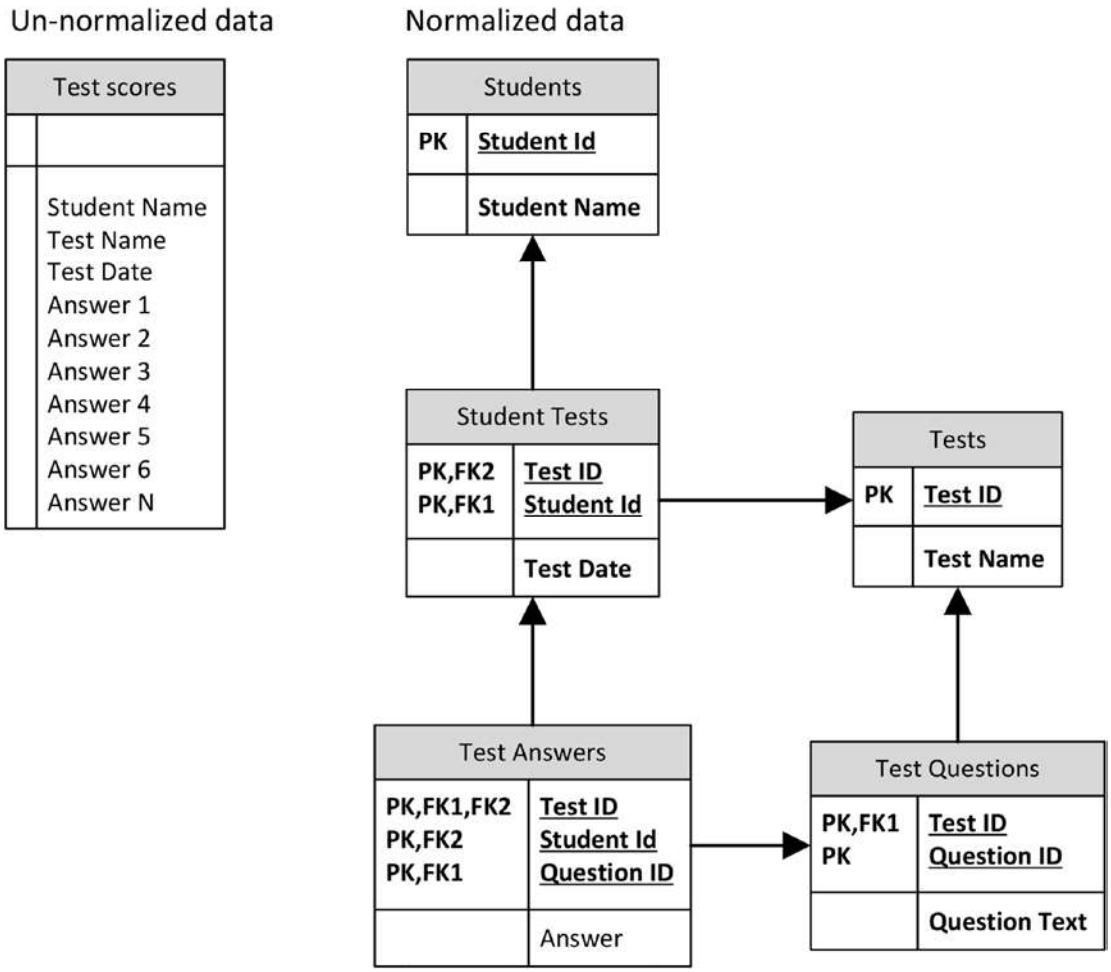

Figure 1-4. *Normalized and un-normalized data*

Transaction Models

The relational model does not itself define the way in which the database handles concurrent data change requests. These changes—generally referred to as database *transactions*—raise issues for all database systems because of the need to ensure consistency and integrity of data.

Jim Gray defined the most widely accepted transaction model in the late 1970s. As he put it, "A transaction is a transformation of state which has the properties of atomicity (all or nothing), durability (effects survive failures) and consistency (a correct transformation)."[3] This soon became popularized as *ACID transactions*: Atomic, Consistent, Independent, and Durable. An ACID transaction should be:

- **Atomic**: The transaction is indivisible—either all the statements in the transaction are applied to the database or none are.
- **Consistent**: The database remains in a consistent state before and after transaction execution.

- **Isolated**: While multiple transactions can be executed by one or more users simultaneously, one transaction should not see the effects of other in-progress transactions.
- **Durable**: Once a transaction is saved to the database (in SQL databases via the COMMIT command), its changes are expected to persist even if there is a failure of operating system or hardware.

ACID transactions became the standard for all serious database implementations, but also became most strongly associated with the relational databases that were emerging at about the time of Gray's paper.

As we will see later, the restriction on scalability beyond a single data center implied by the ACID transaction model has been a key motivator for the development of new database architectures.

The First Relational Databases

Initial reaction to the relational model was somewhat lukewarm. Existing vendors including IBM were disinclined to accept Codd's underlying assumption: that the databases of the day were based on a flawed foundation. Furthermore, many had sincere reservations about the ability of a system to deliver adequate performance if the data representation was not fine-tuned to the underlying access mechanisms. Would it be possible to create a high-performance database system that allowed data to be accessed in any way the user could possibly imagine?

IBM did, however, initiate a research program to develop a prototype relational database system in 1974, called System R. System R demonstrated that relational databases could deliver adequate performance, and it pioneered the *SQL language*. (Codd had specified that the relational system should include a query language, but had not mandated a specific syntax.) Also during this period, Mike Stonebraker at Berkeley started work on a database system that eventually was called *INGRES*. INGRES was also relational, but it used a non-SQL query language called *QUEL*.

At this point, Larry Ellison enters our story. Ellison was more entrepreneurial than academic by nature, though extremely technically sophisticated, having worked at Amdahl. Ellison was familiar both with Codd's work and with System R, and he believed that relational databases represented the future of database technology. In 1977, Ellison founded the company that would eventually become Oracle Corporation and which would release the first commercially successful relational database system.

Database Wars!

It was during this period that minicomputers challenged and eventually ended the dominance of the mainframe computer. Compared with today's computer hardware, the minicomputers of the late '70s and early '80s were hardly "mini". But unlike mainframes, they required little or no specialized facilities, and they allowed mid-size companies for the first time to own their own computing infrastructure. These new hardware platforms ran new operating systems and created a demand for new databases that could run on these operating systems.

By 1981, IBM had released a commercial relational database called SQL/DS, but since it only ran on IBM mainframe operating systems, it had no influence in the rapidly growing minicomputer market. Ellison's *Oracle* database system was commercially released in 1979 and rapidly gained traction on the minicomputers provided by companies such as Digital and Data General. At the same time, the Berkeley INGRES project had given birth to the commercial relational database *Ingres*. Oracle and Ingres fought for dominance in the early minicomputer relational database market.

By the mid-'80s, the benefits of the relational database—if not the nuances of relational theory—had become widely understood. Database buyers appreciated in particular that the SQL language, now adopted by all vendors including Ingres, provided massive productivity gains for report writing and analytic queries. Furthermore, a next generation of database development tools—known at the time as 4GLs—were

becoming increasingly popular and these new tools typically worked best with relational database servers. Finally, minicomputers offered clear price/performance advantages over mainframes especially in the midmarket, and here the relational database was pretty much the only game in town.

Indeed, relational databases became so dominant in terms of mindshare that the vendors of the older database systems became obliged to describe their offerings as also being relational. This prompted Codd to pen his famous 12 rules (actually 13 rules, starting at rule 0) as a sort of acid test to distinguish legitimate relational databases from pretenders.

During the succeeding decades many new database systems were introduced. These include *Sybase*, Microsoft *SQL Server*, *Informix*, *MySQL*, and *DB2*. While each of these systems attempts to differentiate by claiming superior performance, availability, functionality, or economy, they are virtually identical in their reliance on three key principles: Codd's relational model, the SQL language, and the ACID transaction model.

■ **Note** When we say RDBMS, we generally refer to a database that implements the relational data model, supports ACID transactions, and uses SQL for query and data manipulation.

Client-server Computing

By the late 1980s, the relational model had clearly achieved decisive victory in the battle for database mindshare. This mindshare dominance translated into market dominance during the shift to *client-server computing*.

Minicomputers were in some respects "little mainframes": in a minicomputer application, all processing occurred on the minicomputer itself, and the user interacted with the application through dumb "green screen" terminals. However, even as the minicomputer was becoming a mainstay of business computing, a new revolution in application architecture was emerging.

The increasing prevalence of microcomputer platforms based on the IBM PC standard, and the emergence of graphical user interfaces such as Microsoft Windows, prompted a new application paradigm: client-server. In the client-server model, presentation logic was hosted on a PC terminal typically running Microsoft Windows. These PC-based client programs communicated with a database server typically running on a minicomputer. Application logic was often concentrated on the client side, but could also be located within the database server using the *stored procedures*—programs that ran inside the database.

Client-server allowed for a richness of experience that was unparalleled in the green-screen era, and by the early '90s, virtually all new applications aspired to the client-server architecture. Practically all client-development platforms assumed an RDBMS backend—indeed, usually assumed SQL as the vehicle for all requests between client and server.

Object-oriented Programming and the OODBMS

Shortly after the client-server revolution, another significant paradigm shift impacted mainstream application-development languages. In traditional "procedural" programming languages, data and logic were essentially separate. Procedures would load and manipulate data within their logic, but the procedure itself did not contain the data in any meaningful way. *Object-oriented (OO) programming* merged attributes and behaviors into a single object. So, for instance, an employee object might represent the structure of employee records as well as operations that can be performed on those

records—changing salary, promoting, retiring, and so on. For our purposes, the two most relevant principles of object-oriented programming are:

- **Encapsulation**: An object class encapsulates both data and actions (methods) that may be performed on that data. Indeed, an object may restrict direct access to the underlying data, requiring that modifications to the data be possible only via an objects methods. For instance, an employee class might include a method to retrieve salary and another method to modify salary. The salary-modification method might include restrictions on minimum and maximum salaries, and the class might allow for no manipulation of salary outside of these methods.
- **Inheritance**: Object classes can inherit the characteristics of a parent class. The employee class might inherit all the properties of a people class (DOB, name, etc.) while adding properties and methods such as salary, employee date, and so on.

Object-oriented programming represented a huge gain in programmer productivity, application reliability, and performance. Throughout the late '80s and early '90s, most programming languages converted to an object-oriented model, and many significant new languages— such as Java—emerged that were natively object-oriented.

The object-oriented programming revolution set the stage for the first serious challenge to the relational database, which came along in the mid-1990s. Object-oriented developers were frustrated by what they saw as an impedance mismatch between the object-oriented representations of their data within their programs and the relational representation within the database. In an object-0riented program, all the details relevant to a logical unit of work would be stored within the one class or directly linked to that class. For instance, a customer object would contain all details about the customer, with links to objects that contained customer orders, which in turn had links to order line items. This representation was inherently nonrelational; indeed, the representation of data matched more closely to the network databases of the CODASYL era.

When an object was stored into or retrieved from a relational database, multiple SQL operations would be required to convert from the object-0riented representation to the relational representation. This was cumbersome for the programmer and could lead to performance or reliability issues. Figure 1-5 illustrates the problem.

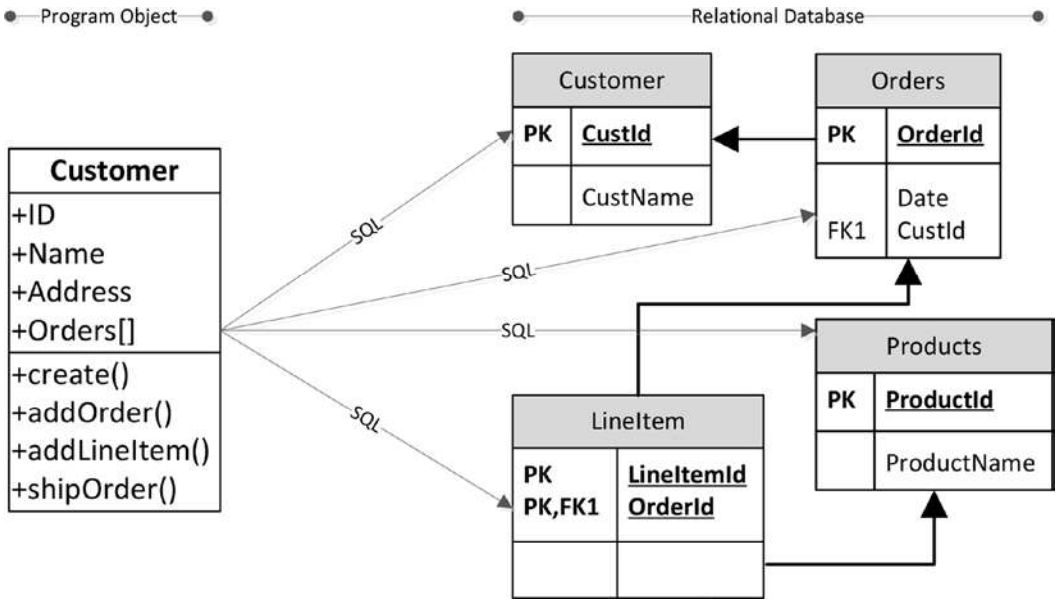

Figure 1-5. *Storing an object in an RDBMS requires multiple SQL operations*

Advocates of object-oriented programming began to see the relational database as a relic of the procedural past. This led to the rather infamous quote: "A relational database is like a garage that forces you to take your car apart and store the pieces in little drawers."

The rapid success of object-oriented programming led almost inevitably to the proposition that an *Object Oriented Database Management System* (OODBMS) was better suited to meet the demands of modern applications. An OODBMS would store program objects directly without normalization, and would allow applications to load and store their objects easily. The object-oriented database movement created a manifesto outlining the key arguments for and properties of OODBMS.[4] In implementation, OODBMS resembles the navigational model from the pre-relational era—pointers within one object (a customer, for instance) would allow navigation to related objects (such as orders).

Advocacy for the OODBMS model grew during the mid-'90s, and to many it did seem natural that the OODBMS would be the logical successor to the RDBMS. The incumbent relational database vendors—at this point, primarily Oracle, Informix, Sybase, and IBM—rapidly scrambled to implement OODBMS features within their RDBMS. Meanwhile, some pure OODBMS systems were developed and gained initial traction.

However, by the end of the decade, OODBMS systems had completely failed to gain market share. Mainstream database vendors such as Oracle and Informix had successfully implemented many OODBMS features, but even these features were rarely used. OO programmers became resigned to the use of RDBMS systems to persist objects, and the pain was somewhat alleviated by *Object-Relational Mapping* (*ORM*) frameworks that automated the most tedious aspects of the translation.

There are competing and not necessarily contradictory explanations for the failure of the OO database. For my part, I felt the advocates of the OODBMS model were concentrating on only the advantages an OODBMS offered to the application developer, and ignoring the disadvantages the new model had for those who wished to consume information for business purposes. Databases don't exist simply for the benefit of programmers; they represent significant assets that must be accessible to those who want to mine the information for decision making and business intelligence. By implementing a data model that could only be used by the programmer, and depriving the business user of a usable SQL interface, the OODBMS failed to gain support outside programming circles.

However, as we shall see in Chapter 4, the motivations for an OODBMS heavily influenced some of today's most popular nonrelational databases.

The Relational Plateau

Once the excitement over object-oriented databases had run its course, relational databases remained unchallenged until the latter half of the 2000s. In fact, for a period of roughly 10 years (1995–2005), no significant new databases were introduced: there were already enough RDBMS systems to saturate the market, and the stranglehold that the RDBMS model held on the market meant no nonrelational alternatives could emerge. Considering that this period essentially represents a time when the Internet grew from geeky curiosity to an all-pervasive global network, that no new database architectures emerged during this period is astonishing, and it is a testament to the power of the relational model.

The Third Database Revolution

By the middle of the 2000s, the relational database seemed completely entrenched. Looking forward from 2005, it seemed that although we would see continuing and significant innovation within the relational database systems of the day, there were no signs of any radical changes to come. But in fact, the era of complete relational database supremacy was just about to end.

In particular, the difference in application architectures between the client-server era and the era of massive web-scale applications created pressures on the relational database that could not be relieved through incremental innovation.

Google and Hadoop

By 2005, Google was by far the biggest website in the world—and this had been true since a few years after Google first launched. When Google began, the relational database was already well established, but it was inadequate to deal with the volumes and velocity of the data confronting Google. The challenges that enterprises face with "big data" today are problems that Google first encountered almost 20 years ago. Very early on, Google had to invent new hardware and software architectures to store and process the exponentially growing quantity of websites it needed to index.

In 2003, Google revealed details of the distributed file system *GFS* that formed a foundation for its storage architecture,[5] and in 2004 it revealed details of the distributed parallel processing algorithm *MapReduce,* which was used to create World Wide Web indexes.[6] In 2006, Google revealed details about its *BigTable* distributed structured database.[7]

These concepts, together with other technologies, many of which also came from Google, formed the basis for the *Hadoop* project, which matured within Yahoo! and which experienced rapid uptake from 2007 on. The Hadoop ecosystem more than anything else became a technology enabler for the Big Data ecosystem we'll discuss in more detail in Chapter 2.

The Rest of the Web

While Google had an overall scale of operation and data volume way beyond that of any other web company, other websites had challenges of their own. Websites dedicated to online e-commerce—Amazon, for example—had a need for a transactional processing capability that could operate at massive scale. Early social networking sites such as MySpace and eventually Facebook faced similar challenges in scaling their infrastructure from thousands to millions of users.

Again, even the most expensive commercial RDBMS such as Oracle could not provide sufficient scalability to meet the demands of these sites. Oracle's scaled-out RDBMS architecture (Oracle RAC) attempted to provide a roadmap for limitless scalability, but it was economically unattractive and never seemed to offer the scale required at the leading edge.

Many early websites attempted to scale open-source databases through a variety of do-it-yourself techniques. This involved utilizing distributed object cases such as Memcached to offload database load, database replication to spread database read activity, and eventually—when all else failed—"Sharding."

Sharding involves partitioning the data across multiple databases based on a key attribute, such as the customer identifier. For instance, in Twitter and Facebook, customer data is split up across a very large number of MySQL databases. Most data for a specific user ends up on the one database, so that operations for a specific customer are quick. It's up to the application to work out the correct shard and to route requests appropriately.

Sharding at sites like Facebook has allowed a MySQL-based system to scale up to massive levels, but the downsides of doing this are immense. Many relational operations and database-level ACID transactions are lost. It becomes impossible to perform joins or maintain transactional integrity across shards. The operational costs of sharding, together with the loss of relational features, made many seek alternatives to the RDBMS.

Meanwhile, a similar dilemma within Amazon had resulted in development of an alternative model to strict ACID consistency within its homegrown data store. Amazon revealed details of this system, "Dynamo," in 2008.[8]

Amazon's Dynamo model, together with innovations from web developers seeking a "webscale" database, led to the emergence of what came to be known as *key-value databases*. We'll discuss these in more detail in Chapter 3.

Cloud Computing

The existence of applications and databases "in the cloud"—that is, accessed from the Internet—had been a persistent feature of the application landscape since the late 1990s. However, around 2008, cloud computing erupted somewhat abruptly as a major concern for large organizations and a huge opportunity for startups.

For the previous 5 to 10 years, mainstream adoption of computer applications had shifted from rich desktop applications based on the client-server model to web-based applications whose data stores and application servers resided somewhere accessible via the Internet—"the cloud." This created a real challenge for emerging companies that needed somehow to establish sufficient hosting for early adopters, as well as the ability to scale up rapidly should they experience the much-desired exponential growth.

Between 2006 and 2008, Amazon rolled out *Elastic Compute Cloud* (*EC2*). EC2 made available virtual machine images hosted on Amazon's hardware infrastructure and accessible via the Internet. EC2 could be used to host web applications, and computing power could be relatively rapidly added on demand. Amazon added other services such as storage (S3, EBS), Virtual Private Cloud (VPC), a MapReduce service (EMR), and so on. The entire platform was known as *Amazon Web Services* (*AWS*) and was the first practical implementation of an *Infrastructure as a Service* (*IaaS*) cloud. AWS became the inspiration for cloud computing offerings from Google, Microsoft, and others.

For applications wishing to exploit the elastic scalability allowed by cloud computing platforms, existing relational databases were a poor fit. Oracle's attempts to integrate grid computing into its architecture had met with only limited success and were economically and practically inadequate for these applications, which needed to be able to expand on demand. That demand for elastically scalable databases fueled the demand generated by web-based startups and accelerated the growth of key-value stores, often based on Amazon's own Dynamo design. Indeed, Amazon offered nonrelational services in its cloud starting with *SimpleDB,* which eventually was replaced by *DynamoDB*.

Document Databases

Programmers continued to be unhappy with the impedance mismatch between object-oriented and relational models. Object relational mapping systems only relieved a small amount of the inconvenience that occurred when a complex object needed to be stored on a relational database in normal form.

Starting about 2004, an increasing number of websites were able to offer a far richer interactive experience than had been the case previously. This was enabled by the programming style known as *AJAX (Asynchronous JavaScript and XML),* in which JavaScript within the browser communicates directly with a backend by transferring XML messages. XML was soon superseded by *JavaScript Object Notation* (*JSON*), which is a self-describing format similar to XML but is more compact and tightly integrated into the JavaScript language.

JSON became the de facto format for storing—serializing—objects to disk. Some websites started storing JSON documents directly into columns within relational tables. It was only a matter of time before someone decided to eliminate the relational middleman and create a database in which JSON could be directly stored. These became known as *document databases*.

CouchBase and *MongoDB* are two popular JSON-oriented databases, though virtually all nonrelational databases—and most relational databases, as well—support JSON. Programmers like document databases for the same reasons they liked OODBMS: it relieves them of the laborious process of translating objects to relational format. We'll look at document databases in some detail in Chapter 4.

The "NewSQL"

Neither the relational nor the ACID transaction model dictated the physical architecture for a relational database. However, partly because of a shared ancestry and partly because of the realities of the hardware of the day, most relational databases ended up being implemented in a very similar manner. The format of data on disk, the use of memory, the nature of locks, and so on varied only slightly among the major RDBMS implementations.

In 2007, Michael Stonebraker, pioneer of the Ingres and Postgres database systems, led a research team that published the seminal paper "The End of an Architectural Era (It's Time for a Complete Rewrite)."[9] This paper pointed out that the hardware assumptions that underlie the consensus relational architecture no longer applied, and that the variety of modern database workloads suggested a single architecture might not be optimal across all workloads.

Stonebraker and his team proposed a number of variants on the existing RDBMS design, each of which was optimized for a specific application workload. Two of these designs became particularly significant (although to be fair, neither design was necessarily completely unprecedented). *H-Store* described a pure in-memory distributed database while *C-Store* specified a design for a columnar database. Both these designs were extremely influential in the years to come and are the first examples of what came to be known as *NewSQL* database systems—databases that retain key characteristics of the RDBMS but that diverge from the common architecture exhibited by traditional systems such as Oracle and SQL Server. We'll examine these database types in Chapters 6 and 7.

The Nonrelational Explosion

As we saw in Figure 1-1, a huge number of relational database systems emerged in the first half of the 2000s. In particular, a sort of "Cambrian explosion" occurred in the years 2008–2009: literally dozens of new database systems emerged in this short period. Many of these have fallen into disuse, but some—such as MongoDB, Cassandra, and HBase—have today captured significant market share.

At first, these new breeds of database systems lacked a common name. "Distributed Non-Relational Database Management System" (DNRDBMS) was proposed, but clearly wasn't going to capture anybody's imagination. However, in late 2009, the term *NoSQL* quickly caught on as shorthand for any database system that broke with the traditional SQL database.

In the opinion of many, NoSQL is an unfortunate term: it defines what a database is not rather than what it is, and it focuses attention on the presence or absence of the SQL language. Although it's true that most nonrelational systems do not support SQL, actually it is variance from the strict transactional and relational data model that motivated most NoSQL database designs.

By 2011, the term *NewSQL* became popularized as a means of describing this new breed of databases that, while not representing a complete break with the relational model, enhanced or significantly modified the fundamental principles—and this included columnar databases, discussed in Chapter 6, and in some of the in-memory databases discussed in Chapter 7.

Finally, the term *Big Data* burst onto mainstream consciousness in early 2012. Although the term refers mostly to the new ways in which data is being leveraged to create value, we generally understand "Big Data solutions" as convenient shorthand for technologies that support large and unstructured datasets such as Hadoop.

■ **Note** NoSQL, NewSQL, and Big Data are in many respects vaguely defined, overhyped, and overloaded terms. However, they represent the most widely understood phrases for referring to next-generation database technologies.

Loosely speaking, NoSQL databases reject the constraints of the relational model, including strict consistency and schemas. NewSQL databases retain many features of the relational model but amend the underlying technology in significant ways. Big Data systems are generally oriented around technologies within the Hadoop ecosystem, increasingly including Spark.

Conclusion: One Size Doesn't Fit All

The first database revolution arose as an inevitable consequence of the emergence of electronic digital computers. In some respect, the databases of the first wave were electronic analogs of pre-computer technologies such as punched cards and tabulating machines. Early attempts to add a layer of structure and consistency to these databases may have improved programmer efficiency and data consistency, but they left the data locked in systems to which only programmers held the keys.

The second database revolution resulted from Edgar Codd's realization that database systems would be well served if they were based on a solid, formal, and mathematical foundation; that the representation of data should be independent of the physical storage implementation; and that databases should support flexible query mechanisms that do not require sophisticated programming skills.

The successful development of the modern relational database over such an extended time —more than 30 years of commercial dominance—represents a triumph of computer science and software engineering. Rarely has a software theoretical concept been so successfully and widely implemented as the relational database.

The third database revolution is not based on a single architectural foundation. If anything, it rests on the proposition that a single database architecture cannot meet the challenges posed in our modern digital world. The existence of massive social networking applications with hundreds of millions of users and the emergence of *Internet of Things* (*IoT*) applications with potentially billions of machine inputs, strain the relational database—and particularly the ACID transaction model—to the breaking point. At the other end of the scale we have applications that must run on mobile and wearable devices with limited memory and computing power. And we are awash with data, much of which is of unpredictable structure for which rendering to relational form is untenable.

The third wave of databases roughly corresponds to a third wave of computer applications. IDC and others often refer to this as "the third platform." The first platform was the mainframe, which was supported by pre-relational database systems. The second platform, client-server and early web applications, was supported by relational databases. The third platform is characterized by applications that involve cloud deployment, mobile presence, social networking, and the Internet of Things. The third platform demands a third wave of database technologies that include but are not limited to relational systems. Figure 1-6 summarizes how the three platforms correspond to our three waves of database revolutions.

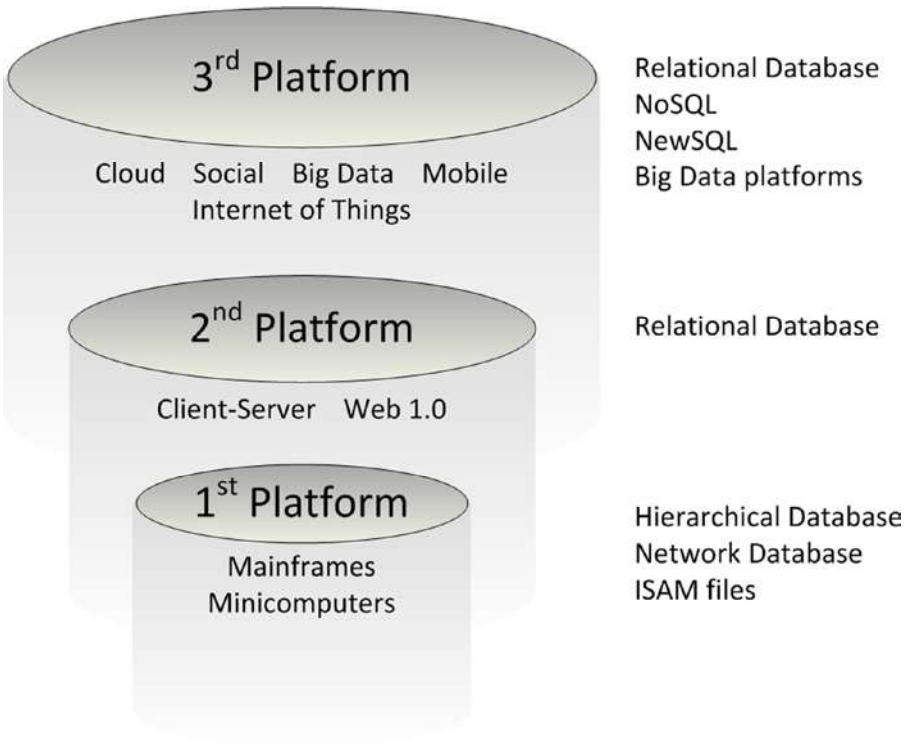

Figure 1-6. *IDC's "three platforms" model corresponds to three waves of database technology*

It's an exciting time to be working in the database industry. For a generation of software professionals (and most of my professional life), innovation in database technology occurred largely within the constraints of the ACID-compliant relational databases. Now that the hegemony of the RDBMS has been broken, we are free to design database systems whose only constraint is our imagination. It's well known that failure drives innovation. Some of these new database system concepts might not survive the test of time; however, there seems little chance that a single model will dominate the immediate future as completely as had the relational model. Database professionals will need to choose the most appropriate technology for their circumstances with care; in many cases, relational technology will continue be the best fit—but not always.

In the following chapters, we'll look at each of the major categories of next-generation database systems. We'll examine their ambitions, their architectures, and their ability to meet the challenges posed by modern application systems.

Notes

1. http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf
2. William Kent, "A Simple Guide to Five Normal Forms in Relational Database Theory," 1983.
3. http://research.microsoft.com/en-us/um/people/gray/papers/theTransactionConcept.pdf
4. https://www.cs.cmu.edu/~clamen/OODBMS/Manifesto/

5. http://research.google.com/archive/gfs.html
6. http://research.google.com/archive/mapreduce.html
7. http://research.google.com/archive/bigtable.html
8. http://queue.acm.org/detail.cfm?id=1466448
9. http://nms.csail.mit.edu/~stavros/pubs/hstore.pdf

CHAPTER 2

Google, Big Data, and Hadoop

Information is the oil of the 21st century, and analytics is the combustion engine.

—Peter Sondergaard, Gartner Research, 2011

Data creation is exploding. With all the selfies and useless files people refuse to delete on the cloud. . . . The world's data storage capacity will be overtaken. . . . Data shortages, data rationing, data black markets . . . data-geddon!

—Gavin Belson, HBOs Silicon Valley, 2015

In the history of computing, nothing has raised the profile of data processing, storage, and analytics as much as the concept of *Big Data*. We have considered ourselves to be an information-age society since the 1980s, but the concentration of media and popular attention to the role of data in our society have never been greater than in the past few years–thanks to Big Data.

Big Data technologies include those that allow us to derive more meaning from data–*machine learning,* for instance—and those that permit us to store greater volumes of data at higher granularity than ever before.

Google pioneered many of these Big Data technologies, and they found their way into the broader IT community in the form of *Hadoop*. In this chapter we review the history of Google's data management technologies, the emergence of Hadoop, and the development of other technologies for massive unstructured data storage.

The Big Data Revolution

Big Data is a broad term with multiple, competing definitions. For this author, it suggests two complementary significant shifts in the role of data within computer science and society:

- **More data**: We now have the ability to store and process *all* data—machine generated, multimedia, social networking, and transactional data—in its original raw format and maintain this data potentially in perpetuity.
- **To more effect**: Advances in machine learning, predictive analytics, and collective intelligence allow more value to be generated from data than ever before.

This is not a book about the Big Data revolution; there are enough of those already. However, we should spend a few pages articulating the significance of Big Data as a concept so as to put these technologies into context.

Big Data often seems like a meaningless buzz phrase to older database professionals who have been experiencing exponential growth in database volumes since time immemorial. There has never been a moment in the history of database management systems when the increasing volume of data has not been remarkable.

However, it is true that the nature of organizational data today is qualitatively different from that of the recent past. Some have referred to this paradigm shift as an "industrial revolution" in data, and indeed this term seems apt. Before the industrial revolution, all products were created essentially by hand, whereas after the industrial revolution, products were created in assembly lines that were in factories. In a similar way, before the industrial revolution of data, all data was generated "in house." Now, data comes in from everywhere: customers, social networks, and sensors, as well as all the traditional sources, such as sales and internal operational systems.

Cloud, Mobile, Social, and Big Data

Most would agree that the three leading information technology trends over the last decade have been in cloud, mobile, and social media. These three megatrends have transformed our economy, our society, and our everyday lives. The evolution of online retail over the past 15 years provides perhaps the most familiar example of these trends in motion.

The term *cloud computing* started to gain mindshare in 2008, but what we now call "The Cloud" was truly born in the e-commerce revolution of the late 1990s. The emergence of the Internet as a universal wide area network and the World Wide Web as a business-to-customer portal drove virtually all businesses "into the cloud" via the creation of web-based storefronts.

For some industries—music and books, for instance—the Internet rapidly became a significant or even dominant sales channel. But across the wider retail landscape, physical storefronts (brick-and-mortar stores) continued to dominate consumer interactions. Although retailers were fully represented in the cloud, consumers had only a sporadic and shallow presence. For most consumers, Internet connectivity was limited to a shared home computer or a desktop at work. The consumer was only intermittently connected to the Internet and had no online identity.

The emergence of *social networks* and *smartphones* occurred almost simultaneously. Smartphones allowed people to be online at all times, while social networks provided the motivation for frequent Internet engagement. Within a few years, the average consumer's Internet interactions accelerated: people who had previously interacted with the Internet just a few times a day were now continually online, monitoring their professional and social engagements through email and social networks.

Consumers now found it convenient to shop online whenever the impulse arose. Furthermore, retailers quickly discovered that they could leverage information from social networks and other Internet sources to target marketing campaigns and personalize engagement. Retailers themselves created social network presences that complemented their online stores and allowed the retailer to engage the consumer directly through the social network.

The synergy between the online business and the social network—mediated and enabled by the always connected mobile Internet—has resulted in a seismic shift in the retail landscape. The key to the effectiveness of the new retail model is data: the social-mobile-cloud generates masses of data that can be used to refine and enhance the online experience. This positive feedback loop drives the Big Data solutions that can represent success or failure for the next generation of retail operations.

Retail is a familiar example, but similar dynamics drive almost every other industry. In some cases the *Internet of Things* (*IoT*)—by hooking virtually every physical device that collects or consumes data into the Internet—plays the equivalent role as the smartphone in the retail context. New connected devices—Internet-enabled cars, wearable devices, home automation, and so on—propel the virtuous data cycle that generates and depends on data to drive competitive advantage. Figure 2-1 illustrates this cycle.

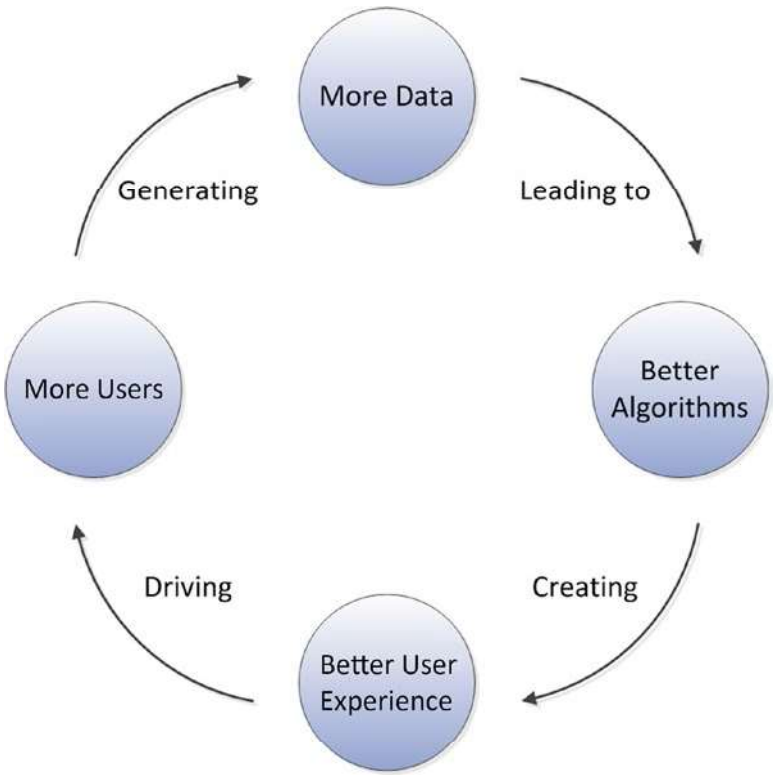

Figure 2-1. *The virtuous cycle of big data*

Google: Pioneer of Big Data

When Google was first created in 1996, the World Wide Web was already a network of unparalleled scale; indeed, it was that very large scale that made Google's key innovation—*PageRank*—such a breakthrough. Existing search engines simply indexed on keywords within webpages. This was inadequate, given the sheer number of possible matches for any search term; the results were primarily weighted by the number of occurrences of the search term within a page, with no account for usefulness or popularity. PageRank allowed the relevance of a page to be weighted based on the number of links to that page, and it allowed Google to immediately provide a better search outcome than its competitors.

PageRank is a great example of a data-driven algorithm that leverages the "wisdom of the crowd" (*collective intelligence*) and that can adapt intelligently as more data is available (*machine learning*). Google is, therefore, the first clear example of a company that succeeded on the web through what we now call *data science*.

Google Hardware

Google serves as a prime example of how better algorithm were forged by intelligently harnessing massive datasets. However, for our purposes, it is Google's innovations in data architecture that are most relevant.

Google's initial hardware platform consisted of the typical collection of off-the-shelf servers sitting under a desk that might have been found in any research lab. However, it took only a few years for Google to shift to masses of rack-mounted servers in commercial-grade data centers. As Google grew, it outstripped

the capacity limits of even the most massive existing data center architectures. The economics and practicalities of delivering a hardware infrastructure capable of unbounded exponential growth required Google to create a new hardware and software architecture.

Google committed to a number of key tenants when designing its data center architecture. Most significantly—and at the time, uniquely—Google committed to massively parallelizing and distributing processing across very large numbers of commodity servers. Google also adopted a "Jedis build their own lightsabers" attitude: very little third party— and virtually no commercial—software would be found in the Google architecture. "Build" was considered better than "buy" at Google.

By 2005, Google no longer thought of individual servers as the fundamental unit of computing. Rather, Google constructed data centers around the *Google Modular Data Center*. The Modular Data Center comprises shipping containers that house about a thousand custom-designed Intel servers running Linux. Each module includes an independent power supply and air conditioning. Data center capacity is increased not by adding new servers individually but by adding new 1,000-server modules! Figure 2-2 shows a diagram of a module as described in Google's patent.[1]

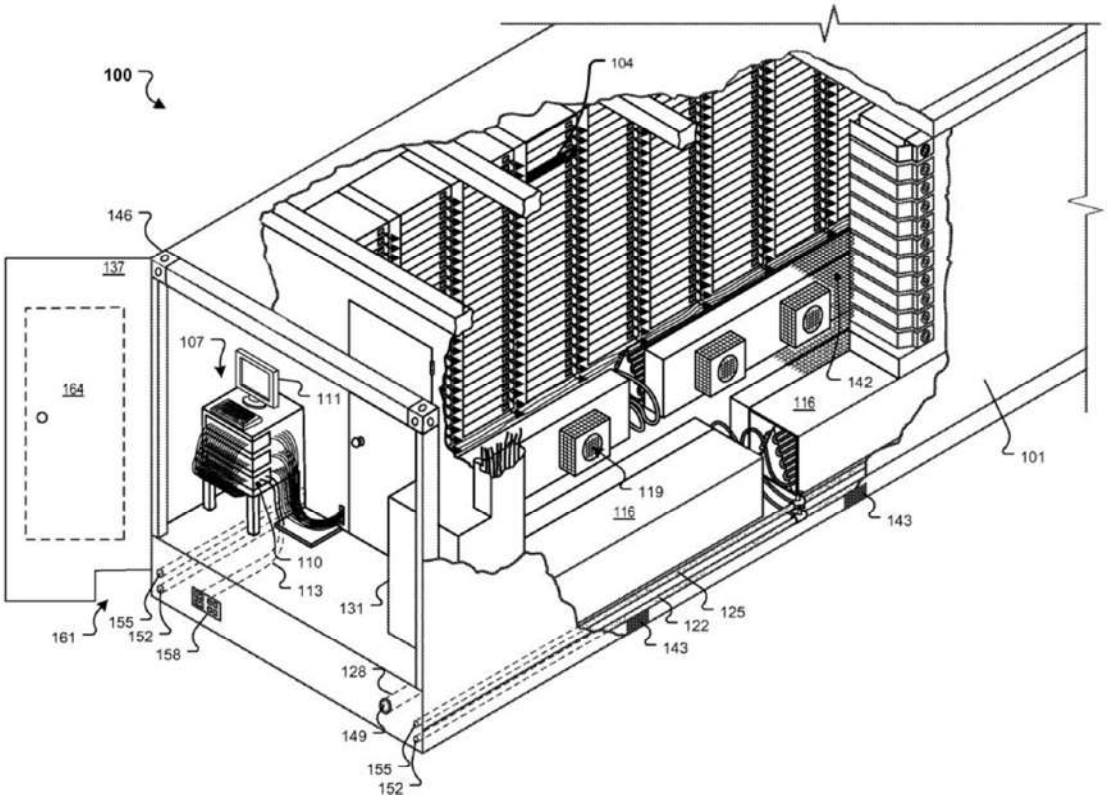

Figure 2-2. *Google Modular Data Center as described in their patent*

At this time, the prevailing architecture for data processing was to separate its storage to dedicated storage servers built by companies such as EMC. This storage would be made available to databases or other applications via Fibre Channel (or similar protocol) as a *storage area network* (*SAN*) or across TCP/IP as *network attached storage* (*NAS*). Google rejected these concepts; in the Google architecture, storage would be on directly attached disks within the same servers as would be providing computing power.

The Google Software Stack

There are many fascinating aspects to Google's hardware architecture. However, it's enough for our purposes to understand that the Google architecture at that time comprised hundreds of thousands of low-cost servers, each of which had its own directly attached storage.

It goes without saying that this unique hardware architecture required a unique software architecture as well. No operating system or database platform available at the time could come close to operating across such a huge number of servers. So, Google developed three major software layers to serve as the foundation for the Google platform. These were:

- **Google File System (GFS)**: a distributed cluster file system that allows all of the disks within the Google data center to be accessed as one massive, distributed, redundant file system.
- **MapReduce**: a distributed processing framework for parallelizing algorithms across large numbers of potentially unreliable servers and being capable of dealing with massive datasets.
- **BigTable**: a nonrelational database system that uses the Google File System for storage.

Google was generous enough to reveal the essential designs for each of these components in a series of papers released in 2003,[2] 2004,[3] and 2006.[4] These three technologies—together with other utilities and components—served as the foundation for many Google products.

A high level, very simplified representation of the architecture is shown in Figure 2-3. GFS abstracts the storage contained in the servers that make up Google's modular data centers. MapReduce abstracts the processing power contained within these servers, while BigTable allows for structured storage of massive datasets using GFS for storage.

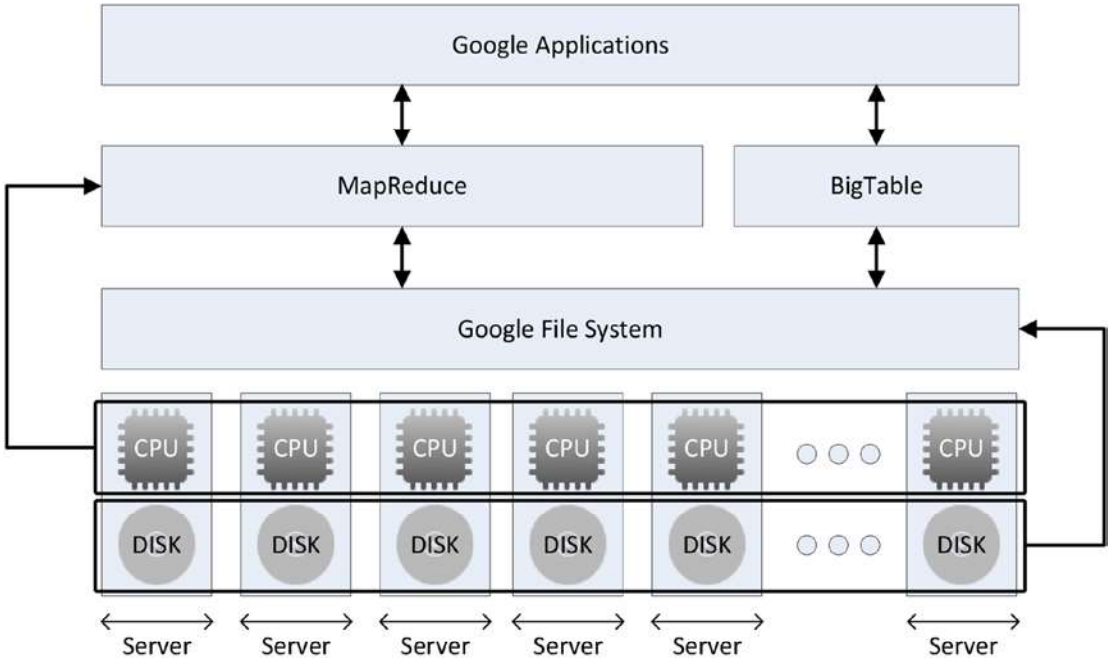

Figure 2-3. *Google software architecture*

More about MapReduce

MapReduce is a programming model for general-purpose parallelization of data-intensive processing. MapReduce divides the processing into two phases: a *mapping phase,* in which data is broken up into chunks that can be processed by separate threads—potentially running on separate machines; and a *reduce phase,* which combines the output from the mappers into the final result.

The canonical example of MapReduce is the word-count program, shown in Figure 2-4. For example, suppose we wish to count the occurrences of pet types in some input file. We break the data into equal chunks in the map phase. The data is then shuffled into groups of pet types. Finally, the reduce phase counts the occurrences to provide a total that is fed into the output.

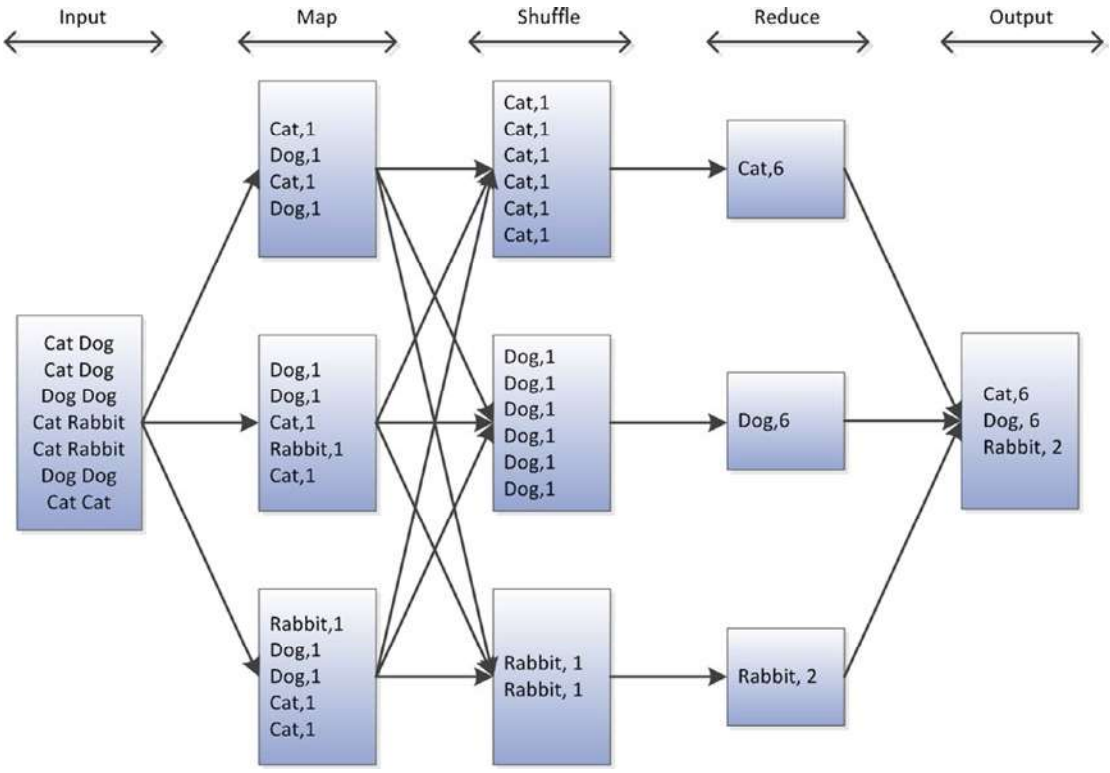

Figure 2-4. *Simple MapReduce pipeline*

Simple MapReduce pipelines as shown in Figure 2-4 are rare; it's far more typical that multiple MapReduce phases are chained together to achieve more complex results. For instance, there might be multiple input files that need to be merged in some way, or there may be some complex iterative processing to perform a statistical or machine learning analysis.

Figure 2-5 illustrates a more complex multistage MapReduce pipeline. In this example, a file containing information about visits to various product webpages is joined with a file containing product details—to obtain the product category—and then to a file containing customer details, so as to determine the customer's country. This joined data is then aggregated to provide a report of product categories, customer geographies, and page visits.

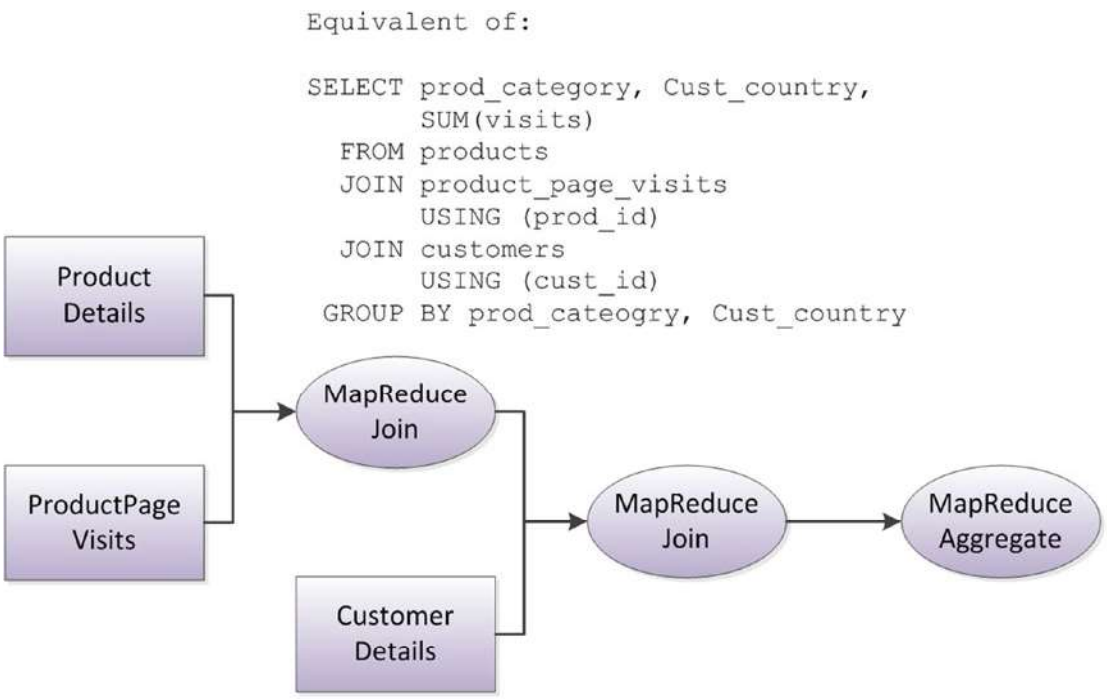

Figure 2-5. *Multistage MapReduce*

MapReduce processes can be assembled into arbitrarily complex pipelines capable of solving a very wide range of data processing problems. However, in many respects, MapReduce represents a brute-force approach to processing, and is not always the most efficient or elegant solution. There also exist a category of computational problems for which MapReduce cannot offer scalable solutions. For all these reasons, MapReduce has been extended within and without Google by more sophisticated and specialized algorithms; we will look at some of these in Chapter 11. However, despite the increasing prevalence of alternative processing models, MapReduce remains a default and widely applicable paradigm.

Hadoop: Open-Source Google Stack

While the technologies outlined in the previous section underlie Google products that we have all used, they are not generally directly exposed to the public. However, they did serve as the inspiration for *Hadoop,* which contains analogs for each of these important technologies and which is available to all as an open-source Apache project.

No other single technology has had as great an influence on Big Data as Hadoop. Hadoop—by providing an economically viable means of mass unstructured data storage and processing— has bought Google-style Big Data processing within the reach of mainstream IT.

Hadoop's Origins

In 2004, Doug Cutting and Mike Cafarella were working on an open-source project to build a web search engine called Nutch, which would be on top of Apache Lucene. Lucene is a Java-based text indexing and searching library. The Nutch team aspired to use Lucene to build a scalable web search engine with the explicit intent of creating an open-source equivalent of the proprietary technologies used within Google.

Initial versions of Nutch could not demonstrate the scalability required to index the entire web. When Google published the GFS and MapReduce papers in 2003 and 2004, the Nutch team quickly realized that these offered a proven architectural foundation for solving Nutch's scalability challenges.

As the Nutch team implemented their GFS and MapReduce equivalents, it soon became obvious that these were applicable to a broad range of data processing challenges and that a dedicated project to exploit the technology was warranted. The resulting project was named Hadoop. Hadoop was made available for download in 2007, and became a top-level Apache project in 2008.

Many open-source projects start small and grow slowly. Hadoop was the exact opposite. Yahoo! hired Cutting to work on improving Hadoop in 2006, with the objective of maturing Hadoop to the point that it could contribute to the Yahoo! platform. In early 2008, Yahoo! announced that a Hadoop cluster with over 5 petabytes of storage and more than 10,000 cores was generating the index that resolved Yahoo!'s web searches. As a result, before almost anybody outside the IT community had heard of Hadoop, it had already been proven on a massive scale.

The other significant early adopter of Hadoop was Facebook. Facebook started experimenting with Hadoop in 2007, and by 2008 it had a cluster utilizing 2,500 CPU cores in production. Facebook's initial implementation of Hadoop supplemented its Oracle-based data warehouse. By 2012, Facebook`s Hadoop cluster had exceeded 100 petabytes of disk, and it had completely overtaken Oracle as a data warehousing solution, as well as powering many core Facebook products.

Hadoop has been adopted—at least in pilot form—by many of the Fortune 500 companies. As with all new technologies, some overhyping and subsequent disillusionment is to be expected. However, most organizations that are actively pursuing a Big Data solution use Hadoop in some form.

The degree to which Hadoop has become the de facto solution for massive unstructured data storage and processing can be illustrated by the positions taken by the top three database vendors: Microsoft, Oracle, and IBM. By 2012, each of these giants had ceased to offer any form of Hadoop alternative and instead were offering Hadoop within its product portfolio.

The Power of Hadoop

Hadoop provides an economically attractive storage solution for Big Data, as well as a scalable processing model for analytic processing. Specifically, it has:

- **An economical scalable storage model**. As data volumes increase, so does the cost of storing that data online. Because Hadoop can run on commodity hardware that in turn utilizes commodity disks, the price point per terabyte is lower than that of almost any other technology.
- **Massive scaleable IO capability**. Because Hadoop uses a large number of commodity devices, the aggregate IO and network capacity is higher than that provided by dedicated storage arrays in which smaller numbers of larger disks are provided by even smaller numbers of processors. Furthermore, adding new servers to Hadoop adds storage, IO, CPU, and network capacity all at once, whereas adding disks to a storage array might simply exacerbate a network or CPU bottleneck within the array.

- **Reliability**: Data in Hadoop is stored redundantly in multiple servers and can be distributed across multiple computer racks. Failure of a server does not result in a loss of data; in fact, a Hadoop job will continue even if a server fails—the processing simply switches to another server.
- **A scalable processing model**: MapReduce represents a widely applicable and scalable distributed processing model. While MapReduce is not the most efficient implementation for all algorithms, it is capable of brute-forcing acceptable performance for almost all.
- **Schema on read**: Data can be loaded into Hadoop without having to be converted to a highly structured normalized format. This makes it easy for Hadoop to quickly ingest data from various forms. The imposition of structure can be delayed until the data is accessed; this is sometimes referred to as *schema on read*, as opposed to the *schema on write* mode of relational data warehouses.

Hadoop's Architecture

Hadoop's architecture roughly parallels that of Google. Google File System capabilities are provided by the *Hadoop Distributed File System* (*HDFS*), which allows all the disk storage in the cluster to be accessed using familiar file system idioms.

There are currently two major iterations of Hadoop architecture. Hadoop 2.0 layers on top of the 1.0 architecture, so let's consider each in turn.

In Hadoop 1.0, the majority of servers in a Hadoop cluster function both as *data nodes* and as *task trackers*, which is to say that each server supplies both data storage and processing capacity (CPU and memory).

Specialized nodes within the Hadoop 1.0 architecture are also defined. The *job tracker* node coordinates the scheduling of jobs run on the Hadoop cluster, while the *name node* is a sort of directory that provides the mapping from blocks on data nodes to files on HDFS. Every piece of data will usually be replicated across three nodes, which can be located on separate server racks to avoid any single point of failure. Figure 2-6 illustrates the Hadoop 1.0 architecture.

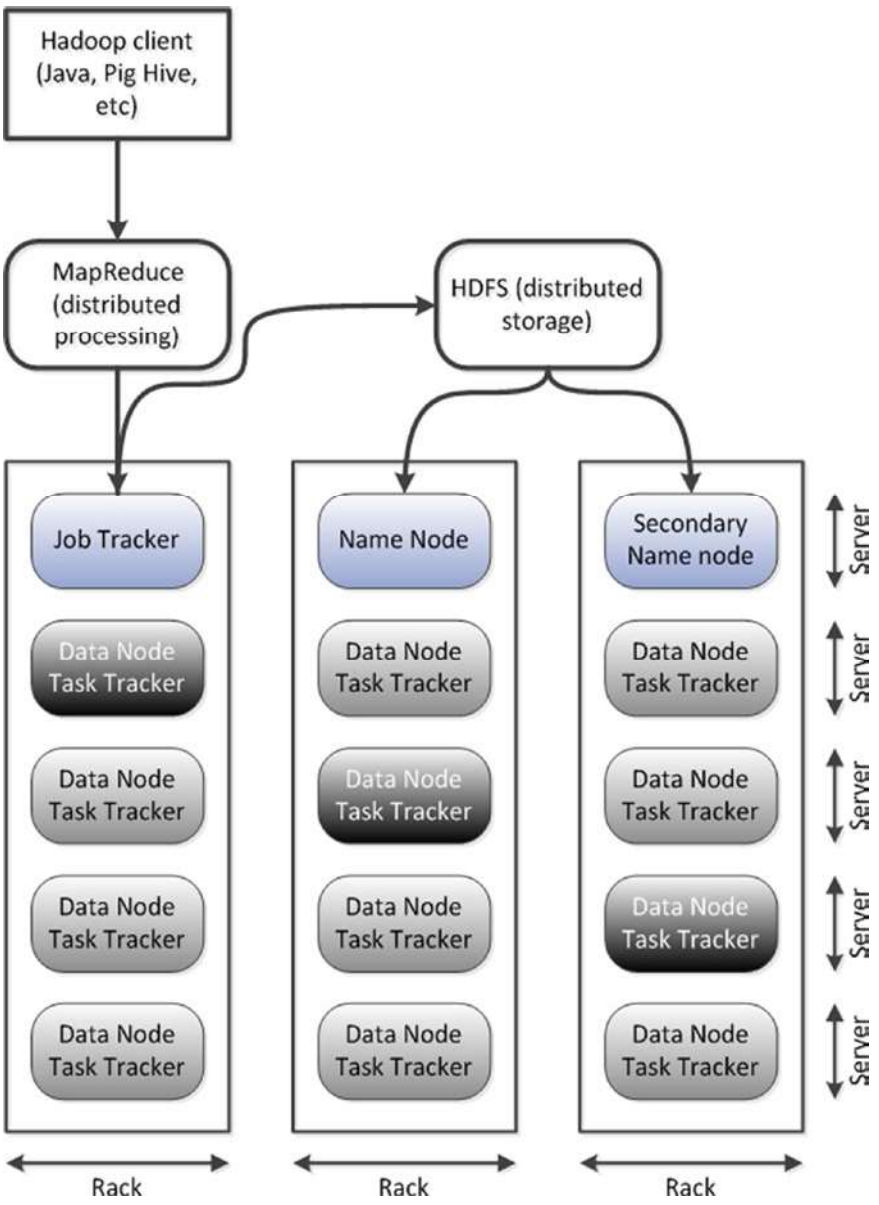

Figure 2-6. *Hadoop 1.0 architecture*

The Hadoop 1.0 architecture is powerful and easy to understand, but it is limited to MapReduce workloads and it provides limited flexibility with regard to scheduling and resource allocation. In the Hadoop 2.0 architecture, *YARN* (*Yet Another Resource Negotiator* or, recursively, *YARN Application Resource Negotiator*) improves scalability and flexibility by splitting the roles of the Task Tracker into two processes. A *Resource Manager* controls access to the clusters resources (memory, CPU, etc.) while the *Application Manager* (one per job) controls task execution.

YARN provides much more than just improved scalability. YARN treats traditional MapReduce as just one of the possible frameworks that can run on the cluster, allowing Hadoop to run tasks based on more complex processing models, some of which we'll discuss in Chapter 11.

Figure 2-7 illustrates the resource allocation and application execution aspects of YARN. For example, a Hadoop client submits an application execution request to the Resource Manager (1). The Resource Manager coordinates with the various Node Managers to determine which nodes have available resource (2). The Resource Manager then creates an Application Manager (3) on an available node. The Application Manager coordinates tasks that run in Containers on the selected nodes (4). The Containers control the amount of CPU and memory resource the application task may use.

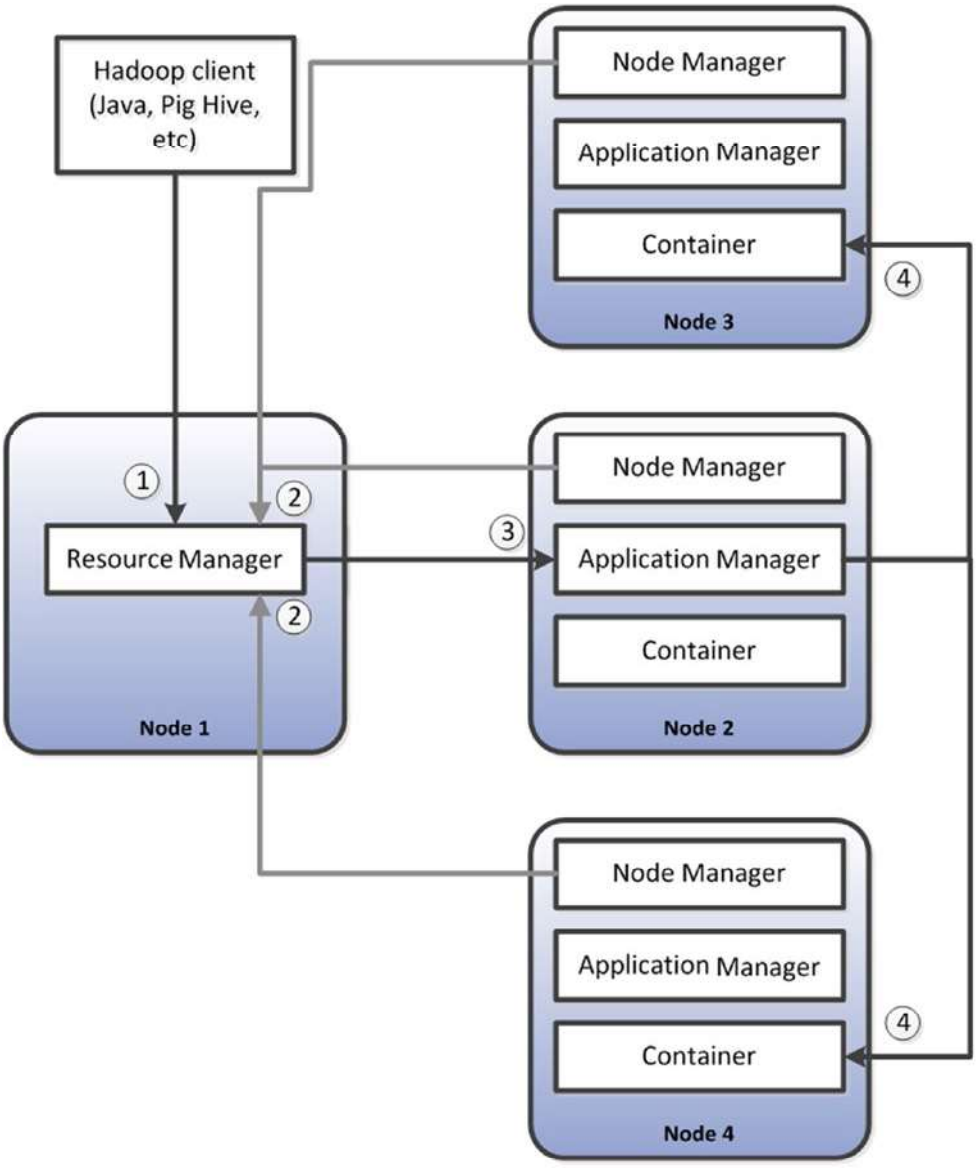

Figure 2-7. *Hadoop 2.0 YARN architecture*

HBase

As mentioned earlier in the chapter, Google published three key papers revealing the architecture of their platform between 2003 and 2006. The GFS and MapReduce papers served as the basis for the core Hadoop architecture. The third paper—on BigTable—served as the basis for one of the first formal NoSQL database systems: *HBase*.

HBase uses Hadoop HDFS as a file system in the same way that most traditional relational databases use the operating system file system. For instance, in a MySQL database using the MyISAM option, each table is represented as a file stored on the file system. By using Hadoop HDFS as its file system, HBase is able to create tables of truly massive size—way beyond the possible size for a system like MySQL, or even for Oracle. In addition, the fault tolerance of HDFS provides automatic redundancy for HBase tables. As we saw in Figure 2-6, each data item in an HDFS file system is replicated (by default) three times. Since HDFS provides this inherent redundancy, HBase need not store multiple copies of data to protect against data loss.

While HDFS allows a file of any structure to be stored within Hadoop, HBase does enforce structure on the data. The terminology of HBase objects seem pretty familiar—columns, rows, tables, keys. However, HBase tables vary significantly from the relational tables with which we are familiar.

First, in each cell—a column value for a particular row—there will usually be multiple versions of a data value. Each version of data within a cell is identified by a timestamp. This provides HBase tables with a sort of temporal "third dimension."

Second, HBase columns are more like the key values in a distributed Map of key : value pairs than the fixed and relatively small number of columns found in a relational database table. Each row can have a huge number of "sparse" columns. Each row in an HBase table can appear to consist of a unique set of columns.

To get a sense of how the HBase data model works, consider the data shown in Figure 2-8. First we see the raw data (1)—a list of users, websites, and the number of times each user visited the web site. The relational representation (2) involves three tables—sites, people, and visits—with foreign key relationships between sites and visits and people and visits.

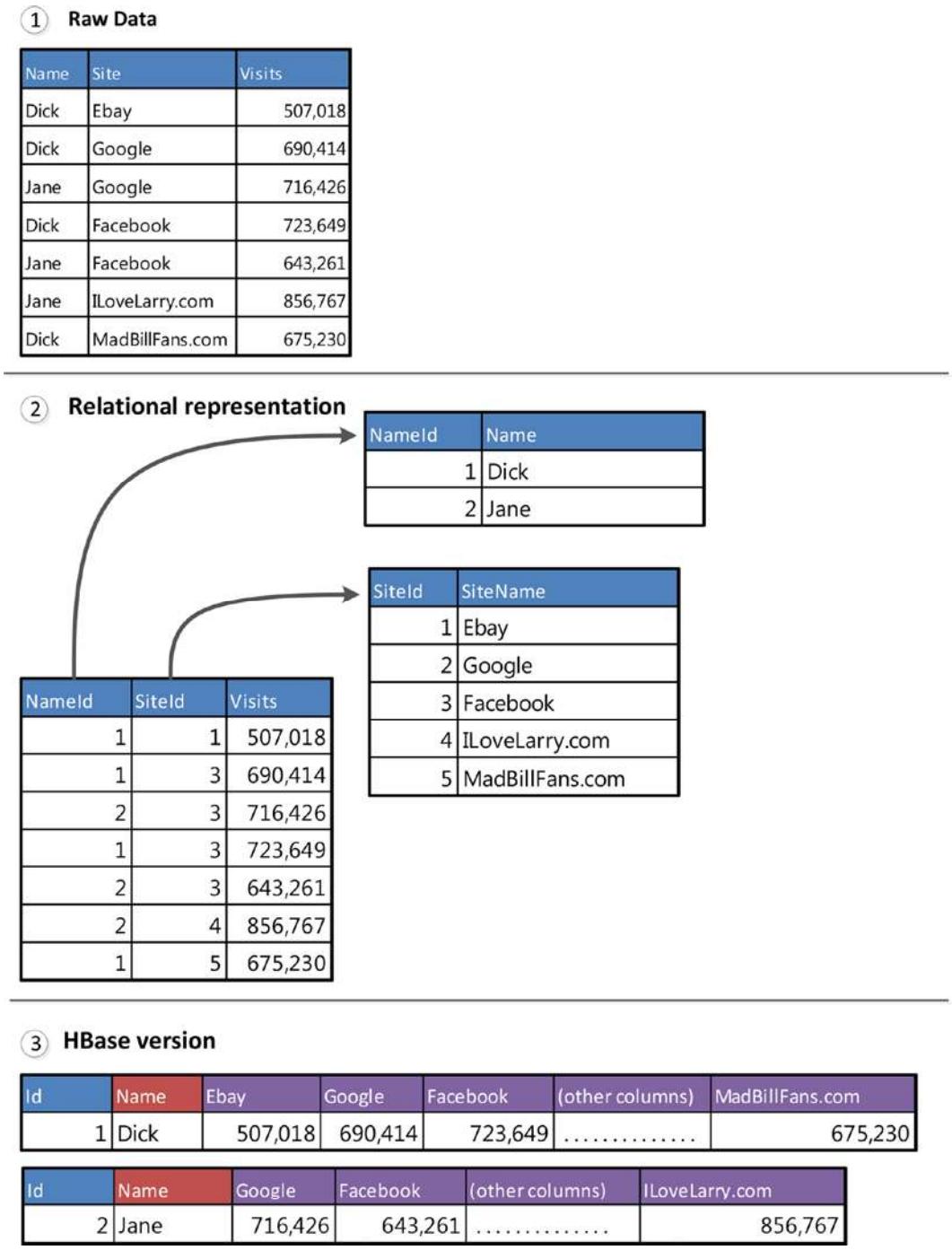

Figure 2-8. *HBase data model compared to relational model*

In the HBase representation (3), each person's information is held in a single row. That row contains columns for every website visited by that person. The column name represents the site name and the column value represents the number of visits. Because people visit thousands to hundreds of thousands of sites, there can be potentially thousands or hundreds of thousands of columns in a row. And while there are some sites that almost everyone visits—Google.com, for instance—rows only have columns corresponding to websites that they actually visited. So for instance, if you have never visited dell.com, then there will be no dell.com column in your row.

The HBase data model and storage system is examined in more detail in Chapter 10.

Hive

The pioneers of Hadoop realized fairly early that the full value of the platform could not be realized if only people capable of coding MapReduce programs could access the system. Non-programmers needed flexible, powerful, and accessible query tools to extract data from the Hadoop system. Even for programmers, the laborious and tedious process of coding MapReduce to perform repetitive reporting tasks seemed terribly inefficient. Two solutions to this problem were independently developed at Facebook and Yahoo!: *Hive* and *Pig*, respectively.

Hive is usually thought of as "SQL for Hadoop," although Hive provides a catalog for the Hadoop system, as well as a SQL processing layer. The Hive metadata service contains information about the structure of registered files in the HDFS file system. This metadata effectively "schematizes" these files, providing definitions of column names and data types. The Hive client or server (depending on the Hive configuration) accepts SQL-like commands called *Hive Query Language* (*HQL*). These commands are translated into Hadoop jobs that process the query and return the results to the user. Most of the time, Hive creates MapReduce programs that implement query operations such as joins, sorts, aggregation, and so on. However, recent versions of Hive can employ more modern YARN-based processing paradigms such as *Tez,* a programming model designed to speed up operations of certain data processing patterns; we'll look more at Tez in Chapter 11.

Figure 2-9 illustrates the Hive architecture. The Hive metastore maps HDFS files to Hive tables (1). A Hive client or server (depending on the installation mode) accepts HQL commands that perform SQL operations on those tables. Hive translates HQL to Hadoop code (3)—usually MapReduce. This code operates against the HDFS files (4) and returns query results to Hive (5).

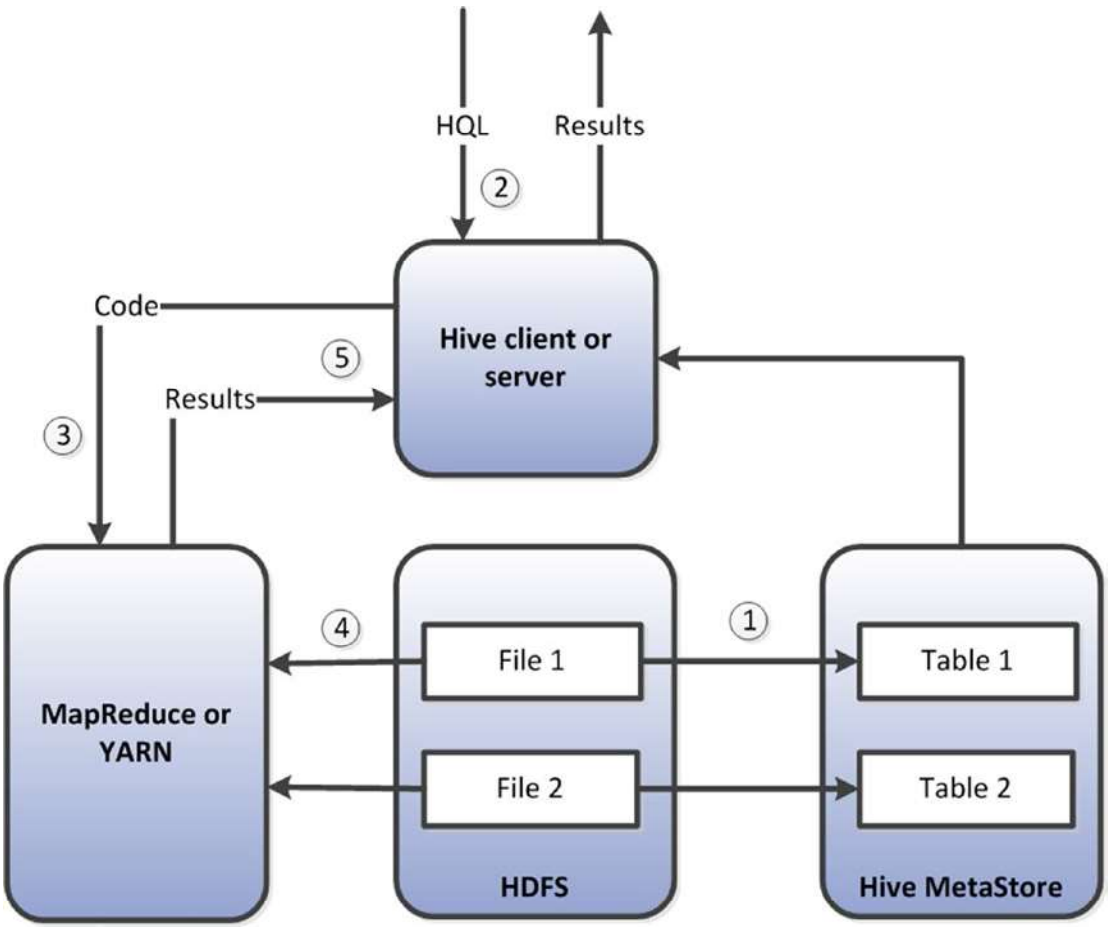

Figure 2-9. *Hive architecture*

It's hard to overstate the importance of Hive. Hive opened up Hadoop to anybody familiar with SQL, illustrated to the community at large that Hadoop could operate as a form of data warehouse, and set the stage for integration of Hadoop into Business Intelligence tools. However, by raising expectations that Hadoop could operate as a traditional database, Hive also contributed to some unrealistic expectations. SQL is generally used as a real-time query tool, but Hadoop's batch orientation means that even the simplest HQL query cannot run in a real-time mode.

The Hadoop community has attempted to deal with Hive performance issues in two ways. The predominant Hadoop vendor Cloudera has created an alternative proprietary SQL on the Hadoop framework called *Impala*, while others—including another major Hadoop vendor Hortonworks—have attempted to improve Hive's performance through incremental changes and better integration with post-MapReduce frameworks such as YARN and Tez. Meanwhile, traditional database vendors such as Oracle and Teradata have attempted to provide SQL on Hadoop functionality through their existing SQL engines. We'll spend some more time on SQL interfaces to Hadoop and NoSQL in Chapter 11.

Pig

Facebook created Hive to empower analysts wanting to access data in Hadoop. Yahoo!, in response to similar demands, independently created another solution: *Pig*.

Pig supports a procedural, high-level data flow language called *Pig Latin*. Like Hive, Pig Latin is compiled to MapReduce code. However, Pig is more of a scripting language than a SQL alternative. While it is possible to create Pig equivalents to virtually all Hive HQL queries, Pig is capable of expressing far more complex pipelines of operations than is HQL.

Figure 2-10 compares a Pig Latin script with an equivalent Hive HQL statement. Note that the Pig script is a procedural representation—it explicitly specifies the sequence of events that must be undertaken in order to achieve the result. Like SQL, HQL is nonprocedural: it's up to the Hive optimizer to determine the means of execution; the HQL only specifies the logical operations to be performed on the data.

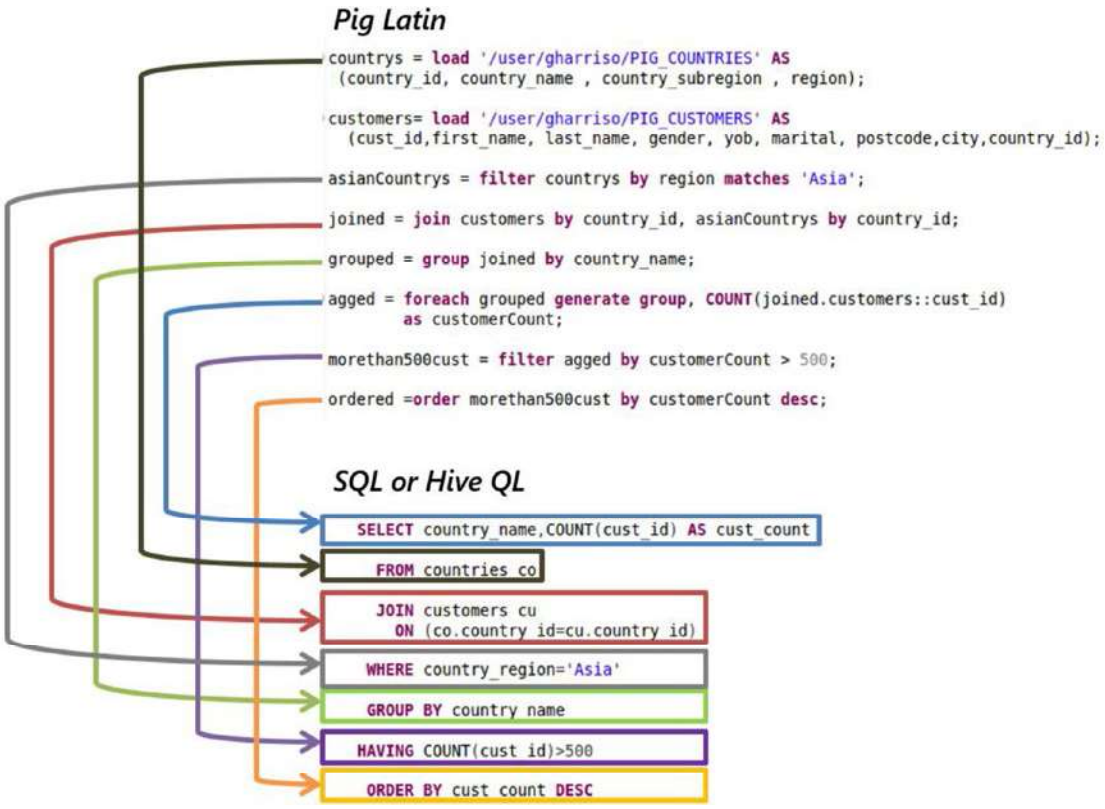

Figure 2-10. *Pig Latin as compared with Hive HQL*

The Hadoop Ecosystem

MapReduce, YARN, and HDFS represent the foundations of the Hadoop architecture. HBase, Pig, and Hive are built on top of those foundations. The Hadoop ecosystem includes an ever expanding family of utilities and applications built on top of or designed to work with core Hadoop. Some of the most significant are:

- **Flume**, a utility for loading file-based data into HDFS.
- **SQOOP**, a utility for exchanging data with relational databases, either by importing relational tables into HDFS files or by exporting HDFS files to relational databases.
- **Zookeeper**, which provides coordination and synchronization services within the cluster.
- **Oozie**, a workflow scheduler that allows complex workflows to be constructed from lower level jobs (for instance, running a Sqoop or Flume job prior to a MapReduce application).
- **Hue**, a graphical user interface that simplifies Hadoop administrative and development tasks.

In addition, there are many Apache and open-source projects that, while not entirely dependent on Hadoop, are often integrated into a Hadoop implementation. This includes the machine-learning framework *Mahout*, the distributed streaming messaging system *Kafka,* and many other significant Apache projects.

The most important addition to the Hadoop family in recent years has been *Spark* and other elements of the *Berkeley Data Analytics Stack* (*BDAS*) to which it belongs. If we think about Hadoop as a disk-oriented framework for running MapReduce-style programs, Spark represents a memory-oriented framework for running similar workloads. We'll look at Spark in more detail in Chapter 7.

Conclusion

Hadoop represents one of the most significant transformations in database architecture since the relational model. It provides economies of storage and processing that are beyond the reach of the traditional RDBMS, and it offers an ability for the storage and processing of unstructured and semi-structured data for which the RDBMS had no real solution. More than any other technology, Hadoop has rightfully been associated with the Big Data movement.

However, while Hadoop provides a framework for the mass processing of data, it doesn't have a framework for transactional and online operations. As we will see in the next chapter, bleeding-edge websites demanded not just a storage and batch processing solution but also an online transactional solution. These demands led to what we now call NoSQL, and that will be examined in the next chapter.

Notes

1. http://www.google.com/patents/US20100251629
2. http://research.google.com/arcHive/gfs.html
3. http://research.google.com/arcHive/mapreduce.html
4. http://research.google.com/arcHive/bigtable.html