part 10

Distributed Databases, NOSQL Systems, and Big Data

This page intentionally left blank

chapter 23

Distributed Database Concepts

In this chapter, we turn our attention to distributed databases (DDBs), distributed database management systems (DDBMSs), and how the client-server architecture is used as a platform for database application development. Distributed databases bring the advantages of distributed computing to the database domain. A **distributed computing system** consists of a number of processing sites or nodes that are interconnected by a computer network and that cooperate in performing certain assigned tasks. As a general goal, distributed computing systems partition a big, unmanageable problem into smaller pieces and solve it efficiently in a coordinated manner. Thus, more computing power is harnessed to solve a complex task, and the autonomous processing nodes can be managed independently while they cooperate to provide the needed functionalities to solve the problem. DDB technology resulted from a merger of two technologies: database technology and distributed systems technology.

Several distributed database prototype systems were developed in the 1980s and 1990s to address the issues of data distribution, data replication, distributed query and transaction processing, distributed database metadata management, and other topics. More recently, many new technologies have emerged that combine distributed and database technologies. These technologies and systems are being developed for dealing with the storage, analysis, and mining of the vast amounts of data that are being produced and collected, and they are referred to generally as **big data technologies**. The origins of big data technologies come from distributed systems and database systems, as well as data mining and machine learning algorithms that can process these vast amounts of data to extract needed knowledge.

In this chapter, we discuss the concepts that are central to data distribution and the management of distributed data. Then in the following two chapters, we give an overview of some of the new technologies that have emerged to manage and process big data. Chapter 24 discusses the new class of database systems known as NOSQL

systems, which focus on providing distributed solutions to manage the vast amounts of data that are needed in applications such as social media, healthcare, and security, to name a few. Chapter 25 introduces the concepts and systems being used for processing and analysis of big data, such as map-reduce and other distributed processing technologies. We also discuss cloud computing concepts in Chapter 25.

Section 23.1 introduces distributed database management and related concepts. Issues of distributed database design, involving fragmenting and sharding of data and distributing it over multiple sites, as well as data replication, are discussed in Section 23.2. Section 23.3 gives an overview of concurrency control and recovery in distributed databases. Sections 23.4 and 23.5 introduce distributed transaction processing and distributed query processing techniques, respectively. Sections 23.6 and 23.7 introduce different types of distributed database systems and their architectures, including federated and multidatabase systems. The problems of heterogeneity and the needs of autonomy in federated database systems are also highlighted. Section 23.8 discusses catalog management schemes in distributed databases. Section 23.9 summarizes the chapter.

For a short introduction to the topic of distributed databases, Sections 23.1 through 23.5 may be covered and the other sections may be omitted.

23.1 Distributed Database Concepts

We can define a **distributed database (DDB)** as a collection of multiple logically interrelated databases distributed over a computer network, and a **distributed database management system (DDBMS)** as a software system that manages a distributed database while making the distribution transparent to the user.

23.1.1 What Constitutes a DDB

For a database to be called distributed, the following minimum conditions should be satisfied:

- **Connection of database nodes over a computer network.** There are multiple computers, called **sites** or **nodes**. These sites must be connected by an underlying **network** to transmit data and commands among sites.
- **Logical interrelation of the connected databases.** It is essential that the information in the various database nodes be logically related.
- **Possible absence of homogeneity among connected nodes.** It is not necessary that all nodes be identical in terms of data, hardware, and software.

The sites may all be located in physical proximity—say, within the same building or a group of adjacent buildings—and connected via a **local area network**, or they may be geographically distributed over large distances and connected via a **long-haul** or **wide area network**. Local area networks typically use wireless hubs or cables, whereas long-haul networks use telephone lines, cables, wireless communication infrastructures, or satellites. It is common to have a combination of various types of networks.

Networks may have different **topologies** that define the direct communication paths among sites. The type and topology of the network used may have a significant impact on the performance and hence on the strategies for distributed query processing and distributed database design. For high-level architectural issues, however, it does not matter what type of network is used; what matters is that each site be able to communicate, directly or indirectly, with every other site. For the remainder of this chapter, we assume that some type of network exists among nodes, regardless of any particular topology. We will not address any network-specific issues, although it is important to understand that for an efficient operation of a distributed database system (DDBS), network design and performance issues are critical and are an integral part of the overall solution. The details of the underlying network are invisible to the end user.

23.1.2 Transparency

The concept of transparency extends the general idea of hiding implementation details from end users. A highly transparent system offers a lot of flexibility to the end user/application developer since it requires little or no awareness of underlying details on their part. In the case of a traditional centralized database, transparency simply pertains to logical and physical data independence for application developers. However, in a DDB scenario, the data and software are distributed over multiple nodes connected by a computer network, so additional types of transparencies are introduced.

Consider the company database in Figure 5.5 that we have been discussing throughout the book. The EMPLOYEE, PROJECT, and WORKS_ON tables may be fragmented horizontally (that is, into sets of rows, as we will discuss in Section 23.2) and stored with possible replication, as shown in Figure 23.1. The following types of transparencies are possible:

- **Data organization transparency (also known as *distribution* or *network transparency*).** This refers to freedom for the user from the operational details of the network and the placement of the data in the distributed system. It may be divided into location transparency and naming transparency. **Location transparency** refers to the fact that the command used to perform a task is independent of the location of the data and the location of the node where the command was issued. **Naming transparency** implies that once a name is associated with an object, the named objects can be accessed unambiguously without additional specification as to where the data is located.
- **Replication transparency.** As we show in Figure 23.1, copies of the same data objects may be stored at multiple sites for better availability, performance, and reliability. Replication transparency makes the user unaware of the existence of these copies.
- **Fragmentation transparency.** Two types of fragmentation are possible. **Horizontal fragmentation** distributes a relation (table) into subrelations that are subsets of the tuples (rows) in the original relation; this is also known

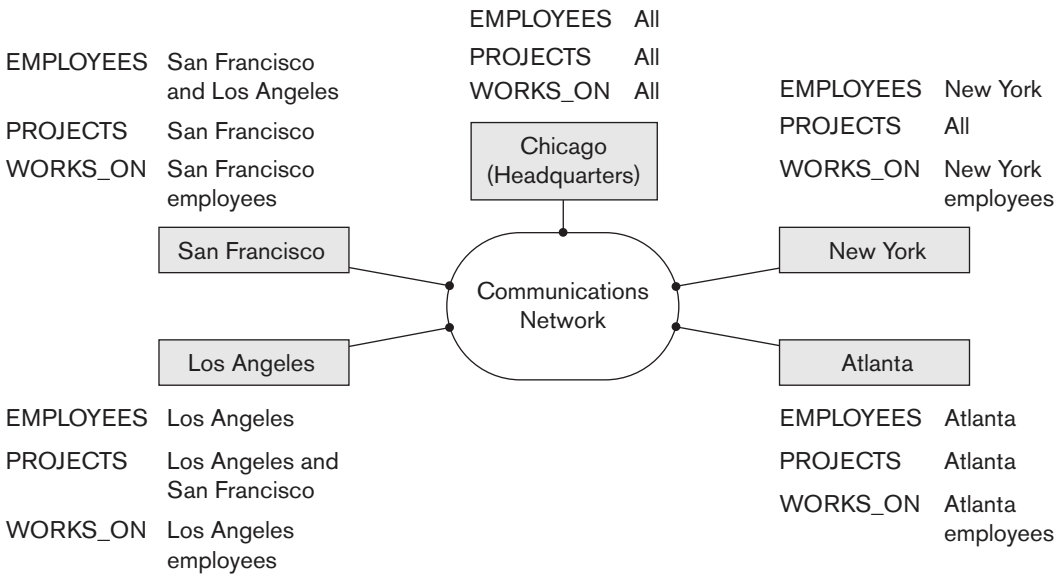

Figure 23.1
Data distribution and replication among distributed databases.

as **sharding** in the newer big data and cloud computing systems. **Vertical fragmentation** distributes a relation into subrelations where each subrelation is defined by a subset of the columns of the original relation. Fragmentation transparency makes the user unaware of the existence of fragments.

- Other transparencies include **design transparency** and **execution transparency**—which refer, respectively, to freedom from knowing how the distributed database is designed and where a transaction executes.

23.1.3 Availability and Reliability

Reliability and availability are two of the most common potential advantages cited for distributed databases. **Reliability** is broadly defined as the probability that a system is running (not down) at a certain time point, whereas **availability** is the probability that the system is continuously available during a time interval. We can directly relate reliability and availability of the database to the faults, errors, and failures associated with it. A failure can be described as a deviation of a system's behavior from that which is specified in order to ensure correct execution of operations. **Errors** constitute that subset of system states that causes the failure. **Fault** is the cause of an error.

To construct a system that is reliable, we can adopt several approaches. One common approach stresses *fault tolerance*; it recognizes that faults will occur, and it designs mechanisms that can detect and remove faults before they can result in a

system failure. Another more stringent approach attempts to ensure that the final system does not contain any faults. This is done through an exhaustive design process followed by extensive quality control and testing. A reliable DDBMS tolerates failures of underlying components, and it processes user requests as long as database consistency is not violated. A DDBMS recovery manager has to deal with failures arising from transactions, hardware, and communication networks. Hardware failures can either be those that result in loss of main memory contents or loss of secondary storage contents. Network failures occur due to errors associated with messages and line failures. Message errors can include their loss, corruption, or out-of-order arrival at destination.

The previous definitions are used in computer systems in general, where there is a technical distinction between reliability and availability. In most discussions related to DDB, the term **availability** is used generally as an umbrella term to cover both concepts.

23.1.4 Scalability and Partition Tolerance

Scalability determines the extent to which the system can expand its capacity while continuing to operate without interruption. There are two types of scalability:

1. **Horizontal scalability:** This refers to expanding the number of nodes in the distributed system. As nodes are added to the system, it should be possible to distribute some of the data and processing loads from existing nodes to the new nodes.
2. **Vertical scalability:** This refers to expanding the capacity of the individual nodes in the system, such as expanding the storage capacity or the processing power of a node.

As the system expands its number of nodes, it is possible that the network, which connects the nodes, may have faults that cause the nodes to be partitioned into groups of nodes. The nodes within each partition are still connected by a subnetwork, but communication among the partitions is lost. The concept of **partition tolerance** states that the system should have the capacity to continue operating while the network is partitioned.

23.1.5 Autonomy

Autonomy determines the extent to which individual nodes or DBs in a connected DDB can operate independently. A high degree of autonomy is desirable for increased flexibility and customized maintenance of an individual node. Autonomy can be applied to design, communication, and execution. **Design autonomy** refers to independence of data model usage and transaction management techniques among nodes. **Communication autonomy** determines the extent to which each node can decide on sharing of information with other nodes. **Execution autonomy** refers to independence of users to act as they please.

23.1.6 Advantages of Distributed Databases

Some important advantages of DDB are listed below.

1. **Improved ease and flexibility of application development**. Developing and maintaining applications at geographically distributed sites of an organization is facilitated due to transparency of data distribution and control.
2. **Increased availability.** This is achieved by the isolation of faults to their site of origin without affecting the other database nodes connected to the network. When the data and DDBMS software are distributed over many sites, one site may fail while other sites continue to operate. Only the data and software that exist at the failed site cannot be accessed. Further improvement is achieved by judiciously replicating data and software at more than one site. In a centralized system, failure at a single site makes the whole system unavailable to all users. In a distributed database, some of the data may be unreachable, but users may still be able to access other parts of the database. If the data in the failed site has been replicated at another site prior to the failure, then the user will not be affected at all. The ability of the system to survive network partitioning also contributes to high availability.
3. **Improved performance.** A distributed DBMS fragments the database by keeping the data closer to where it is needed most. **Data localization** reduces the contention for CPU and I/O services and simultaneously reduces access delays involved in wide area networks. When a large database is distributed over multiple sites, smaller databases exist at each site. As a result, local queries and transactions accessing data at a single site have better performance because of the smaller local databases. In addition, each site has a smaller number of transactions executing than if all transactions are submitted to a single centralized database. Moreover, interquery and intraquery parallelism can be achieved by executing multiple queries at different sites, or by breaking up a query into a number of subqueries that execute in parallel. This contributes to improved performance.
4. **Easier expansion via scalability**. In a distributed environment, expansion of the system in terms of adding more data, increasing database sizes, or adding more nodes is much easier than in centralized (non-distributed) systems.

The transparencies we discussed in Section 23.1.2 lead to a compromise between ease of use and the overhead cost of providing transparency. Total transparency provides the global user with a view of the entire DDBS as if it is a single centralized system. Transparency is provided as a complement to **autonomy**, which gives the users tighter control over local databases. Transparency features may be implemented as a part of the user language, which may translate the required services into appropriate operations.

23.2 Data Fragmentation, Replication, and Allocation Techniques for Distributed Database Design

In this section, we discuss techniques that are used to break up the database into logical units, called **fragments**, which may be assigned for storage at the various nodes. We also discuss the use of **data replication**, which permits certain data to be stored in more than one site to increase availability and reliability; and the process of **allocating** fragments—or replicas of fragments—for storage at the various nodes. These techniques are used during the process of **distributed database design**. The information concerning data fragmentation, allocation, and replication is stored in a **global directory** that is accessed by the DDBS applications as needed.

23.2.1 Data Fragmentation and Sharding

In a DDB, decisions must be made regarding which site should be used to store which portions of the database. For now, we will assume that there is *no replication*; that is, each relation—or portion of a relation—is stored at one site only. We discuss replication and its effects later in this section. We also use the terminology of relational databases, but similar concepts apply to other data models. We assume that we are starting with a relational database schema and must decide on how to distribute the relations over the various sites. To illustrate our discussion, we use the relational database schema shown in Figure 5.5.

Before we decide on how to distribute the data, we must determine the *logical units* of the database that are to be distributed. The simplest logical units are the relations themselves; that is, each *whole* relation is to be stored at a particular site. In our example, we must decide on a site to store each of the relations EMPLOYEE, DEPARTMENT, PROJECT, WORKS_ON, and DEPENDENT in Figure 5.5. In many cases, however, a relation can be divided into smaller logical units for distribution. For example, consider the company database shown in Figure 5.6, and assume there are three computer sites—one for each department in the company.[1]

We may want to store the database information relating to each department at the computer site for that department. A technique called *horizontal fragmentation* or *sharding* can be used to partition each relation by department.

Horizontal Fragmentation (Sharding). A **horizontal fragment** or **shard** of a relation is a subset of the tuples in that relation. The tuples that belong to the horizontal fragment can be specified by a condition on one or more attributes of the relation, or by some other mechanism. Often, only a single attribute is involved in the condition. For example, we may define three horizontal fragments on the EMPLOYEE relation in Figure 5.6 with the following conditions: (Dno = 5), (Dno = 4), and (Dno = 1)—each

[1]Of course, in an actual situation, there will be many more tuples in the relation than those shown in Figure 5.6.

fragment contains the EMPLOYEE tuples working for a particular department. Similarly, we may define three horizontal fragments for the PROJECT relation, with the conditions (Dnum = 5), (Dnum = 4), and (Dnum = 1)—each fragment contains the PROJECT tuples controlled by a particular department. **Horizontal fragmentation** divides a relation *horizontally* by grouping rows to create subsets of tuples, where each subset has a certain logical meaning. These fragments can then be assigned to different sites (nodes) in the distributed system. **Derived horizontal fragmentation** applies the partitioning of a primary relation (DEPARTMENT in our example) to other secondary relations (EMPLOYEE and PROJECT in our example), which are related to the primary via a foreign key. Thus, related data between the primary and the secondary relations gets fragmented in the same way.

Vertical Fragmentation. Each site may not need all the attributes of a relation, which would indicate the need for a different type of fragmentation. **Vertical fragmentation** divides a relation "vertically" by columns. A **vertical fragment** of a relation keeps only certain attributes of the relation. For example, we may want to fragment the EMPLOYEE relation into two vertical fragments. The first fragment includes personal information—Name, Bdate, Address, and Sex—and the second includes work-related information—Ssn, Salary, Super_ssn, and Dno. This vertical fragmentation is not quite proper, because if the two fragments are stored separately, we cannot put the original employee tuples back together since there is *no common attribute* between the two fragments. It is necessary to include the primary key or some unique key attribute in *every* vertical fragment so that the full relation can be reconstructed from the fragments. Hence, we must add the Ssn attribute to the personal information fragment.

Notice that each horizontal fragment on a relation R can be specified in the relational algebra by a $\sigma_{C_i}(R)$ (select) operation. A set of horizontal fragments whose conditions $C_1, C_2, \ldots, C_n$ include all the tuples in R—that is, every tuple in R satisfies (C_1 OR C_2 OR … OR C_n)—is called a **complete horizontal fragmentation** of R. In many cases a complete horizontal fragmentation is also **disjoint**; that is, no tuple in R satisfies (C_i AND C_j) for any $i \neq j$. Our two earlier examples of horizontal fragmentation for the EMPLOYEE and PROJECT relations were both complete and disjoint. To reconstruct the relation R from a *complete* horizontal fragmentation, we need to apply the UNION operation to the fragments.

A vertical fragment on a relation R can be specified by a $\pi_{L_i}(R)$ operation in the relational algebra. A set of vertical fragments whose projection lists $L_1, L_2, \ldots, L_n$ include all the attributes in R but share only the primary key attribute of R is called a **complete vertical fragmentation** of R. In this case the projection lists satisfy the following two conditions:

- $L_1 \cup L_2 \cup \ldots \cup L_n = \text{ATTRS}(R)$
- $L_i \cap L_j = \text{PK}(R)$ for any $i \neq j$, where ATTRS(R) is the set of attributes of R and PK(R) is the primary key of R

To reconstruct the relation R from a *complete* vertical fragmentation, we apply the OUTER UNION operation to the vertical fragments (assuming no horizontal

fragmentation is used). Notice that we could also apply a FULL OUTER JOIN operation and get the same result for a complete vertical fragmentation, even when some horizontal fragmentation may also have been applied. The two vertical fragments of the EMPLOYEE relation with projection lists L_1 = {Ssn, Name, Bdate, Address, Sex} and L_2 = {Ssn, Salary, Super_ssn, Dno} constitute a complete vertical fragmentation of EMPLOYEE.

Two horizontal fragments that are neither complete nor disjoint are those defined on the EMPLOYEE relation in Figure 5.5 by the conditions (Salary > 50000) and (Dno = 4); they may not include all EMPLOYEE tuples, and they may include common tuples. Two vertical fragments that are not complete are those defined by the attribute lists L_1 = {Name, Address} and L_2 = {Ssn, Name, Salary}; these lists violate both conditions of a complete vertical fragmentation.

Mixed (Hybrid) Fragmentation. We can intermix the two types of fragmentation, yielding a **mixed fragmentation**. For example, we may combine the horizontal and vertical fragmentations of the EMPLOYEE relation given earlier into a mixed fragmentation that includes six fragments. In this case, the original relation can be reconstructed by applying UNION *and* OUTER UNION (or OUTER JOIN) operations in the appropriate order. In general, a **fragment** of a relation R can be specified by a SELECT-PROJECT combination of operations $\pi_L(\sigma_C(R))$. If C = TRUE (that is, all tuples are selected) and $L \neq$ ATTRS(R), we get a vertical fragment, and if $C \neq$ TRUE and L = ATTRS(R), we get a horizontal fragment. Finally, if $C \neq$ TRUE and $L \neq$ ATTRS(R), we get a mixed fragment. Notice that a relation can itself be considered a fragment with C = TRUE and L = ATTRS(R). In the following discussion, the term *fragment* is used to refer to a relation or to any of the preceding types of fragments.

A **fragmentation schema** of a database is a definition of a set of fragments that includes *all* attributes and tuples in the database and satisfies the condition that the whole database can be reconstructed from the fragments by applying some sequence of OUTER UNION (or OUTER JOIN) and UNION operations. It is also sometimes useful—although not necessary—to have all the fragments be disjoint except for the repetition of primary keys among vertical (or mixed) fragments. In the latter case, all replication and distribution of fragments is clearly specified at a subsequent stage, separately from fragmentation.

An **allocation schema** describes the allocation of fragments to nodes (sites) of the DDBS; hence, it is a mapping that specifies for each fragment the site(s) at which it is stored. If a fragment is stored at more than one site, it is said to be **replicated**. We discuss data replication and allocation next.

23.2.2 Data Replication and Allocation

Replication is useful in improving the availability of data. The most extreme case is replication of the *whole database* at every site in the distributed system, thus creating a **fully replicated distributed database**. This can improve availability remarkably because the system can continue to operate as long as at least one site is up. It

also improves performance of retrieval (read performance) for global queries because the results of such queries can be obtained locally from any one site; hence, a retrieval query can be processed at the local site where it is submitted, if that site includes a server module. The disadvantage of full replication is that it can slow down update operations (write performance) drastically, since a single logical update must be performed on *every copy* of the database to keep the copies consistent. This is especially true if many copies of the database exist. Full replication makes the concurrency control and recovery techniques more expensive than they would be if there was no replication, as we will see in Section 23.3.

The other extreme from full replication involves having **no replication**—that is, each fragment is stored at exactly one site. In this case, all fragments *must be* disjoint, except for the repetition of primary keys among vertical (or mixed) fragments. This is also called **nonredundant allocation**.

Between these two extremes, we have a wide spectrum of **partial replication** of the data—that is, some fragments of the database may be replicated whereas others may not. The number of copies of each fragment can range from one up to the total number of sites in the distributed system. A special case of partial replication is occurring heavily in applications where mobile workers—such as sales forces, financial planners, and claims adjustors—carry partially replicated databases with them on laptops and PDAs and synchronize them periodically with the server database. A description of the replication of fragments is sometimes called a **replication schema**.

Each fragment—or each copy of a fragment—must be assigned to a particular site in the distributed system. This process is called **data distribution** (or **data allocation**). The choice of sites and the degree of replication depend on the performance and availability goals of the system and on the types and frequencies of transactions submitted at each site. For example, if high availability is required, transactions can be submitted at any site, and most transactions are retrieval only, a fully replicated database is a good choice. However, if certain transactions that access particular parts of the database are mostly submitted at a particular site, the corresponding set of fragments can be allocated at that site only. Data that is accessed at multiple sites can be replicated at those sites. If many updates are performed, it may be useful to limit replication. Finding an optimal or even a good solution to distributed data allocation is a complex optimization problem.

23.2.3 Example of Fragmentation, Allocation, and Replication

We now consider an example of fragmenting and distributing the company database in Figures 5.5 and 5.6. Suppose that the company has three computer sites—one for each current department. Sites 2 and 3 are for departments 5 and 4, respectively. At each of these sites, we expect frequent access to the EMPLOYEE and PROJECT information for the employees *who work in that department* and the projects *controlled by that department.* Further, we assume that these sites mainly access the Name, Ssn, Salary, and Super_ssn attributes of EMPLOYEE. Site 1 is used

by company headquarters and accesses all employee and project information regularly, in addition to keeping track of DEPENDENT information for insurance purposes.

According to these requirements, the whole database in Figure 5.6 can be stored at site 1. To determine the fragments to be replicated at sites 2 and 3, first we can horizontally fragment DEPARTMENT by its key Dnumber. Then we apply derived fragmentation to the EMPLOYEE, PROJECT, and DEPT_LOCATIONS relations based on their foreign keys for department number—called Dno, Dnum, and Dnumber, respectively, in Figure 5.5. We can vertically fragment the resulting EMPLOYEE fragments to include only the attributes {Name, Ssn, Salary, Super_ssn, Dno}. Figure 23.2 shows the mixed fragments EMPD_5 and EMPD_4, which include the EMPLOYEE tuples satisfying the conditions Dno = 5 and Dno = 4, respectively. The horizontal fragments of PROJECT, DEPARTMENT, and DEPT_LOCATIONS are similarly fragmented by department number. All these fragments—stored at sites 2 and 3—are replicated because they are also stored at headquarters—site 1.

We must now fragment the WORKS_ON relation and decide which fragments of WORKS_ON to store at sites 2 and 3. We are confronted with the problem that no attribute of WORKS_ON directly indicates the department to which each tuple belongs. In fact, each tuple in WORKS_ON relates an employee *e* to a project *P*. We could fragment WORKS_ON based on the department *D* in which *e* works *or* based on the department *D′* that controls *P*. Fragmentation becomes easy if we have a constraint stating that $D = D'$ for all WORKS_ON tuples—that is, if employees can work only on projects controlled by the department they work for. However, there is no such constraint in our database in Figure 5.6. For example, the WORKS_ON tuple <333445555, 10, 10.0> relates an employee who works for department 5 with a project controlled by department 4. In this case, we could fragment WORKS_ON based on the department in which the employee works (which is expressed by the condition *C*) and then fragment further based on the department that controls the projects that employee is working on, as shown in Figure 23.3.

In Figure 23.3, the union of fragments G_1, G_2, and G_3 gives all WORKS_ON tuples for employees who work for department 5. Similarly, the union of fragments G_4, G_5, and G_6 gives all WORKS_ON tuples for employees who work for department 4. On the other hand, the union of fragments G_1, G_4, and G_7 gives all WORKS_ON tuples for projects controlled by department 5. The condition for each of the fragments G_1 through G_9 is shown in Figure 23.3. The relations that represent M:N relationships, such as WORKS_ON, often have several possible logical fragmentations. In our distribution in Figure 23.2, we choose to include all fragments that can be joined to either an EMPLOYEE tuple or a PROJECT tuple at sites 2 and 3. Hence, we place the union of fragments G_1, G_2, G_3, G_4, and G_7 at site 2 and the union of fragments G_4, G_5, G_6, G_2, and G_8 at site 3. Notice that fragments G_2 and G_4 are replicated at both sites. This allocation strategy permits the join between the local EMPLOYEE or PROJECT fragments at site 2 or site 3 and the local WORKS_ON fragment to be performed completely locally. This clearly demonstrates how complex the problem of database fragmentation and allocation is for large databases. The Selected Bibliography at the end of this chapter discusses some of the work done in this area.

(a)

EMPD_5

Fname	Minit	Lname	Ssn	Salary	Super_ssn	Dno
John	B	Smith	123456789	30000	333445555	5
Franklin	T	Wong	333445555	40000	888665555	5
Ramesh	K	Narayan	666884444	38000	333445555	5
Joyce	A	English	453453453	25000	333445555	5

DEP_5

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22

DEP_5_LOCS

Dnumber	Location
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON_5

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0

PROJS_5

Pname	Pnumber	Plocation	Dnum
Product X	1	Bellaire	5
Product Y	2	Sugarland	5
Product Z	3	Houston	5

Data at site 2

(b)

EMPD_4

Fname	Minit	Lname	Ssn	Salary	Super_ssn	Dno
Alicia	J	Zelaya	999887777	25000	987654321	4
Jennifer	S	Wallace	987654321	43000	888665555	4
Ahmad	V	Jabbar	987987987	25000	987654321	4

DEP_4

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Administration	4	987654321	1995-01-01

DEP_4_LOCS

Dnumber	Location
4	Stafford

WORKS_ON_4

Essn	Pno	Hours
333445555	10	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0

PROJS_4

Pname	Pnumber	Plocation	Dnum
Computerization	10	Stafford	4
New_benefits	30	Stafford	4

Data at site 3

Figure 23.2
Allocation of fragments to sites. (a) Relation fragments at site 2 corresponding to department 5. (b) Relation fragments at site 3 corresponding to department 4.

Figure 23.3
Complete and disjoint fragments of the WORKS_ON relation. (a) Fragments of WORKS_ON for employees working in department 5 (C = [Essn in (SELECT Ssn FROM EMPLOYEE WHERE Dno = 5)]). (b) Fragments of WORKS_ON for employees working in department 4 (C = [Essn in (SELECT Ssn FROM EMPLOYEE WHERE Dno = 4)]). (c) Fragments of WORKS_ON for employees working in department 1 (C = [Essn in (SELECT Ssn FROM EMPLOYEE WHERE Dno = 1)]).

(a) Employees in Department 5

G1

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0

C1 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 5))

G2

Essn	Pno	Hours
333445555	10	10.0

C2 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 4))

G3

Essn	Pno	Hours
333445555	20	10.0

C3 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 1))

(b) Employees in Department 4

G4

Essn	Pno	Hours

C4 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 5))

G5

Essn	Pno	Hours
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0

C5 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 4))

G6

Essn	Pno	Hours
987654321	20	15.0

C6 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 1))

(c) Employees in Department 1

G7

Essn	Pno	Hours

C7 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 5))

G8

Essn	Pno	Hours

C8 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 4))

G9

Essn	Pno	Hours
888665555	20	Null

C9 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 1))

23.3 Overview of Concurrency Control and Recovery in Distributed Databases

For concurrency control and recovery purposes, numerous problems arise in a distributed DBMS environment that are not encountered in a centralized DBMS environment. These include the following:

- **Dealing with *multiple copies* of the data items.** The concurrency control method is responsible for maintaining consistency among these copies. The recovery method is responsible for making a copy consistent with other copies if the site on which the copy is stored fails and recovers later.
- **Failure of individual sites.** The DDBMS should continue to operate with its running sites, if possible, when one or more individual sites fail. When a site recovers, its local database must be brought up-to-date with the rest of the sites before it rejoins the system.
- **Failure of communication links.** The system must be able to deal with the failure of one or more of the communication links that connect the sites. An extreme case of this problem is that **network partitioning** may occur. This breaks up the sites into two or more partitions, where the sites within each partition can communicate only with one another and not with sites in other partitions.
- **Distributed commit.** Problems can arise with committing a transaction that is accessing databases stored on multiple sites if some sites fail during the commit process. The **two-phase commit protocol** (see Section 21.6) is often used to deal with this problem.
- **Distributed deadlock.** Deadlock may occur among several sites, so techniques for dealing with deadlocks must be extended to take this into account.

Distributed concurrency control and recovery techniques must deal with these and other problems. In the following subsections, we review some of the techniques that have been suggested to deal with recovery and concurrency control in DDBMSs.

23.3.1 Distributed Concurrency Control Based on a Distinguished Copy of a Data Item

To deal with replicated data items in a distributed database, a number of concurrency control methods have been proposed that extend the concurrency control techniques that are used in centralized databases. We discuss these techniques in the context of extending centralized *locking*. Similar extensions apply to other concurrency control techniques. The idea is to designate *a particular copy* of each data item as a **distinguished copy**. The locks for this data item are associated *with the distinguished copy,* and all locking and unlocking requests are sent to the site that contains that copy.

A number of different methods are based on this idea, but they differ in their method of choosing the distinguished copies. In the **primary site technique**, all distinguished copies are kept at the same site. A modification of this approach is the primary site with a **backup site**. Another approach is the **primary copy** method, where the distinguished copies of the various data items can be stored in different sites. A site that includes a distinguished copy of a data item basically acts as the **coordinator site** for concurrency control on that item. We discuss these techniques next.

Primary Site Technique. In this method, a single **primary site** is designated to be the **coordinator site** *for all database items.* Hence, all locks are kept at that site, and all requests for locking or unlocking are sent there. This method is thus an extension of the centralized locking approach. For example, if all transactions follow the two-phase locking protocol, serializability is guaranteed. The advantage of this approach is that it is a simple extension of the centralized approach and thus is not overly complex. However, it has certain inherent disadvantages. One is that all locking requests are sent to a single site, possibly overloading that site and causing a system bottleneck. A second disadvantage is that failure of the primary site paralyzes the system, since all locking information is kept at that site. This can limit system reliability and availability.

Although all locks are accessed at the primary site, the items themselves can be accessed at any site at which they reside. For example, once a transaction obtains a Read_lock on a data item from the primary site, it can access any copy of that data item. However, once a transaction obtains a Write_lock and updates a data item, the DDBMS is responsible for updating *all copies* of the data item before releasing the lock.

Primary Site with Backup Site. This approach addresses the second disadvantage of the primary site method by designating a second site to be a **backup site**. All locking information is maintained at both the primary and the backup sites. In case of primary site failure, the backup site takes over as the primary site, and a new backup site is chosen. This simplifies the process of recovery from failure of the primary site, since the backup site takes over and processing can resume after a new backup site is chosen and the lock status information is copied to that site. It slows down the process of acquiring locks, however, because all lock requests and granting of locks must be recorded at *both the primary and the backup sites* before a response is sent to the requesting transaction. The problem of the primary and backup sites becoming overloaded with requests and slowing down the system remains undiminished.

Primary Copy Technique. This method attempts to distribute the load of lock coordination among various sites by having the distinguished copies of different data items *stored at different sites.* Failure of one site affects any transactions that are accessing locks on items whose primary copies reside at that site, but other transactions are not affected. This method can also use backup sites to enhance reliability and availability.

Choosing a New Coordinator Site in Case of Failure. Whenever a coordinator site fails in any of the preceding techniques, the sites that are still running must choose a new coordinator. In the case of the primary site approach with *no* backup site, all executing transactions must be aborted and restarted in a tedious recovery process. Part of the recovery process involves choosing a new primary site and creating a lock manager process and a record of all lock information at that site. For methods that use backup sites, transaction processing is suspended while the backup site is designated as the new primary site and a new backup site is chosen and is sent copies of all the locking information from the new primary site.

If a backup site *X* is about to become the new primary site, *X* can choose the new backup site from among the system's running sites. However, if no backup site existed, or if both the primary and the backup sites are down, a process called **election** can be used to choose the new coordinator site. In this process, any site *Y* that attempts to communicate with the coordinator site repeatedly and fails to do so can assume that the coordinator is down and can start the election process by sending a message to all running sites proposing that *Y* become the new coordinator. As soon as *Y* receives a majority of yes votes, *Y* can declare that it is the new coordinator. The election algorithm itself is complex, but this is the main idea behind the election method. The algorithm also resolves any attempt by two or more sites to become coordinator at the same time. The references in the Selected Bibliography at the end of this chapter discuss the process in detail.

23.3.2 Distributed Concurrency Control Based on Voting

The concurrency control methods for replicated items discussed earlier all use the idea of a distinguished copy that maintains the locks for that item. In the **voting method**, there is no distinguished copy; rather, a lock request is sent to all sites that includes a copy of the data item. Each copy maintains its own lock and can grant or deny the request for it. If a transaction that requests a lock is granted that lock by *a majority* of the copies, it holds the lock and informs *all copies* that it has been granted the lock. If a transaction does not receive a majority of votes granting it a lock within a certain *time-out period,* it cancels its request and informs all sites of the cancellation.

The voting method is considered a truly distributed concurrency control method, since the responsibility for a decision resides with all the sites involved. Simulation studies have shown that voting has higher message traffic among sites than do the distinguished copy methods. If the algorithm takes into account possible site failures during the voting process, it becomes extremely complex.

23.3.3 Distributed Recovery

The recovery process in distributed databases is quite involved. We give only a very brief idea of some of the issues here. In some cases it is difficult even to determine whether a site is down without exchanging numerous messages with other sites. For

example, suppose that site *X* sends a message to site *Y* and expects a response from *Y* but does not receive it. There are several possible explanations:

- The message was not delivered to *Y* because of communication failure.
- Site *Y* is down and could not respond.
- Site *Y* is running and sent a response, but the response was not delivered.

Without additional information or the sending of additional messages, it is difficult to determine what actually happened.

Another problem with distributed recovery is distributed commit. When a transaction is updating data at several sites, it cannot commit until it is sure that the effect of the transaction on *every* site cannot be lost. This means that every site must first have recorded the local effects of the transactions permanently in the local site log on disk. The two-phase commit protocol is often used to ensure the correctness of distributed commit (see Section 21.6).

23.4 Overview of Transaction Management in Distributed Databases

The global and local transaction management software modules, along with the concurrency control and recovery manager of a DDBMS, collectively guarantee the ACID properties of transactions (see Chapter 20).

An additional component called the **global transaction manager** is introduced for supporting distributed transactions. The site where the transaction originated can temporarily assume the role of global transaction manager and coordinate the execution of database operations with transaction managers across multiple sites. Transaction managers export their functionality as an interface to the application programs. The operations exported by this interface are similar to those covered in Section 20.2.1, namely BEGIN_TRANSACTION, READ or WRITE, END_TRANSACTION, COMMIT_TRANSACTION, and ROLLBACK (or ABORT). The manager stores bookkeeping information related to each transaction, such as a unique identifier, originating site, name, and so on. For READ operations, it returns a local copy if valid and available. For WRITE operations, it ensures that updates are visible across all sites containing copies (replicas) of the data item. For ABORT operations, the manager ensures that no effects of the transaction are reflected in any site of the distributed database. For COMMIT operations, it ensures that the effects of a write are persistently recorded on all databases containing copies of the data item. Atomic termination (COMMIT/ ABORT) of distributed transactions is commonly implemented using the two-phase commit protocol (see Section 22.6).

The transaction manager passes to the concurrency controller module the database operations and associated information. The controller is responsible for acquisition and release of associated locks. If the transaction requires access to a locked resource, it is blocked until the lock is acquired. Once the lock is acquired, the operation is sent to the runtime processor, which handles the actual execution of the

database operation. Once the operation is completed, locks are released and the transaction manager is updated with the result of the operation.

23.4.1 Two-Phase Commit Protocol

In Section 22.6, we described the *two-phase commit protocol* (**2PC**), which requires a **global recovery manager**, or **coordinator**, to maintain information needed for recovery, in addition to the local recovery managers and the information they maintain (log, tables). The two-phase commit protocol has certain drawbacks that led to the development of the three-phase commit protocol, which we discuss next.

23.4.2 Three-Phase Commit Protocol

The biggest drawback of 2PC is that it is a blocking protocol. Failure of the coordinator blocks all participating sites, causing them to wait until the coordinator recovers. This can cause performance degradation, especially if participants are holding locks to shared resources. Other types of problems may also occur that make the outcome of the transaction nondeterministic.

These problems are solved by the three-phase commit (3PC) protocol, which essentially divides the second commit phase into two subphases called **prepare-to-commit** and **commit**. The prepare-to-commit phase is used to communicate the result of the vote phase to all participants. If all participants vote yes, then the coordinator instructs them to move into the prepare-to-commit state. The commit subphase is identical to its two-phase counterpart. Now, if the coordinator crashes during this subphase, another participant can see the transaction through to completion. It can simply ask a crashed participant if it received a prepare-to-commit message. If it did not, then it safely assumes to abort. Thus the state of the protocol can be recovered irrespective of which participant crashes. Also, by limiting the time required for a transaction to commit or abort to a maximum time-out period, the protocol ensures that a transaction attempting to commit via 3PC releases locks on time-out.

The main idea is to limit the wait time for participants who have prepared to commit and are waiting for a global commit or abort from the coordinator. When a participant receives a precommit message, it knows that the rest of the participants have voted to commit. If a precommit message has not been received, then the participant will abort and release all locks.

23.4.3 Operating System Support for Transaction Management

The following are the main benefits of operating system (OS)-supported transaction management:

- Typically, DBMSs use their own semaphores[2] to guarantee mutually exclusive access to shared resources. Since these semaphores are implemented in

[2]Semaphores are data structures used for synchronized and exclusive access to shared resources for preventing race conditions in a parallel computing system.

user space at the level of the DBMS application software, the OS has no knowledge about them. Hence if the OS deactivates a DBMS process holding a lock, other DBMS processes wanting this locked resource get blocked. Such a situation can cause serious performance degradation. OS-level knowledge of semaphores can help eliminate such situations.

- Specialized hardware support for locking can be exploited to reduce associated costs. This can be of great importance, since locking is one of the most common DBMS operations.
- Providing a set of common transaction support operations though the kernel allows application developers to focus on adding new features to their products as opposed to reimplementing the common functionality for each application. For example, if different DDBMSs are to coexist on the same machine and they chose the two-phase commit protocol, then it is more beneficial to have this protocol implemented as part of the kernel so that the DDBMS developers can focus more on adding new features to their products.

23.5 Query Processing and Optimization in Distributed Databases

Now we give an overview of how a DDBMS processes and optimizes a query. First we discuss the steps involved in query processing and then elaborate on the communication costs of processing a distributed query. Then we discuss a special operation, called a *semijoin,* which is used to optimize some types of queries in a DDBMS. A detailed discussion about optimization algorithms is beyond the scope of this text. We attempt to illustrate optimization principles using suitable examples.[3]

23.5.1 Distributed Query Processing

A distributed database query is processed in stages as follows:

1. **Query Mapping.** The input query on distributed data is specified formally using a query language. It is then translated into an algebraic query on global relations. This translation is done by referring to the global conceptual schema and does not take into account the actual distribution and replication of data. Hence, this translation is largely identical to the one performed in a centralized DBMS. It is first normalized, analyzed for semantic errors, simplified, and finally restructured into an algebraic query.
2. **Localization.** In a distributed database, fragmentation results in relations being stored in separate sites, with some fragments possibly being replicated. This stage maps the distributed query on the global schema to separate queries on individual fragments using data distribution and replication information.

[3]For a detailed discussion of optimization algorithms, see Ozsu and Valduriez (1999).

3. **Global Query Optimization.** Optimization consists of selecting a strategy from a list of candidates that is closest to optimal. A list of candidate queries can be obtained by permuting the ordering of operations within a fragment query generated by the previous stage. Time is the preferred unit for measuring cost. The total cost is a weighted combination of costs such as CPU cost, I/O costs, and communication costs. Since DDBs are connected by a network, often the communication costs over the network are the most significant. This is especially true when the sites are connected through a wide area network (WAN).
4. **Local Query Optimization.** This stage is common to all sites in the DDB. The techniques are similar to those used in centralized systems.

The first three stages discussed above are performed at a central control site, whereas the last stage is performed locally.

23.5.2 Data Transfer Costs of Distributed Query Processing

We discussed the issues involved in processing and optimizing a query in a centralized DBMS in Chapter 19. In a distributed system, several additional factors further complicate query processing. The first is the cost of transferring data over the network. This data includes intermediate files that are transferred to other sites for further processing, as well as the final result files that may have to be transferred to the site where the query result is needed. Although these costs may not be very high if the sites are connected via a high-performance local area network, they become significant in other types of networks. Hence, DDBMS query optimization algorithms consider the goal of reducing the *amount of data transfer* as an optimization criterion in choosing a distributed query execution strategy.

We illustrate this with two simple sample queries. Suppose that the EMPLOYEE and DEPARTMENT relations in Figure 3.5 are distributed at two sites as shown in Figure 23.4. We will assume in this example that neither relation is fragmented. According to Figure 23.4, the size of the EMPLOYEE relation is $100 * 10{,}000 = 10^6$ bytes, and the size of the DEPARTMENT relation is $35 * 100 = 3{,}500$ bytes. Consider the query Q: *For each employee, retrieve the employee name and the name of the department for which the employee works.* This can be stated as follows in the relational algebra:

$$Q: \pi_{Fname,Lname,Dname}(EMPLOYEE \bowtie_{Dno=Dnumber} DEPARTMENT)$$

The result of this query will include 10,000 records, assuming that every employee is related to a department. Suppose that each record in the query result is *40 bytes long*. The query is submitted at a distinct site 3, which is called the **result site** because the query result is needed there. Neither the EMPLOYEE nor the DEPARTMENT relations reside at site 3. There are three simple strategies for executing this distributed query:

1. Transfer both the EMPLOYEE and the DEPARTMENT relations to the result site, and perform the join at site 3. In this case, a total of 1,000,000 + 3,500 = 1,003,500 bytes must be transferred.

Site 1:

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno

10,000 records
each record is 100 bytes long
Ssn field is 9 bytes long
Fname field is 15 bytes long
Dno field is 4 bytes long
Lname field is 15 bytes long

Site 2:

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date

100 records
each record is 35 bytes long
Dnumber field is 4 bytes long
Dname field is 10 bytes long
Mgr_ssn field is 9 bytes long

Figure 23.4
Example to illustrate volume of data transferred.

2. Transfer the EMPLOYEE relation to site 2, execute the join at site 2, and send the result to site 3. The size of the query result is 40 * 10,000 = 400,000 bytes, so 400,000 + 1,000,000 = 1,400,000 bytes must be transferred.
3. Transfer the DEPARTMENT relation to site 1, execute the join at site 1, and send the result to site 3. In this case, 400,000 + 3,500 = 403,500 bytes must be transferred.

If minimizing the amount of data transfer is our optimization criterion, we should choose strategy 3. Now consider another query Q′: *For each department, retrieve the department name and the name of the department manager.* This can be stated as follows in the relational algebra:

$$Q': \pi_{Fname,Lname,Dname}(DEPARTMENT \bowtie_{Mgr\_ssn=Ssn} EMPLOYEE)$$

Again, suppose that the query is submitted at site 3. The same three strategies for executing query Q apply to Q′, except that the result of Q′ includes only 100 records, assuming that each department has a manager:

1. Transfer both the EMPLOYEE and the DEPARTMENT relations to the result site, and perform the join at site 3. In this case, a total of 1,000,000 + 3,500 = 1,003,500 bytes must be transferred.
2. Transfer the EMPLOYEE relation to site 2, execute the join at site 2, and send the result to site 3. The size of the query result is 40 * 100 = 4,000 bytes, so 4,000 + 1,000,000 = 1,004,000 bytes must be transferred.
3. Transfer the DEPARTMENT relation to site 1, execute the join at site 1, and send the result to site 3. In this case, 4,000 + 3,500 = 7,500 bytes must be transferred.

Again, we would choose strategy 3—this time by an overwhelming margin over strategies 1 and 2. The preceding three strategies are the most obvious ones for the

case where the result site (site 3) is different from all the sites that contain files involved in the query (sites 1 and 2). However, suppose that the result site is site 2; then we have two simple strategies:

1. Transfer the EMPLOYEE relation to site 2, execute the query, and present the result to the user at site 2. Here, the same number of bytes—1,000,000—must be transferred for both Q and Q′.
2. Transfer the DEPARTMENT relation to site 1, execute the query at site 1, and send the result back to site 2. In this case 400,000 + 3,500 = 403,500 bytes must be transferred for Q and 4,000 + 3,500 = 7,500 bytes for Q′.

A more complex strategy, which sometimes works better than these simple strategies, uses an operation called **semijoin**. We introduce this operation and discuss distributed execution using semijoins next.

23.5.3 Distributed Query Processing Using Semijoin

The idea behind distributed query processing using the *semijoin operation* is to reduce the number of tuples in a relation before transferring it to another site. Intuitively, the idea is to send the *joining column* of one relation *R* to the site where the other relation *S* is located; this column is then joined with *S*. Following that, the join attributes, along with the attributes required in the result, are projected out and shipped back to the original site and joined with *R*. Hence, only the joining column of *R* is transferred in one direction, and a subset of *S* with no extraneous tuples or attributes is transferred in the other direction. If only a small fraction of the tuples in *S* participate in the join, this can be an efficient solution to minimizing data transfer.

To illustrate this, consider the following strategy for executing Q or Q′:

1. Project the join attributes of DEPARTMENT at site 2, and transfer them to site 1. For Q, we transfer $F = \pi_{\text{Dnumber}}(\text{DEPARTMENT})$, whose size is $4 * 100 = 400$ bytes, whereas for Q′, we transfer $F' = \pi_{\text{Mgr\_ssn}}(\text{DEPARTMENT})$, whose size is $9 * 100 = 900$ bytes.
2. Join the transferred file with the EMPLOYEE relation at site 1, and transfer the required attributes from the resulting file to site 2. For Q, we transfer $R = \pi_{\text{Dno, Fname, Lname}}(F \ltimes_{\text{Dnumber=Dno}} \text{EMPLOYEE})$, whose size is $34 * 10{,}000 = 340{,}000$ bytes, whereas for Q′, we transfer $R' = \pi_{\text{Mgr\_ssn, Fname, Lname}}(F' \ltimes_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE})$, whose size is $39 * 100 = 3{,}900$ bytes.
3. Execute the query by joining the transferred file *R* or *R*′ with DEPARTMENT, and present the result to the user at site 2.

Using this strategy, we transfer 340,400 bytes for Q and 4,800 bytes for Q′. We limited the EMPLOYEE attributes and tuples transmitted to site 2 in step 2 to only those that will *actually be joined* with a DEPARTMENT tuple in step 3. For query Q, this turned out to include all EMPLOYEE tuples, so little improvement was achieved. However, for Q′ only 100 out of the 10,000 EMPLOYEE tuples were needed.

The semijoin operation was devised to formalize this strategy. A **semijoin operation** $R \ltimes_{A=B} S$, where A and B are domain-compatible attributes of R and S, respectively, produces the same result as the relational algebra expression $\pi R(R \ltimes_{A=B} S)$. In a distributed environment where R and S reside at different sites, the semijoin is typically implemented by first transferring $F = \pi_B(S)$ to the site where R resides and then joining F with R, thus leading to the strategy discussed here.

Notice that the semijoin operation is not commutative; that is,

$$R \ltimes S \neq S \ltimes R$$

23.5.4 Query and Update Decomposition

In a DDBMS with *no distribution transparency,* the user phrases a query directly in terms of specific fragments. For example, consider another query Q: *Retrieve the names and hours per week for each employee who works on some project controlled by department 5*, which is specified on the distributed database where the relations at sites 2 and 3 are shown in Figure 23.2, and those at site 1 are shown in Figure 5.6, as in our earlier example. A user who submits such a query must specify whether it references the PROJS_5 and WORKS_ON_5 relations at site 2 (Figure 23.2) or the PROJECT and WORKS_ON relations at site 1 (Figure 5.6). The user must also maintain consistency of replicated data items when updating a DDBMS with *no replication transparency.*

On the other hand, a DDBMS that supports *full distribution, fragmentation,* and *replication transparency* allows the user to specify a query or update request on the schema in Figure 5.5 just as though the DBMS were centralized. For updates, the DDBMS is responsible for maintaining *consistency among replicated items* by using one of the distributed concurrency control algorithms discussed in Section 23.3. For queries, a **query decomposition** module must break up or **decompose** a query into **subqueries** that can be executed at the individual sites. Additionally, a strategy for combining the results of the subqueries to form the query result must be generated. Whenever the DDBMS determines that an item referenced in the query is replicated, it must choose or **materialize** a particular replica during query execution.

To determine which replicas include the data items referenced in a query, the DDBMS refers to the fragmentation, replication, and distribution information stored in the DDBMS catalog. For vertical fragmentation, the attribute list for each fragment is kept in the catalog. For horizontal fragmentation, a condition, sometimes called a **guard**, is kept for each fragment. This is basically a selection condition that specifies which tuples exist in the fragment; it is called a guard because *only tuples that satisfy this condition* are permitted to be stored in the fragment. For mixed fragments, both the attribute list and the guard condition are kept in the catalog.

In our earlier example, the guard conditions for fragments at site 1 (Figure 5.6) are TRUE (all tuples), and the attribute lists are $*$ (all attributes). For the fragments

(a) EMPD5
attribute list: Fname, Minit, Lname, Ssn, Salary, Super_ssn, Dno
guard condition: Dno = 5
DEP5
attribute list: * (all attributes Dname, Dnumber, Mgr_ssn, Mgr_start_date)
guard condition: Dnumber = 5
DEP5_LOCS
attribute list: * (all attributes Dnumber, Location)
guard condition: Dnumber = 5
PROJS5
attribute list: * (all attributes Pname, Pnumber, Plocation, Dnum)
guard condition: Dnum = 5
WORKS_ON5
attribute list: * (all attributes Essn, Pno,Hours)
guard condition: Essn IN (π_{Ssn} (EMPD5)) OR Pno IN ($\pi_{Pnumber}$ (PROJS5))

(b) EMPD4
attribute list: Fname, Minit, Lname, Ssn, Salary, Super_ssn, Dno
guard condition: Dno = 4
DEP4
attribute list: * (all attributes Dname, Dnumber, Mgr_ssn, Mgr_start_date)
guard condition: Dnumber = 4
DEP4_LOCS
attribute list: * (all attributes Dnumber, Location)
guard condition: Dnumber = 4
PROJS4
attribute list: * (all attributes Pname, Pnumber, Plocation, Dnum)
guard condition: Dnum = 4
WORKS_ON4
attribute list: * (all attributes Essn, Pno, Hours)
guard condition: Essn IN (π_{Ssn} (EMPD4))
OR Pno IN ($\pi_{Pnumber}$ (PROJS4))

Figure 23.5
Guard conditions and attributes lists for fragments.
(a) Site 2 fragments.
(b) Site 3 fragments.

shown in Figure 23.2, we have the guard conditions and attribute lists shown in Figure 23.5. When the DDBMS decomposes an update request, it can determine which fragments must be updated by examining their guard conditions. For example, a user request to insert a new EMPLOYEE tuple <'Alex', 'B', 'Coleman', '345671239', '22-APR-64', '3306 Sandstone, Houston, TX', M, 33000, '987654321', 4> would be decomposed by the DDBMS into two insert requests: the first inserts the preceding tuple in the EMPLOYEE fragment at site 1, and the second inserts the projected tuple <'Alex', 'B', 'Coleman', '345671239', 33000, '987654321', 4> in the EMPD4 fragment at site 3.

For query decomposition, the DDBMS can determine which fragments may contain the required tuples by comparing the query condition with the guard conditions. For

example, consider the query Q: *Retrieve the names and hours per week for each employee who works on some project controlled by department 5*. This can be specified in SQL on the schema in Figure 5.5 as follows:

```
Q:  SELECT Fname, Lname, Hours
    FROM   EMPLOYEE, PROJECT, WORKS_ON
    WHERE  Dnum=5 AND Pnumber=Pno AND Essn=Ssn;
```

Suppose that the query is submitted at site 2, which is where the query result will be needed. The DDBMS can determine from the guard condition on PROJS5 and WORKS_ON5 that all tuples satisfying the conditions (Dnum = 5 AND Pnumber = Pno) reside at site 2. Hence, it may decompose the query into the following relational algebra subqueries:

$$T_1 \leftarrow \pi_{\text{Essn}}(\text{PROJS5} \bowtie_{\text{Pnumber=Pno}} \text{WORKS\_ON5})$$
$$T_2 \leftarrow \pi_{\text{Essn, Fname, Lname}}(T_1 \bowtie_{\text{Essn=Ssn}} \text{EMPLOYEE})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Hours}}(T_{2*} \text{ WORKS\_ON5})$$

This decomposition can be used to execute the query by using a semijoin strategy. The DDBMS knows from the guard conditions that PROJS5 contains exactly those tuples satisfying (Dnum = 5) and that WORKS_ON5 contains all tuples to be joined with PROJS5; hence, subquery T_1 can be executed at site 2, and the projected column Essn can be sent to site 1. Subquery T_2 can then be executed at site 1, and the result can be sent back to site 2, where the final query result is calculated and displayed to the user. An alternative strategy would be to send the query Q itself to site 1, which includes all the database tuples, where it would be executed locally and from which the result would be sent back to site 2. The query optimizer would estimate the costs of both strategies and would choose the one with the lower cost estimate.

23.6 Types of Distributed Database Systems

The term *distributed database management system* can describe various systems that differ from one another in many respects. The main thing that all such systems have in common is the fact that data and software are distributed over multiple sites connected by some form of communication network. In this section, we discuss a number of types of DDBMSs and the criteria and factors that make some of these systems different.

The first factor we consider is the **degree of homogeneity** of the DDBMS software. If all servers (or individual local DBMSs) use identical software and all users (clients) use identical software, the DDBMS is called **homogeneous**; otherwise, it is called **heterogeneous**. Another factor related to the degree of homogeneity is the **degree of local autonomy**. If there is no provision for the local site to function as a standalone DBMS, then the system has **no local autonomy**. On the other hand, if *direct access* by local transactions to a server is permitted, the system has some degree of local autonomy.

Figure 23.6 shows classification of DDBMS alternatives along orthogonal axes of distribution, autonomy, and heterogeneity. For a centralized database, there is

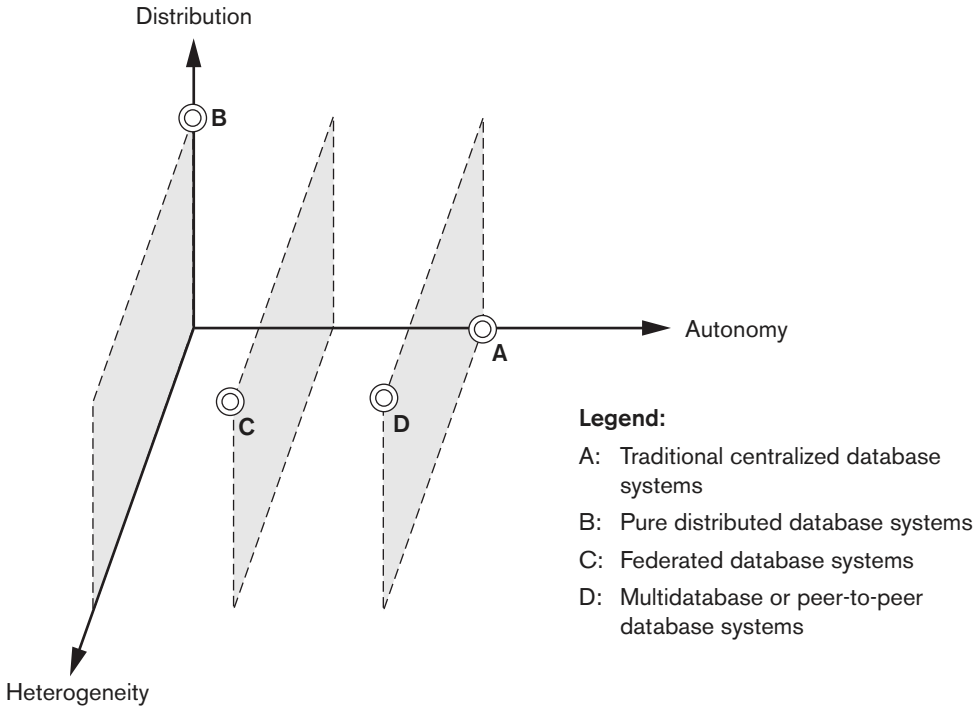

Figure 23.6
Classification of distributed databases.

complete autonomy but a total lack of distribution and heterogeneity (point A in the figure). We see that the degree of local autonomy provides further ground for classification into federated and multidatabase systems. At one extreme of the autonomy spectrum, we have a DDBMS that *looks like* a centralized DBMS to the user, with zero autonomy (point B). A single conceptual schema exists, and all access to the system is obtained through a site that is part of the DDBMS—which means that no local autonomy exists. Along the autonomy axis we encounter two types of DDBMSs called *federated database system* (point C) and *multidatabase system* (point D). In such systems, each server is an independent and autonomous centralized DBMS that has its own local users, local transactions, and DBA, and hence has a very high degree of *local autonomy*. The term **federated database system (FDBS)** is used when there is some global view or schema of the federation of databases that is shared by the applications (point C). On the other hand, a **multidatabase system** has full local autonomy in that it does not have a global schema but interactively constructs one as needed by the application (point D). Both systems are hybrids between distributed and centralized systems, and the distinction we made between them is not strictly followed. We will refer to them as FDBSs in a generic sense. Point D in the diagram may also stand for a system with full local autonomy and full heterogeneity—this could be a peer-to-peer database system. In a heterogeneous FDBS, one server may be a relational DBMS, another a network DBMS (such as Computer Associates' IDMS or HP'S IMAGE/3000), and

a third an object DBMS (such as Object Design's ObjectStore) or hierarchical DBMS (such as IBM's IMS); in such a case, it is necessary to have a canonical system language and to include language translators to translate subqueries from the canonical language to the language of each server.

We briefly discuss the issues affecting the design of FDBSs next.

23.6.1 Federated Database Management Systems Issues

The type of heterogeneity present in FDBSs may arise from several sources. We discuss these sources first and then point out how the different types of autonomies contribute to a semantic heterogeneity that must be resolved in a heterogeneous FDBS.

- **Differences in data models.** Databases in an organization come from a variety of data models, including the so-called legacy models (hierarchical and network), the relational data model, the object data model, and even files. The modeling capabilities of the models vary. Hence, to deal with them uniformly via a single global schema or to process them in a single language is challenging. Even if two databases are both from the RDBMS environment, the same information may be represented as an attribute name, as a relation name, or as a value in different databases. This calls for an intelligent query-processing mechanism that can relate information based on metadata.
- **Differences in constraints.** Constraint facilities for specification and implementation vary from system to system. There are comparable features that must be reconciled in the construction of a global schema. For example, the relationships from ER models are represented as referential integrity constraints in the relational model. Triggers may have to be used to implement certain constraints in the relational model. The global schema must also deal with potential conflicts among constraints.
- **Differences in query languages.** Even with the same data model, the languages and their versions vary. For example, SQL has multiple versions like SQL-89, SQL-92, SQL-99, and SQL:2008, and each system has its own set of data types, comparison operators, string manipulation features, and so on.

Semantic Heterogeneity. Semantic heterogeneity occurs when there are differences in the meaning, interpretation, and intended use of the same or related data. Semantic heterogeneity among component database systems (DBSs) creates the biggest hurdle in designing global schemas of heterogeneous databases. The **design autonomy** of component DBSs refers to their freedom of choosing the following design parameters; the design parameters in turn affect the eventual complexity of the FDBS:

- **The universe of discourse from which the data is drawn.** For example, for two customer accounts, databases in the federation may be from the United States and Japan and have entirely different sets of attributes about customer accounts required by the accounting practices. Currency rate fluctuations

would also present a problem. Hence, relations in these two databases that have identical names—CUSTOMER or ACCOUNT—may have some common and some entirely distinct information.

- **Representation and naming.** The representation and naming of data elements and the structure of the data model may be prespecified for each local database.
- **The understanding, meaning, and subjective interpretation of data.** This is a chief contributor to semantic heterogeneity.
- **Transaction and policy constraints.** These deal with serializability criteria, compensating transactions, and other transaction policies.
- **Derivation of summaries.** Aggregation, summarization, and other data-processing features and operations supported by the system.

The above problems related to semantic heterogeneity are being faced by all major multinational and governmental organizations in all application areas. In today's commercial environment, most enterprises are resorting to heterogeneous FDBSs, having heavily invested in the development of individual database systems using diverse data models on different platforms over the last 20 to 30 years. Enterprises are using various forms of software—typically called the **middleware**; or Web-based packages called **application servers** (for example, WebLogic or WebSphere); and even generic systems, called **enterprise resource planning (ERP) systems** (for example, SAP, J. D. Edwards ERP)—to manage the transport of queries and transactions from the global application to individual databases (with possible additional processing for business rules) and the data from the heterogeneous database servers to the global application. Detailed discussion of these types of software systems is outside the scope of this text.

Just as providing the ultimate transparency is the goal of any distributed database architecture, local component databases strive to preserve autonomy. **Communication autonomy** of a component DBS refers to its ability to decide whether to communicate with another component DBS. **Execution autonomy** refers to the ability of a component DBS to execute local operations without interference from external operations by other component DBSs and its ability to decide the order in which to execute them. The **association autonomy** of a component DBS implies that it has the ability to decide whether and how much to share its functionality (operations it supports) and resources (data it manages) with other component DBSs. The major challenge of designing FDBSs is to let component DBSs interoperate while still providing the above types of autonomies to them.

23.7 Distributed Database Architectures

In this section, we first briefly point out the distinction between parallel and distributed database architectures. Although both are prevalent in industry today, there are various manifestations of the distributed architectures that are continuously evolving among large enterprises. The parallel architecture is more common in high-per-

formance computing, where there is a need for multiprocessor architectures to cope with the volume of data undergoing transaction processing and warehousing applications. We then introduce a generic architecture of a distributed database. This is followed by discussions on the architecture of three-tier client/server and federated database systems.

23.7.1 Parallel versus Distributed Architectures

There are two main types of multiprocessor system architectures that are commonplace:

- **Shared memory (tightly coupled) architecture.** Multiple processors share secondary (disk) storage and also share primary memory.
- **Shared disk (loosely coupled) architecture.** Multiple processors share secondary (disk) storage but each has their own primary memory.

These architectures enable processors to communicate without the overhead of exchanging messages over a network.[4] Database management systems developed using the above types of architectures are termed **parallel database management systems** rather than DDBMSs, since they utilize parallel processor technology. Another type of multiprocessor architecture is called **shared-nothing architecture**. In this architecture, every processor has its own primary and secondary (disk) memory, no common memory exists, and the processors communicate over a high-speed interconnection network (bus or switch). Although the shared-nothing architecture resembles a distributed database computing environment, major differences exist in the mode of operation. In shared-nothing multiprocessor systems, there is symmetry and homogeneity of nodes; this is not true of the distributed database environment, where heterogeneity of hardware and operating system at each node is very common. Shared-nothing architecture is also considered as an environment for parallel databases. Figure 23.7(a) illustrates a parallel database (shared nothing), whereas Figure 23.7(b) illustrates a centralized database with distributed access and Figure 23.7(c) shows a pure distributed database. We will not expand on parallel architectures and related data management issues here.

23.7.2 General Architecture of Pure Distributed Databases

In this section, we discuss both the logical and component architectural models of a DDB. In Figure 23.8, which describes the generic schema architecture of a DDB, the enterprise is presented with a consistent, unified view showing the logical structure of underlying data across all nodes. This view is represented by the global conceptual schema (GCS), which provides network transparency (see Section 23.1.2). To accommodate potential heterogeneity in the DDB, each node is shown as having its own local internal schema (LIS) based on physical organization details at that

[4]If both primary and secondary memories are shared, the architecture is also known as *shared- everything architecture*.

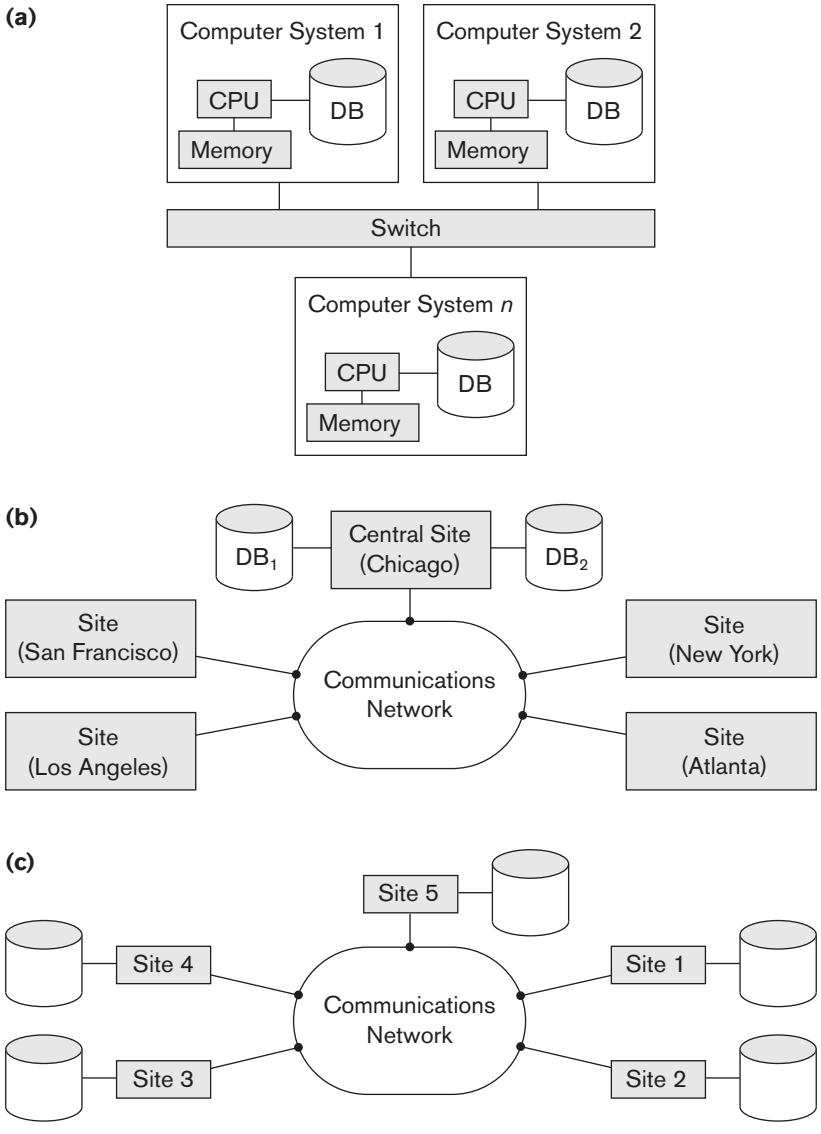

Figure 23.7
Some different database system architectures. (a) Shared-nothing architecture. (b) A networked architecture with a centralized database at one of the sites. (c) A truly distributed database architecture.

particular site. The logical organization of data at each site is specified by the local conceptual schema (LCS). The GCS, LCS, and their underlying mappings provide the fragmentation and replication transparency discussed in Section 23.1.2. Figure 23.8 shows the component architecture of a DDB. It is an extension of its centralized counterpart (Figure 2.3) in Chapter 2. For the sake of simplicity, common

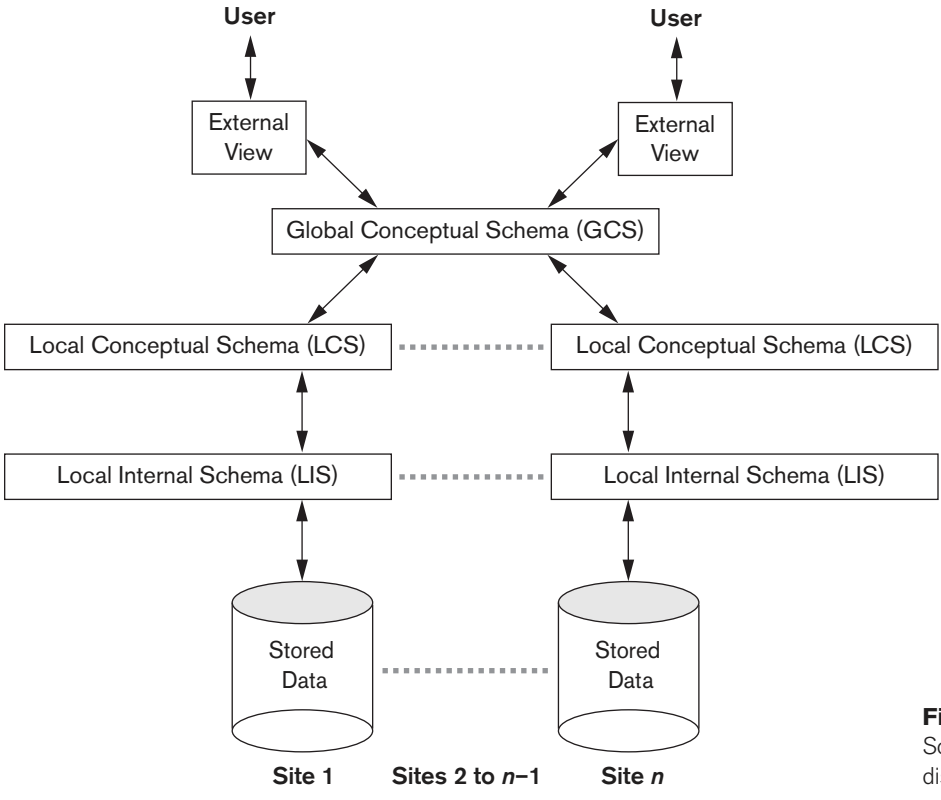

Figure 23.8
Schema architecture of distributed databases.

elements are not shown here. The global query compiler references the global conceptual schema from the global system catalog to verify and impose defined constraints. The global query optimizer references both global and local conceptual schemas and generates optimized local queries from global queries. It evaluates all candidate strategies using a cost function that estimates cost based on response time (CPU, I/O, and network latencies) and estimated sizes of intermediate results. The latter is particularly important in queries involving joins. Having computed the cost for each candidate, the optimizer selects the candidate with the minimum cost for execution. Each local DBMS would have its local query optimizer, transaction manager, and execution engines as well as the local system catalog, which houses the local schemas. The global transaction manager is responsible for coordinating the execution across multiple sites in conjunction with the local transaction manager at those sites.

23.7.3 Federated Database Schema Architecture

Typical five-level schema architecture to support global applications in the FDBS environment is shown in Figure 23.9. In this architecture, the **local schema** is the

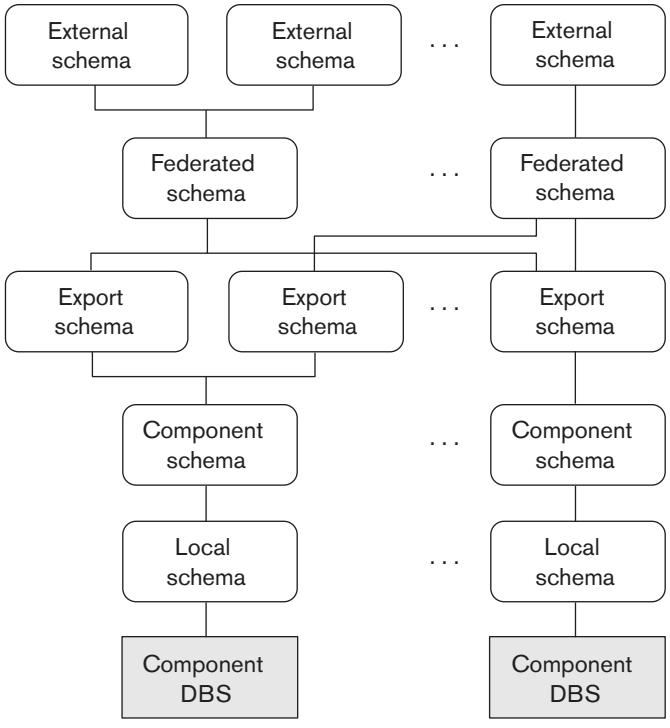

Figure 23.9
The five-level schema architecture in a federated database system (FDBS).

Source: Adapted from Sheth and Larson, "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases." *ACM Computing Surveys* (Vol. 22: No. 3, September 1990).

conceptual schema (full database definition) of a component database, and the **component schema** is derived by translating the local schema into a canonical data model or common data model (CDM) for the FDBS. Schema translation from the local schema to the component schema is accompanied by generating mappings to transform commands on a component schema into commands on the corresponding local schema. The **export schema** represents the subset of a component schema that is available to the FDBS. The **federated schema** is the global schema or view, which is the result of integrating all the shareable export schemas. The **external schemas** define the schema for a user group or an application, as in the three-level schema architecture.

All the problems related to query processing, transaction processing, and directory and metadata management and recovery apply to FDBSs with additional considerations. It is not within our scope to discuss them in detail here.

23.7.4 An Overview of Three-Tier Client/Server Architecture

As we pointed out in the chapter introduction, full-scale DDBMSs have not been developed to support all the types of functionalities that we have discussed so far. Instead, distributed database applications are being developed in the context of the client/server architectures. We introduced the two-tier client/server architecture in

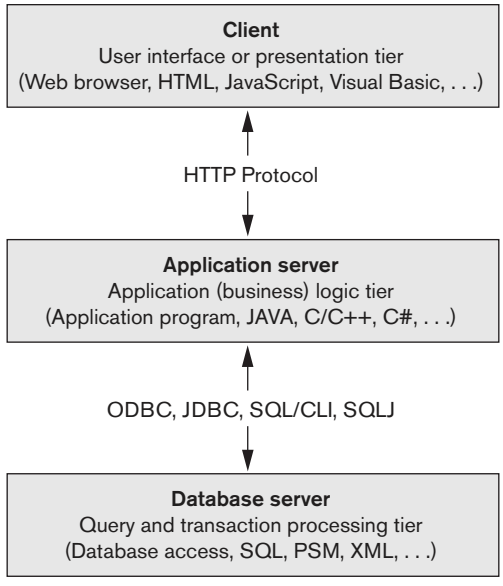

Figure 23.10
The three-tier client/server architecture.

Section 2.5. It is now more common to use a three-tier architecture rather than a two-tier architecture, particularly in Web applications. This architecture is illustrated in Figure 23.10.

In the three-tier client/server architecture, the following three layers exist:

1. **Presentation layer (client).** This provides the user interface and interacts with the user. The programs at this layer present Web interfaces or forms to the client in order to interface with the application. Web browsers are often utilized, and the languages and specifications used include HTML, XHTML, CSS, Flash, MathML, Scalable Vector Graphics (SVG), Java, JavaScript, Adobe Flex, and others. This layer handles user input, output, and navigation by accepting user commands and displaying the needed information, usually in the form of static or dynamic Web pages. The latter are employed when the interaction involves database access. When a Web interface is used, this layer typically communicates with the application layer via the HTTP protocol.
2. **Application layer (business logic).** This layer programs the application logic. For example, queries can be formulated based on user input from the client, or query results can be formatted and sent to the client for presentation. Additional application functionality can be handled at this layer, such as security checks, identity verification, and other functions. The application layer can interact with one or more databases or data sources as needed by connecting to the database using ODBC, JDBC, SQL/CLI, or other database access techniques.

3. **Database server.** This layer handles query and update requests from the application layer, processes the requests, and sends the results. Usually SQL is used to access the database if it is relational or object-relational, and stored database procedures may also be invoked. Query results (and queries) may be formatted into XML (see Chapter 13) when transmitted between the application server and the database server.

Exactly how to divide the DBMS functionality among the client, application server, and database server may vary. The common approach is to include the functionality of a centralized DBMS at the database server level. A number of relational DBMS products have taken this approach, in which an **SQL server** is provided. The application server must then formulate the appropriate SQL queries and connect to the database server when needed. The client provides the processing for user interface interactions. Since SQL is a relational standard, various SQL servers, possibly provided by different vendors, can accept SQL commands through standards such as ODBC, JDBC, and SQL/CLI (see Chapter 10).

In this architecture, the application server may also refer to a data dictionary that includes information on the distribution of data among the various SQL servers, as well as modules for decomposing a global query into a number of local queries that can be executed at the various sites. Interaction between an application server and database server might proceed as follows during the processing of an SQL query:

1. The application server formulates a user query based on input from the client layer and decomposes it into a number of independent site queries. Each site query is sent to the appropriate database server site.
2. Each database server processes the local query and sends the results to the application server site. Increasingly, XML is being touted as the standard for data exchange (see Chapter 13), so the database server may format the query result into XML before sending it to the application server.
3. The application server combines the results of the subqueries to produce the result of the originally required query, formats it into HTML or some other form accepted by the client, and sends it to the client site for display.

The application server is responsible for generating a distributed execution plan for a multisite query or transaction and for supervising distributed execution by sending commands to servers. These commands include local queries and transactions to be executed, as well as commands to transmit data to other clients or servers. Another function controlled by the application server (or coordinator) is that of ensuring consistency of replicated copies of a data item by employing distributed (or global) concurrency control techniques. The application server must also ensure the atomicity of global transactions by performing global recovery when certain sites fail.

If the DDBMS has the capability to *hide* the details of data distribution from the application server, then it enables the application server to execute global queries and transactions as though the database were centralized, without having to specify

the sites at which the data referenced in the query or transaction resides. This property is called **distribution transparency**. Some DDBMSs do not provide distribution transparency, instead requiring that applications are aware of the details of data distribution.

23.8 Distributed Catalog Management

Efficient catalog management in distributed databases is critical to ensure satisfactory performance related to site autonomy, view management, and data distribution and replication. Catalogs are databases themselves containing metadata about the distributed database system.

Three popular management schemes for distributed catalogs are *centralized* catalogs, *fully replicated* catalogs, and *partitioned* catalogs. The choice of the scheme depends on the database itself as well as the access patterns of the applications to the underlying data.

Centralized Catalogs. In this scheme, the entire catalog is stored in one single site. Due to its central nature, it is easy to implement. On the other hand, the advantages of reliability, availability, autonomy, and distribution of processing load are adversely impacted. For read operations from noncentral sites, the requested catalog data is locked at the central site and is then sent to the requesting site. On completion of the read operation, an acknowledgment is sent to the central site, which in turn unlocks this data. All update operations must be processed through the central site. This can quickly become a performance bottleneck for write-intensive applications.

Fully Replicated Catalogs. In this scheme, identical copies of the complete catalog are present at each site. This scheme facilitates faster reads by allowing them to be answered locally. However, all updates must be broadcast to all sites. Updates are treated as transactions, and a centralized two-phase commit scheme is employed to ensure catalog consistency. As with the centralized scheme, write-intensive applications may cause increased network traffic due to the broadcast associated with the writes.

Partially Replicated Catalogs. The centralized and fully replicated schemes restrict site autonomy since they must ensure a consistent global view of the catalog. Under the partially replicated scheme, each site maintains complete catalog information on data stored locally at that site. Each site is also permitted to cache entries retrieved from remote sites. However, there are no guarantees that these cached copies will be the most recent and updated. The system tracks catalog entries for sites where the object was created and for sites that contain copies of this object. Any changes to copies are propagated immediately to the original (birth) site. Retrieving updated copies to replace stale data may be delayed until an access to this data occurs. In general, fragments of relations across sites should be uniquely accessible. Also, to ensure data distribution transparency, users should be allowed to create synonyms for remote objects and use these synonyms for subsequent referrals.

23.9 Summary

In this chapter, we provided an introduction to distributed databases. This is a very broad topic, and we discussed only some of the basic techniques used with distributed databases. First in Section 23.1 we discussed the reasons for distribution and DDB concepts in Section 23.1.1. Then we defined the concept of distribution transparency and the related concepts of fragmentation transparency and replication transparency in Section 23.1.2. We discussed the concepts of distributed availability and reliability in Section 23.1.3, and gave an overview of scalability and partition tolerance issues in Section 23.1.4. We discussed autonomy of nodes in a distributed system in Section 23.1.5 and the potential advantages of distributed databases over centralized system in Section 23.1.6.

In Section 23.2, we discussed the design issues related to data fragmentation, replication, and distribution. We distinguished between horizontal fragmentation (sharding) and vertical fragmentation of relations in Section 23.2.1. We then discussed in Section 23.2.2 the use of data replication to improve system reliability and availability. In Section 23.3, we briefly discussed the concurrency control and recovery techniques used in DDBMSs, and then reviewed some of the additional problems that must be dealt with in a distributed environment that do not appear in a centralized environment. Then in Section 23.4 we discussed transaction management, including different commit protocols (2-phase commit, 3-phase commit) and operating system support for transaction management.

We then illustrated some of the techniques used in distributed query processing in Section 23.5, and discussed the cost of communication among sites, which is considered a major factor in distributed query optimization. We compared the different techniques for executing joins, and we then presented the semijoin technique for joining relations that reside on different sites in Section 23.5.3.

Following that, in Section 23.6, we categorized DDBMSs by using criteria such as the degree of homogeneity of software modules and the degree of local autonomy. In Section 23.7 we distinguished between parallel and distributed system architectures and then introduced the generic architecture of distributed databases from both a component as well as a schematic architectural perspective. In Section 23.7.3 we discussed in some detail issues of federated database management, and we focused on the needs of supporting various types of autonomies and dealing with semantic heterogeneity. We also reviewed the client/server architecture concepts and related them to distributed databases in Section 23.7.4. We reviewed catalog management in distributed databases and summarized their relative advantages and disadvantages in Section 23.8.

Chapters 24 and 25 will describe recent advances in distributed databases and distributed computing related to big data. Chapter 24 describes the so-called NOSQL systems, which are highly scalable, distributed database systems that handle large volumes of data. Chapter 25 discusses cloud computing and distributed computing technologies that are needed to process big data.

Review Questions

23.1. What are the main reasons for and potential advantages of distributed databases?

23.2. What additional functions does a DDBMS have over a centralized DBMS?

23.3. Discuss what is meant by the following terms: *degree of homogeneity of a DDBMS*, *degree of local autonomy of a DDBMS*, *federated DBMS*, *distribution transparency*, *fragmentation transparency*, *replication transparency*, *multidatabase system*.

23.4. Discuss the architecture of a DDBMS. Within the context of a centralized DBMS, briefly explain new components introduced by the distribution of data.

23.5. What are the main software modules of a DDBMS? Discuss the main functions of each of these modules in the context of the client/server architecture.

23.6. Compare the two-tier and three-tier client/server architectures.

23.7. What is a fragment of a relation? What are the main types of fragments? Why is fragmentation a useful concept in distributed database design?

23.8. Why is data replication useful in DDBMSs? What typical units of data are replicated?

23.9. What is meant by *data allocation* in distributed database design? What typical units of data are distributed over sites?

23.10. How is a horizontal partitioning of a relation specified? How can a relation be put back together from a complete horizontal partitioning?

23.11. How is a vertical partitioning of a relation specified? How can a relation be put back together from a complete vertical partitioning?

23.12. Discuss the naming problem in distributed databases.

23.13. What are the different stages of processing a query in a DDBMS?

23.14. Discuss the different techniques for executing an equijoin of two files located at different sites. What main factors affect the cost of data transfer?

23.15. Discuss the semijoin method for executing an equijoin of two files located at different sites. Under what conditions is an equijoin strategy efficient?

23.16. Discuss the factors that affect query decomposition. How are guard conditions and attribute lists of fragments used during the query decomposition process?

23.17. How is the decomposition of an update request different from the decomposition of a query? How are guard conditions and attribute lists of fragments used during the decomposition of an update request?

23.18. List the support offered by operating systems to a DDBMS and also the benefits of these supports.

23.19. Discuss the factors that do not appear in centralized systems but that affect concurrency control and recovery in distributed systems.

23.20. Discuss the two-phase commit protocol used for transaction management in a DDBMS. List its limitations and explain how they are overcome using the three-phase commit protocol.

23.21. Compare the primary site method with the primary copy method for distributed concurrency control. How does the use of backup sites affect each?

23.22. When are voting and elections used in distributed databases?

23.23. Discuss catalog management in distributed databases.

23.24. What are the main challenges facing a traditional DDBMS in the context of today's Internet applications? How does cloud computing attempt to address them?

23.25. Discuss briefly the support offered by Oracle for homogeneous, heterogeneous, and client/server-based distributed database architectures.

23.26. Discuss briefly online directories, their management, and their role in distributed databases.

Exercises

23.27. Consider the data distribution of the COMPANY database, where the fragments at sites 2 and 3 are as shown in Figure 23.3 and the fragments at site 1 are as shown in Figure 3.6. For each of the following queries, show at least two strategies of decomposing and executing the query. Under what conditions would each of your strategies work well?

a. For each employee in department 5, retrieve the employee name and the names of the employee's dependents.

b. Print the names of all employees who work in department 5 but who work on some project *not* controlled by department 5.

23.28. Consider the following relations:

BOOKS(Book#, Primary_author, Topic, Total_stock, $price)
BOOKSTORE(Store#, City, State, Zip, Inventory_value)
STOCK(Store#, Book#, Qty)

Total_stock is the total number of books in stock, and Inventory_value is the total inventory value for the store in dollars.

a. Give an example of two simple predicates that would be meaningful for the BOOKSTORE relation for horizontal partitioning.

b. How would a derived horizontal partitioning of STOCK be defined based on the partitioning of BOOKSTORE?

c. Show predicates by which BOOKS may be horizontally partitioned by topic.

d. Show how the STOCK may be further partitioned from the partitions in (b) by adding the predicates in (c).

23.29. Consider a distributed database for a bookstore chain called National Books with three sites called EAST, MIDDLE, and WEST. The relation schemas are given in Exercise 23.28. Consider that BOOKS are fragmented by $price amounts into:

B_1: BOOK1: $price up to $20
B_2: BOOK2: $price from $20.01 to $50
B_3: BOOK3: $price from $50.01 to $100
B_4: BOOK4: $price $100.01 and above

Similarly, BOOK_STORES are divided by zip codes into:

S_1: EAST: Zip up to 35000
S_2: MIDDLE: Zip 35001 to 70000
S_3: WEST: Zip 70001 to 99999

Assume that STOCK is a derived fragment based on BOOKSTORE only.

a. Consider the query:

```
SELECT   Book#, Total_stock
FROM     Books
WHERE    $price > 15 AND $price < 55;
```

Assume that fragments of BOOKSTORE are nonreplicated and assigned based on region. Assume further that BOOKS are allocated as:

EAST: B_1, B_4
MIDDLE: B_1, B_2
WEST: B_1, B_2, B_3, B_4

Assuming the query was submitted in EAST, what remote subqueries does it generate? (Write in SQL.)

b. If the price of Book# = 1234 is updated from $45 to $55 at site MIDDLE, what updates does that generate? Write in English and then in SQL.

c. Give a sample query issued at WEST that will generate a subquery for MIDDLE.

d. Write a query involving selection and projection on the above relations and show two possible query trees that denote different ways of execution.

23.70. Consider that you have been asked to propose a database architecture in a large organization (General Motors, for example) to consolidate all data

including legacy databases (from hierarchical and network models; no specific knowledge of these models is needed) as well as relational databases, which are geographically distributed so that global applications can be supported. Assume that alternative 1 is to keep all databases as they are, whereas alternative 2 is to first convert them to relational and then support the applications over a distributed integrated database.

a. Draw two schematic diagrams for the above alternatives showing the linkages among appropriate schemas. For alternative 1, choose the approach of providing export schemas for each database and constructing unified schemas for each application.

b. List the steps that you would have to go through under each alternative from the present situation until global applications are viable.

c. Compare these alternatives from the issues of:
   i. design time considerations
   ii. runtime considerations

Selected Bibliography

The textbooks by Ceri and Pelagatti (1984a) and Ozsu and Valduriez (1999) are devoted to distributed databases. Peterson and Davie (2008), Tannenbaum (2003), and Stallings (2007) cover data communications and computer networks. Comer (2008) discusses networks and internets. Ozsu et al. (1994) has a collection of papers on distributed object management.

Most of the research on distributed database design, query processing, and optimization occurred in the 1980s and 1990s; we quickly review the important references here. Distributed database design has been addressed in terms of horizontal and vertical fragmentation, allocation, and replication. Ceri et al. (1982) defined the concept of minterm horizontal fragments. Ceri et al. (1983) developed an integer programming-based optimization model for horizontal fragmentation and allocation. Navathe et al. (1984) developed algorithms for vertical fragmentation based on attribute affinity and showed a variety of contexts for vertical fragment allocation. Wilson and Navathe (1986) present an analytical model for optimal allocation of fragments. Elmasri et al. (1987) discuss fragmentation for the ECR model; Karlapalem et al. (1996) discuss issues for distributed design of object databases. Navathe et al. (1996) discuss mixed fragmentation by combining horizontal and vertical fragmentation; Karlapalem et al. (1996) present a model for redesign of distributed databases.

Distributed query processing, optimization, and decomposition are discussed in Hevner and Yao (1979), Kerschberg et al. (1982), Apers et al. (1983), Ceri and Pelagatti (1984), and Bodorick et al. (1992). Bernstein and Goodman (1981) discuss the theory behind semijoin processing. Wong (1983) discusses the use of relationships in relation fragmentation. Concurrency control and recovery schemes are discussed in Bernstein and Goodman (1981a). Kumar and Hsu (1998) compile some articles

related to recovery in distributed databases. Elections in distributed systems are discussed in Garcia-Molina (1982). Lamport (1978) discusses problems with generating unique timestamps in a distributed system. Rahimi and Haug (2007) discuss a more flexible way to construct query critical metadata for P2P databases. Ouzzani and Bouguettaya (2004) outline fundamental problems in distributed query processing over Web-based data sources.

A concurrency control technique for replicated data that is based on voting is presented by Thomas (1979). Gifford (1979) proposes the use of weighted voting, and Paris (1986) describes a method called *voting with witnesses*. Jajodia and Mutchler (1990) discuss dynamic voting. A technique called *available copy* is proposed by Bernstein and Goodman (1984), and one that uses the idea of a group is presented in ElAbbadi and Toueg (1988). Other work that discusses replicated data includes Gladney (1989), Agrawal and ElAbbadi (1990), ElAbbadi and Toueg (1989), Kumar and Segev (1993), Mukkamala (1989), and Wolfson and Milo (1991). Bassiouni (1988) discusses optimistic protocols for DDB concurrency control. Garcia-Molina (1983) and Kumar and Stonebraker (1987) discuss techniques that use the semantics of the transactions. Distributed concurrency control techniques based on locking and distinguished copies are presented by Menasce et al. (1980) and Minoura and Wiederhold (1982). Obermark (1982) presents algorithms for distributed deadlock detection. In more recent work, Vadivelu et al. (2008) propose using backup mechanism and multilevel security to develop algorithms for improving concurrency. Madria et al. (2007) propose a mechanism based on a multiversion two-phase locking scheme and timestamping to address concurrency issues specific to mobile database systems. Boukerche and Tuck (2001) propose a technique that allows transactions to be out of order to a limited extent. They attempt to ease the load on the application developer by exploiting the network environment and producing a schedule equivalent to a temporally ordered serial schedule. Han et al. (2004) propose a deadlock-free and serializable extended Petri net model for Web-based distributed real-time databases.

A survey of recovery techniques in distributed systems is given by Kohler (1981). Reed (1983) discusses atomic actions on distributed data. Bhargava (1987) presents an edited compilation of various approaches and techniques for concurrency and reliability in distributed systems.

Federated database systems were first defined in McLeod and Heimbigner (1985). Techniques for schema integration in federated databases are presented by Elmasri et al. (1986), Batini et al. (1987), Hayne and Ram (1990), and Motro (1987). Elmagarmid and Helal (1988) and Gamal-Eldin et al. (1988) discuss the update problem in heterogeneous DDBSs. Heterogeneous distributed database issues are discussed in Hsiao and Kamel (1989). Sheth and Larson (1990) present an exhaustive survey of federated database management.

Since the late 1980s, multidatabase systems and interoperability have become important topics. Techniques for dealing with semantic incompatibilities among multiple databases are examined in DeMichiel (1989), Siegel and Madnick (1991), Krishnamurthy et al. (1991), and Wang and Madnick (1989). Castano et al. (1998)

present an excellent survey of techniques for analysis of schemas. Pitoura et al. (1995) discuss object orientation in multidatabase systems. Xiao et al. (2003) propose an XML-based model for a common data model for multidatabase systems and present a new approach for schema mapping based on this model. Lakshmanan et al. (2001) propose extending SQL for interoperability and describe the architecture and algorithms for achieving the same.

Transaction processing in multidatabases is discussed in Mehrotra et al. (1992), Georgakopoulos et al. (1991), Elmagarmid et al. (1990), and Brietbart et al. (1990), among others. Elmagarmid (1992) discusses transaction processing for advanced applications, including engineering applications that are discussed in Heiler et al. (1992).

The workflow systems, which are becoming popular for managing information in complex organizations, use multilevel and nested transactions in conjunction with distributed databases. Weikum (1991) discusses multilevel transaction management. Alonso et al. (1997) discuss limitations of current workflow systems. Lopes et al. (2009) propose that users define and execute their own workflows using a client-side Web browser. They attempt to leverage Web 2.0 trends to simplify the user's work for workflow management. Jung and Yeom (2008) exploit data workflow to develop an improved transaction management system that provides simultaneous, transparent access to the heterogeneous storages that constitute the HVEM DataGrid. Deelman and Chervanak (2008) list the challenges in data-intensive scientific workflows. Specifically, they look at automated management of data, efficient mapping techniques, and user feedback issues in workflow mapping. They also argue for data reuse as an efficient means to manage data and present the challenges therein.

A number of experimental distributed DBMSs have been implemented. These include distributed INGRES by Epstein et al. (1978), DDTS by Devor and Weeldreyer (1980), SDD-1 by Rothnie et al. (1980), System R* by Lindsay et al. (1984), SIRIUS-DELTA by Ferrier and Stangret (1982), and MULTIBASE by Smith et al. (1981). The OMNIBASE system by Rusinkiewicz et al. (1988) and the Federated Information Base developed using the Candide data model by Navathe et al. (1994) are examples of federated DDBMSs. Pitoura et al. (1995) present a comparative survey of the federated database system prototypes. Most commercial DBMS vendors have products using the client/server approach and offer distributed versions of their systems. Some system issues concerning client/server DBMS architectures are discussed in Carey et al. (1991), DeWitt et al. (1990), and Wang and Rowe (1991). Khoshafian et al. (1992) discuss design issues for relational DBMSs in the client/server environment. Client/server management issues are discussed in many books, such as Zantinge and Adriaans (1996). Di Stefano (2005) discusses data distribution issues specific to grid computing. A major part of this discussion may also apply to cloud computing.

chapter 24

NOSQL Databases and Big Data Storage Systems

We now turn our attention to the class of systems developed to manage large amounts of data in organizations such as Google, Amazon, Facebook, and Twitter and in applications such as social media, Web links, user profiles, marketing and sales, posts and tweets, road maps and spatial data, and e-mail. The term **NOSQL** is generally interpreted as Not Only SQL—rather than NO to SQL—and is meant to convey that many applications need systems other than traditional relational SQL systems to augment their data management needs. Most NOSQL systems are distributed databases or distributed storage systems, with a focus on semistructured data storage, high performance, availability, data replication, and scalability as opposed to an emphasis on immediate data consistency, powerful query languages, and structured data storage.

We start in Section 24.1 with an introduction to NOSQL systems, their characteristics, and how they differ from SQL systems. We also describe four general categories of NOSQL systems—document-based, key-value stores, column-based, and graph-based. Section 24.2 discusses how NOSQL systems approach the issue of consistency among multiple replicas (copies) by using the paradigm known as **eventual consistency**. We discuss the **CAP** theorem, which can be used to understand the emphasis of NOSQL systems on availability. In Sections 24.3 through 24.6, we present an overview of each category of NOSQL systems—starting with document-based systems, followed by key-value stores, then column-based, and finally graph-based. Some systems may not fall neatly into a single category, but rather use techniques that span two or more categories of NOSQL systems. Finally, Section 24.7 is the chapter summary.

24.1 Introduction to NOSQL Systems

24.1.1 Emergence of NOSQL Systems

Many companies and organizations are faced with applications that store vast amounts of data. Consider a free e-mail application, such as Google Mail or Yahoo Mail or other similar service—this application can have millions of users, and each user can have thousands of e-mail messages. There is a need for a storage system that can manage all these e-mails; a structured relational SQL system may not be appropriate because (1) SQL systems offer too many services (powerful query language, concurrency control, etc.), which this application may not need; and (2) a structured data model such the traditional relational model may be too restrictive. Although newer relational systems do have more complex object-relational modeling options (see Chapter 12), they still require schemas, which are not required by many of the NOSQL systems.

As another example, consider an application such as Facebook, with millions of users who submit posts, many with images and videos; then these posts must be displayed on pages of other users using the social media relationships among the users. User profiles, user relationships, and posts must all be stored in a huge collection of data stores, and the appropriate posts must be made available to the sets of users that have signed up to see these posts. Some of the data for this type of application is not suitable for a traditional relational system and typically needs multiple types of databases and data storage systems.

Some of the organizations that were faced with these data management and storage applications decided to develop their own systems:

- Google developed a proprietary NOSQL system known as **BigTable**, which is used in many of Google's applications that require vast amounts of data storage, such as Gmail, Google Maps, and Web site indexing. Apache Hbase is an open source NOSQL system based on similar concepts. Google's innovation led to the category of NOSQL systems known as **column-based** or **wide column** stores; they are also sometimes referred to as **column family** stores.
- Amazon developed a NOSQL system called **DynamoDB** that is available through Amazon's cloud services. This innovation led to the category known as **key-value** data stores or sometimes **key-tuple** or **key-object** data stores.
- Facebook developed a NOSQL system called **Cassandra**, which is now open source and known as Apache Cassandra. This NOSQL system uses concepts from both key-value stores and column-based systems.
- Other software companies started developing their own solutions and making them available to users who need these capabilities—for example, **MongoDB** and **CouchDB**, which are classified as **document-based** NOSQL systems or **document stores**.
- Another category of NOSQL systems is the **graph-based** NOSQL systems, or **graph databases**; these include **Neo4J** and **GraphBase**, among others.

- Some NOSQL systems, such as **OrientDB**, combine concepts from many of the categories discussed above.
- In addition to the newer types of NOSQL systems listed above, it is also possible to classify database systems based on the object model (see Chapter 12) or on the native XML model (see Chapter 13) as NOSQL systems, although they may not have the high-performance and replication characteristics of the other types of NOSQL systems.

These are just a few examples of NOSQL systems that have been developed. There are many systems, and listing all of them is beyond the scope of our presentation.

24.1.2 Characteristics of NOSQL Systems

We now discuss the characteristics of many NOSQL systems, and how these systems differ from traditional SQL systems. We divide the characteristics into two categories—those related to distributed databases and distributed systems, and those related to data models and query languages.

NOSQL characteristics related to distributed databases and distributed systems. NOSQL systems emphasize high availability, so replicating the data is inherent in many of these systems. Scalability is another important characteristic, because many of the applications that use NOSQL systems tend to have data that keeps growing in volume. High performance is another required characteristic, whereas serializable consistency may not be as important for some of the NOSQL applications. We discuss some of these characteristics next.

1. **Scalability:** As we discussed in Section 23.1.4, there are two kinds of scalability in distributed systems: horizontal and vertical. In NOSQL systems, **horizontal scalability** is generally used, where the distributed system is expanded by adding more nodes for data storage and processing as the volume of data grows. Vertical scalability, on the other hand, refers to expanding the storage and computing power of existing nodes. In NOSQL systems, horizontal scalability is employed while the system is operational, so techniques for distributing the existing data among new nodes without interrupting system operation are necessary. We will discuss some of these techniques in Sections 24.3 through 24.6 when we discuss specific systems.
2. **Availability, Replication and Eventual Consistency:** Many applications that use NOSQL systems require continuous system availability. To accomplish this, data is replicated over two or more nodes in a transparent manner, so that if one node fails, the data is still available on other nodes. Replication improves data availability and can also improve read performance, because read requests can often be serviced from any of the replicated data nodes. However, write performance becomes more cumbersome because an update must be applied to every copy of the replicated data items; this can slow down write performance if serializable consistency is required (see Section 23.3). Many NOSQL applications do not require serializable

consistency, so more relaxed forms of consistency known as **eventual consistency** are used. We discuss this in more detail in Section 24.2.

3. **Replication Models:** Two major replication models are used in NOSQL systems: master-slave and master-master replication. **Master-slave replication** requires one copy to be the master copy; all write operations must be applied to the master copy and then propagated to the slave copies, usually using eventual consistency (the slave copies will *eventually* be the same as the master copy). For read, the master-slave paradigm can be configured in various ways. One configuration requires all reads to also be at the master copy, so this would be similar to the primary site or primary copy methods of distributed concurrency control (see Section 23.3.1), with similar advantages and disadvantages. Another configuration would allow reads at the slave copies but would not guarantee that the values are the latest writes, since writes to the slave nodes can be done after they are applied to the master copy. The **master-master replication** allows reads and writes at any of the replicas but may not guarantee that reads at nodes that store different copies see the same values. Different users may write the same data item concurrently at different nodes of the system, so the values of the item will be temporarily inconsistent. A reconciliation method to resolve conflicting write operations of the same data item at different nodes must be implemented as part of the master-master replication scheme.
4. **Sharding of Files:** In many NOSQL applications, files (or collections of data objects) can have many millions of records (or documents or objects), and these records can be accessed concurrently by thousands of users. So it is not practical to store the whole file in one node. **Sharding** (also known as **horizontal partitioning** ; see Section 23.2) of the file records is often employed in NOSQL systems. This serves to distribute the load of accessing the file records to multiple nodes. The combination of sharding the file records and replicating the shards works in tandem to improve load balancing as well as data availability. We will discuss some of the sharding techniques in Sections 24.3 through 24.6 when we discuss specific systems.
5. **High-Performance Data Access:** In many NOSQL applications, it is necessary to find individual records or objects (data items) from among the millions of data records or objects in a file. To achieve this, most systems use one of two techniques: hashing or range partitioning on object keys. The majority of accesses to an object will be by providing the key value rather than by using complex query conditions. The object key is similar to the concept of object id (see Section 12.1). In **hashing**, a hash function *h(K)* is applied to the key *K*, and the location of the object with key *K* is determined by the value of *h(K)*. In **range partitioning**, the location is determined via a range of key values; for example, location *i* would hold the objects whose key values K are in the range $Ki_{\min} \leq K \leq Ki_{\max}$. In applications that require range queries, where multiple objects within a range of key values are retrieved, range partitioned is preferred. Other indexes can also be used to locate objects based on attribute conditions different from the key K. We

will discuss some of the hashing, partitioning, and indexing techniques in Sections 24.3 through 24.6 when we discuss specific systems.

NOSQL characteristics related to data models and query languages. NOSQL systems emphasize performance and flexibility over modeling power and complex querying. We discuss some of these characteristics next.

1. **Not Requiring a Schema:** The flexibility of not requiring a schema is achieved in many NOSQL systems by allowing semi-structured, self-describing data (see Section 13.1). The users can specify a partial schema in some systems to improve storage efficiency, but it is *not required to have a schema* in most of the NOSQL systems. As there may not be a schema to specify constraints, any constraints on the data would have to be programmed in the application programs that access the data items. There are various languages for describing semistructured data, such as JSON (JavaScript Object Notation) and XML (Extensible Markup Language; see Chapter 13). JSON is used in several NOSQL systems, but other methods for describing semi-structured data can also be used. We will discuss JSON in Section 24.3 when we present document-based NOSQL systems.
2. **Less Powerful Query Languages:** Many applications that use NOSQL systems may not require a powerful query language such as SQL, because search (read) queries in these systems often locate single objects in a single file based on their object keys. NOSQL systems typically provide a set of functions and operations as a programming API (application programming interface), so reading and writing the data objects is accomplished by calling the appropriate operations by the programmer. In many cases, the operations are called **CRUD operations**, for Create, Read, Update, and Delete. In other cases, they are known as **SCRUD** because of an added Search (or Find) operation. Some NOSQL systems also provide a high-level query language, but it may not have the full power of SQL; only a subset of SQL querying capabilities would be provided. In particular, many NOSQL systems do not provide join operations as part of the query language itself; the joins need to be implemented in the application programs.
3. **Versioning:** Some NOSQL systems provide storage of multiple versions of the data items, with the timestamps of when the data version was created. We will discuss this aspect in Section 24.5 when we present column-based NOSQL systems.

In the next section, we give an overview of the various categories of NOSQL systems.

24.1.3 Categories of NOSQL Systems

NOSQL systems have been characterized into four major categories, with some additional categories that encompass other types of systems. The most common categorization lists the following four major categories:

1. **Document-based NOSQL systems:** These systems store data in the form of documents using well-known formats, such as JSON (JavaScript Object Notation). Documents are accessible via their document id, but can also be accessed rapidly using other indexes.
2. **NOSQL key-value stores:** These systems have a simple data model based on fast access by the key to the value associated with the key; the value can be a record or an object or a document or even have a more complex data structure.
3. **Column-based or wide column NOSQL systems:** These systems partition a table by column into column families (a form of vertical partitioning; see Section 23.2), where each column family is stored in its own files. They also allow versioning of data values.
4. **Graph-based NOSQL systems:** Data is represented as graphs, and related nodes can be found by traversing the edges using path expressions.

Additional categories can be added as follows to include some systems that are not easily categorized into the above four categories, as well as some other types of systems that have been available even before the term NOSQL became widely used.

5. **Hybrid NOSQL systems:** These systems have characteristics from two or more of the above four categories.
6. **Object databases:** These systems were discussed in Chapter 12.
7. **XML databases:** We discussed XML in Chapter 13.

Even keyword-based search engines store large amounts of data with fast search access, so the stored data can be considered as large NOSQL big data stores.

The rest of this chapter is organized as follows. In each of Sections 24.3 through 24.6, we will discuss one of the four main categories of NOSQL systems, and elaborate further on which characteristics each category focuses on. Before that, in Section 24.2, we discuss in more detail the concept of eventual consistency, and we discuss the associated CAP theorem.

24.2 The CAP Theorem

When we discussed concurrency control in distributed databases in Section 23.3, we assumed that the distributed database system (DDBS) is required to enforce the ACID properties (atomicity, consistency, isolation, durability) of transactions that are running concurrently (see Section 20.3). In a system with data replication, concurrency control becomes more complex because there can be multiple copies of each data item. So if an update is applied to one copy of an item, it must be applied to all other copies in a consistent manner. The possibility exists that one copy of an item X is updated by a transaction T_1 whereas another copy is updated by a transaction T_2, so two inconsistent copies of the same item exist at two different nodes in the distributed system. If two other transactions T_3 and T_4 want to read X, each may read a different copy of item X.

We saw in Section 23.3 that there are distributed concurrency control methods that do not allow this inconsistency among copies of the same data item, thus enforcing serializability and hence the isolation property in the presence of replication. However, these techniques often come with high overhead, which would defeat the purpose of creating multiple copies to improve performance and availability in distributed database systems such as NOSQL. In the field of distributed systems, there are various levels of consistency among replicated data items, from weak consistency to strong consistency. Enforcing serializability is considered the strongest form of consistency, but it has high overhead so it can reduce performance of read and write operations and hence adversely affect system performance.

The CAP theorem, which was originally introduced as the CAP principle, can be used to explain some of the competing requirements in a distributed system with replication. The three letters in CAP refer to three desirable properties of distributed systems with replicated data: **consistency** (among replicated copies), **availability** (of the system for read and write operations) and **partition tolerance** (in the face of the nodes in the system being partitioned by a network fault). *Availability* means that each read or write request for a data item will either be processed successfully or will receive a message that the operation cannot be completed. *Partition tolerance* means that the system can continue operating if the network connecting the nodes has a fault that results in two or more partitions, where the nodes in each partition can only communicate among each other. *Consistency* means that the nodes will have the same copies of a replicated data item visible for various transactions.

It is important to note here that the use of the word *consistency* in CAP and its use in ACID *do not refer to the same identical concept*. In CAP, the term *consistency* refers to the consistency of the values in different copies of the same data item in a replicated distributed system. In ACID, it refers to the fact that a transaction will not violate the integrity constraints specified on the database schema. However, if we consider that the consistency of replicated copies is a *specified constraint*, then the two uses of the term *consistency* would be related.

The **CAP theorem** states that it *is not possible to guarantee all three* of the desirable properties—consistency, availability, and partition tolerance—at the same time in a distributed system with data replication. If this is the case, then the distributed system designer would have to choose two properties out of the three to guarantee. It is generally assumed that in many traditional (SQL) applications, guaranteeing consistency through the ACID properties is important. On the other hand, in a NOSQL distributed data store, a weaker consistency level is often acceptable, and guaranteeing the other two properties (availability, partition tolerance) is important. Hence, weaker consistency levels are often used in NOSQL system instead of guaranteeing serializability. In particular, a form of consistency known as **eventual consistency** is often adopted in NOSQL systems. In Sections 24.3 through 24.6, we will discuss some of the consistency models used in specific NOSQL systems.

The next four sections of this chapter discuss the characteristics of the four main categories of NOSQL systems. We discuss document-based NOSQL systems in Section 24.3, and we use MongoDB as a representative system. In Section 24.4, we discuss

NOSQL systems known as key-value stores. In Section 24.5, we give an overview of column-based NOSQL systems, with a discussion of Hbase as a representative system. Finally, we introduce graph-based NOSQL systems in Section 24.6.

24.3 Document-Based NOSQL Systems and MongoDB

Document-based or document-oriented NOSQL systems typically store data as **collections** of similar **documents**. These types of systems are also sometimes known as **document stores**. The individual documents somewhat resemble *complex objects* (see Section 12.3) or XML documents (see Chapter 13), but a major difference between document-based systems versus object and object-relational systems and XML is that there is no requirement to specify a schema—rather, the documents are specified as **self-describing data** (see Section 13.1). Although the documents in a collection should be *similar*, they can have different data elements (attributes), and new documents can have new data elements that do not exist in any of the current documents in the collection. The system basically extracts the data element names from the self-describing documents in the collection, and the user can request that the system create indexes on some of the data elements. Documents can be specified in various formats, such as XML (see Chapter 13). A popular language to specify documents in NOSQL systems is **JSON** (JavaScript Object Notation).

There are many document-based NOSQL systems, including MongoDB and CouchDB, among many others. We will give an overview of MongoDB in this section. It is important to note that different systems can use different models, languages, and implementation methods, but giving a complete survey of all document-based NOSQL systems is beyond the scope of our presentation.

24.3.1 MongoDB Data Model

MongoDB documents are stored in BSON (Binary JSON) format, which is a variation of JSON with some additional data types and is more efficient for storage than JSON. Individual **documents** are stored in a **collection**. We will use a simple example based on our COMPANY database that we used throughout this book. The operation createCollection is used to create each collection. For example, the following command can be used to create a collection called **project** to hold PROJECT objects from the COMPANY database (see Figures 5.5 and 5.6):

```
db.createCollection("project", { capped : true, size : 1310720, max : 500 } )
```

The first parameter "project" is the **name** of the collection, which is followed by an optional document that specifies **collection options**. In our example, the collection is **capped**; this means it has upper limits on its storage space (**size**) and number of documents (**max**). The capping parameters help the system choose the storage options for each collection. There are other collection options, but we will not discuss them here.

For our example, we will create another document collection called **worker** to hold information about the EMPLOYEEs who work on each project; for example:

db.createCollection("worker", { capped : true, size : 5242880, max : 2000 }))

Each document in a collection has a unique **ObjectId** field, called **_id**, which is automatically indexed in the collection unless the user explicitly requests no index for the _id field. The value of ObjectId can be *specified by the user*, or it can be *system-generated* if the user does not specify an _id field for a particular document. *System-generated* ObjectIds have a specific format, which combines the timestamp when the object is created (4 bytes, in an internal MongoDB format), the node id (3 bytes), the process id (2 bytes), and a counter (3 bytes) into a 16-byte Id value. *User-generated* ObjectsIds can have any value specified by the user as long as it uniquely identifies the document and so these Ids are similar to primary keys in relational systems.

A collection does not have a schema. The structure of the data fields in documents is chosen based on how documents will be accessed and used, and the user can choose a normalized design (similar to normalized relational tuples) or a denormalized design (similar to XML documents or complex objects). Interdocument references can be specified by storing in one document the ObjectId or ObjectIds of other related documents. Figure 24.1(a) shows a simplified MongoDB document showing some of the data from Figure 5.6 from the COMPANY database example that is used throughout the book. In our example, the _id values are user-defined, and the documents whose _id starts with P (for project) will be stored in the "project" collection, whereas those whose _id starts with W (for worker) will be stored in the "worker" collection.

In Figure 24.1(a), the workers information is *embedded in the project document*; so there is no need for the "worker" collection. This is known as the *denormalized pattern*, which is similar to creating a complex object (see Chapter 12) or an XML document (see Chapter 13). A list of values that is enclosed in *square brackets* […] within a document represents a field whose value is an **array**.

Another option is to use the design in Figure 24.1(b), where *worker references* are embedded in the project document, but the worker documents themselves are stored in a separate "worker" collection. A third option in Figure 24.1(c) would use a normalized design, similar to First Normal Form relations (see Section 14.3.4). The choice of which design option to use depends on how the data will be accessed.

It is important to note that the simple design in Figure 24.1(c) *is not the general normalized design* for a many-to-many relationship, such as the one between employees and projects; rather, we would need three collections for "project", "employee", and "works_on", as we discussed in detail in Section 9.1. Many of the design tradeoffs that were discussed in Chapters 9 and 14 (for first normal form relations and for ER-to-relational mapping options), and Chapters 12 and 13 (for complex objects and XML) are applicable for choosing the appropriate design for document structures

Figure 24.1
Example of simple documents in MongoDB.
(a) Denormalized document design with embedded subdocuments.
(b) Embedded array of document references.
(c) Normalized documents.

(a) project document with an array of embedded workers:

```
{
    _id:            "P1",
    Pname:          "ProductX",
    Plocation:      "Bellaire",
    Workers: [
                { Ename: "John Smith",
                  Hours: 32.5
                },
                { Ename: "Joyce English",
                  Hours: 20.0
                }
            ]
);
```

(b) project document with an embedded array of worker ids:

```
{
    _id:            "P1",
    Pname:          "ProductX",
    Plocation:      "Bellaire",
    WorkerIds:      [ "W1", "W2" ]
}
    { _id:          "W1",
    Ename:          "John Smith",
    Hours:          32.5
}
    { _id:          "W2",
    Ename:          "Joyce English",
    Hours:          20.0
}
```

(c) normalized project and worker documents (not a fully normalized design for M:N relationships):

```
{
    _id:            "P1",
    Pname:          "ProductX",
    Plocation:      "Bellaire"
}
{   _id:            "W1",
    Ename:          "John Smith",
    ProjectId:      "P1",
    Hours:          32.5
}
```

```
{   _id:        "W2",
    Ename:      "Joyce English",
    ProjectId:  "P1",
    Hours:      20.0
}
```

(d) inserting the documents in (c) into their collections "project" and "worker":

```
db.project.insert( { _id: "P1", Pname: "ProductX", Plocation: "Bellaire" } )
db.worker.insert( [ { _id: "W1", Ename: "John Smith", ProjectId: "P1", Hours: 32.5 },
                    { _id: "W2", Ename: "Joyce English", ProjectId: "P1",
                      Hours: 20.0 } ] )
```

Figure 24.1 (continued) Example of simple documents in MongoDB. (d) Inserting the documents in Figure 24.1(c) into their collections.

and document collections, so we will not repeat the discussions here. In the design in Figure 24.1(c), an EMPLOYEE who works on several projects would be represented by *multiple worker documents* with different _id values; each document would represent the employee *as worker for a particular project*. This is similar to the design decisions for XML schema design (see Section 13.6). However, it is again important to note that the typical document-based system *does not have a schema*, so the design rules would have to be followed whenever individual documents are inserted into a collection.

24.3.2 MongoDB CRUD Operations

MongoDb has several **CRUD operations**, where CRUD stands for (create, read, update, delete). Documents can be *created* and inserted into their collections using the **insert** operation, whose format is:

db.<collection_name>.insert(<document(s)>)

The parameters of the insert operation can include either a single document or an array of documents, as shown in Figure 24.1(d). The *delete* operation is called **remove**, and the format is:

db.<collection_name>.remove(<condition>)

The documents to be removed from the collection are specified by a Boolean condition on some of the fields in the collection documents. There is also an **update** operation, which has a condition to select certain documents, and a *$set* clause to specify the update. It is also possible to use the update operation to replace an existing document with another one but keep the same ObjectId.

For *read* queries, the main command is called **find**, and the format is:

db.<collection_name>.find(<condition>)

General Boolean conditions can be specified as <condition>, and the documents in the collection that return **true** are selected for the query result. For a full discussion of the MongoDb CRUD operations, see the MongoDB online documentation in the chapter references.

24.3.3 MongoDB Distributed Systems Characteristics

Most MongoDB updates are atomic if they refer to a single document, but MongoDB also provides a pattern for specifying transactions on multiple documents. Since MongoDB is a distributed system, the **two-phase commit** method is used to ensure atomicity and consistency of multidocument transactions. We discussed the atomicity and consistency properties of transactions in Section 20.3, and the two-phase commit protocol in Section 22.6.

Replication in MongoDB. The concept of **replica set** is used in MongoDB to create multiple copies of the same data set on different nodes in the distributed system, and it uses a variation of the **master-slave** approach for replication. For example, suppose that we want to replicate a particular document collection C. A replica set will have one **primary copy** of the collection C stored in one node N1, and at least one **secondary copy** (replica) of C stored at another node N2. Additional copies can be stored in nodes N3, N4, etc., as needed, but the cost of storage and update (write) increases with the number of replicas. The total number of participants in a replica set must be at least three, so if only one secondary copy is needed, a participant in the replica set known as an **arbiter** must run on the third node N3. The arbiter does not hold a replica of the collection but participates in **elections** to choose a new primary if the node storing the current primary copy fails. If the total number of members in a replica set is n (one primary plus i secondaries, for a total of $n = i + 1$), then n must be an odd number; if it is not, an *arbiter* is added to ensure the election process works correctly if the primary fails. We discussed elections in distributed systems in Section 23.3.1.

In MongoDB replication, all write operations must be applied to the primary copy and then propagated to the secondaries. For read operations, the user can choose the particular **read preference** for their application. The *default read preference* processes all reads at the primary copy, so all read and write operations are performed at the primary node. In this case, secondary copies are mainly to make sure that the system continues operation if the primary fails, and MongoDB can ensure that every read request gets the latest document value. To increase read performance, it is possible to set the read preference so that *read requests can be processed at any replica* (primary or secondary); however, a read at a secondary is not guaranteed to get the latest version of a document because there can be a delay in propagating writes from the primary to the secondaries.

Sharding in MongoDB. When a collection holds a very large number of documents or requires a large storage space, storing all the documents in one node can lead to performance problems, particularly if there are many user operations accessing the documents concurrently using various CRUD operations. **Sharding** of the documents in the collection—also known as *horizontal partitioning*—divides the documents into disjoint partitions known as **shards**. This allows the system to add more nodes as needed by a process known as **horizontal scaling** of the distributed system (see Section 23.1.4), and to store the shards of the collection on different nodes to achieve load balancing. Each node will process only those operations pertaining to the documents in the shard stored at that node. Also, each

shard will contain fewer documents than if the entire collection were stored at one node, thus further improving performance.

There are two ways to partition a collection into shards in MongoDB—**range partitioning** and **hash partitioning**. Both require that the user specify a particular document field to be used as the basis for partitioning the documents into shards. The *partitioning field*—known as the **shard key** in MongoDB—must have two characteristics: it must exist in *every document* in the collection, and it must have an *index*. The ObjectId can be used, but any other field possessing these two characteristics can also be used as the basis for sharding. The values of the shard key are divided into **chunks** either through range partitioning or hash partitioning, and the documents are partitioned based on the chunks of shard key values.

Range partitioning creates the chunks by specifying a range of key values; for example, if the shard key values ranged from one to ten million, it is possible to create ten ranges—1 to 1,000,000; 1,000,001 to 2,000,000; … ; 9,000,001 to 10,000,000—and each chunk would contain the key values in one range. *Hash partitioning* applies a hash function *h(K)* to each shard key *K*, and the partitioning of keys into chunks is based on the hash values (we discussed hashing and its advantages and disadvantages in Section 16.8). In general, if **range queries** are commonly applied to a collection (for example, retrieving all documents whose shard key value is between 200 and 400), then range partitioning is preferred because each range query will typically be submitted to a single node that contains all the required documents in one shard. If most searches retrieve one document at a time, hash partitioning may be preferable because it randomizes the distribution of shard key values into chunks.

When sharding is used, MongoDB queries are submitted to a module called the **query router**, which keeps track of which nodes contain which shards based on the particular partitioning method used on the shard keys. The query (CRUD operation) will be routed to the nodes that contain the shards that hold the documents that the query is requesting. If the system cannot determine which shards hold the required documents, the query will be submitted to all the nodes that hold shards of the collection. Sharding and replication are used together; sharding focuses on improving performance via load balancing and horizontal scalability, whereas replication focuses on ensuring system availability when certain nodes fail in the distributed system.

There are many additional details about the distributed system architecture and components of MongoDB, but a full discussion is outside the scope of our presentation. MongoDB also provides many other services in areas such as system administration, indexing, security, and data aggregation, but we will not discuss these features here. Full documentation of MongoDB is available online (see the bibliographic notes).

24.4 NOSQL Key-Value Stores

Key-value stores focus on high performance, availability, and scalability by storing data in a distributed storage system. The data model used in key-value stores is relatively simple, and in many of these systems, there is no query language but rather a

set of operations that can be used by the application programmers. The **key** is a unique identifier associated with a data item and is used to locate this data item rapidly. The **value** is the data item itself, and it can have very different formats for different key-value storage systems. In some cases, the value is just a *string of bytes* or an *array of bytes*, and the application using the key-value store has to interpret the structure of the data value. In other cases, some standard formatted data is allowed; for example, structured data rows (tuples) similar to relational data, or semistructured data using JSON or some other self-describing data format. Different key-value stores can thus store unstructured, semistructured, or structured data items (see Section 13.1). The main characteristic of key-value stores is the fact that every value (data item) must be associated with a unique key, and that retrieving the value by supplying the key must be very fast.

There are many systems that fall under the key-value store label, so rather than provide a lot of details on one particular system, we will give a brief introductory overview for some of these systems and their characteristics.

24.4.1 DynamoDB Overview

The DynamoDB system is an Amazon product and is available as part of Amazon's **AWS/SDK** platforms (Amazon Web Services/Software Development Kit). It can be used as part of Amazon's cloud computing services, for the data storage component.

DynamoDB data model. The basic data model in DynamoDB uses the concepts of tables, items, and attributes. A **table** in DynamoDB *does not have* a **schema**; it holds a collection of *self-describing items*. Each **item** will consist of a number of (attribute, value) pairs, and attribute values can be single-valued or multivalued. So basically, a table will hold a collection of items, and each item is a self-describing record (or object). DynamoDB also allows the user to specify the items in JSON format, and the system will convert them to the internal storage format of DynamoDB.

When a table is created, it is required to specify a **table name** and a **primary key**; the primary key will be used to rapidly locate the items in the table. Thus, the primary key is the **key** and the item is the **value** for the DynamoDB key-value store. The primary key attribute must exist in every item in the table. The primary key can be one of the following two types:

- **A single attribute.** The DynamoDB system will use this attribute to build a hash index on the items in the table. This is called a *hash type primary key*. The items are not ordered in storage on the value of the hash attribute.
- **A pair of attributes.** This is called a *hash and range type primary key*. The primary key will be a pair of attributes (A, B): attribute A will be used for hashing, and because there will be multiple items with the same value of A, the B values will be used for ordering the records with the same A value. A table with this type of key can have additional secondary indexes defined on its attributes. For example, if we want to store multiple versions of some type of items in a table, we could use ItemID as hash and Date or Timestamp (when the version was created) as range in a hash and range type primary key.

DynamoDB Distributed Characteristics. Because DynamoDB is proprietary, in the next subsection we will discuss the mechanisms used for replication, sharding, and other distributed system concepts in an open source key-value system called Voldemort. Voldemort is based on many of the techniques proposed for DynamoDB.

24.4.2 Voldemort Key-Value Distributed Data Store

Voldemort is an open source system available through Apache 2.0 open source licensing rules. It is based on Amazon's DynamoDB. The focus is on high performance and horizontal scalability, as well as on providing replication for high availability and sharding for improving latency (response time) of read and write requests. All three of those features—replication, sharding, and horizontal scalability—are realized through a technique to distribute the key-value pairs among the nodes of a distributed cluster; this distribution is known as **consistent hashing**. Voldemort has been used by LinkedIn for data storage. Some of the features of Voldemort are as follows:

- **Simple basic operations.** A collection of (key, value) pairs is kept in a Voldemort **store**. In our discussion, we will assume the store is called *s*. The basic interface for data storage and retrieval is very simple and includes three operations: get, put, and delete. The operation s.put(k, v) inserts an item as a key-value pair with key k and value v. The operation s.delete(k) deletes the item whose key is k from the store, and the operation v = s.get(k) retrieves the value v associated with key k. The application can use these basic operations to build its own requirements. At the basic storage level, both keys and values are arrays of bytes (strings).
- **High-level formatted data values.** The values v in the (k, v) items can be specified in JSON (JavaScript Object Notation), and the system will convert between JSON and the internal storage format. Other data object formats can also be specified if the application provides the conversion (also known as **serialization**) between the user format and the storage format as a *Serializer class*. The Serializer class must be provided by the user and will include operations to convert the user format into a string of bytes for storage as a value, and to convert back a string (array of bytes) retrieved via *s.get(k)* into the user format. Voldemort has some built-in serializers for formats other than JSON.
- **Consistent hashing for distributing (key, value) pairs.** A variation of the data distribution algorithm known as **consistent hashing** is used in Voldemort for data distribution among the nodes in the distributed cluster of nodes. A hash function $h(k)$ is applied to the key k of each (k, v) pair, and $h(k)$ determines where the item will be stored. The method assumes that $h(k)$ is an integer value, usually in the range *0* to $Hmax = 2^{n-1}$, where n is chosen based on the desired range for the hash values. This method is best visualized by considering the range of all possible integer hash values *0* to *Hmax* to be evenly distributed on a circle (or ring). The nodes in the distributed system are then also located on the same ring; usually each node will have several locations on the ring (see Figure 24.2). The positioning of the points on the ring that represent the nodes is done in a psuedorandom manner.

An item (*k, v*) will be stored on the node whose position in the ring *follows* the position of *h*(*k*) on the ring *in a clockwise direction*. In Figure 24.2(a), we assume there are three nodes in the distributed cluster labeled A, B, and C, where node C has a bigger capacity than nodes A and B. In a typical system, there will be many more nodes. On the circle, two instances each of A and B are placed, and three instances of C (because of its higher capacity), in a pseudorandom manner to cover the circle. Figure 24.2(a) indicates which (k, v) items are placed in which nodes based on the *h(k)* values.

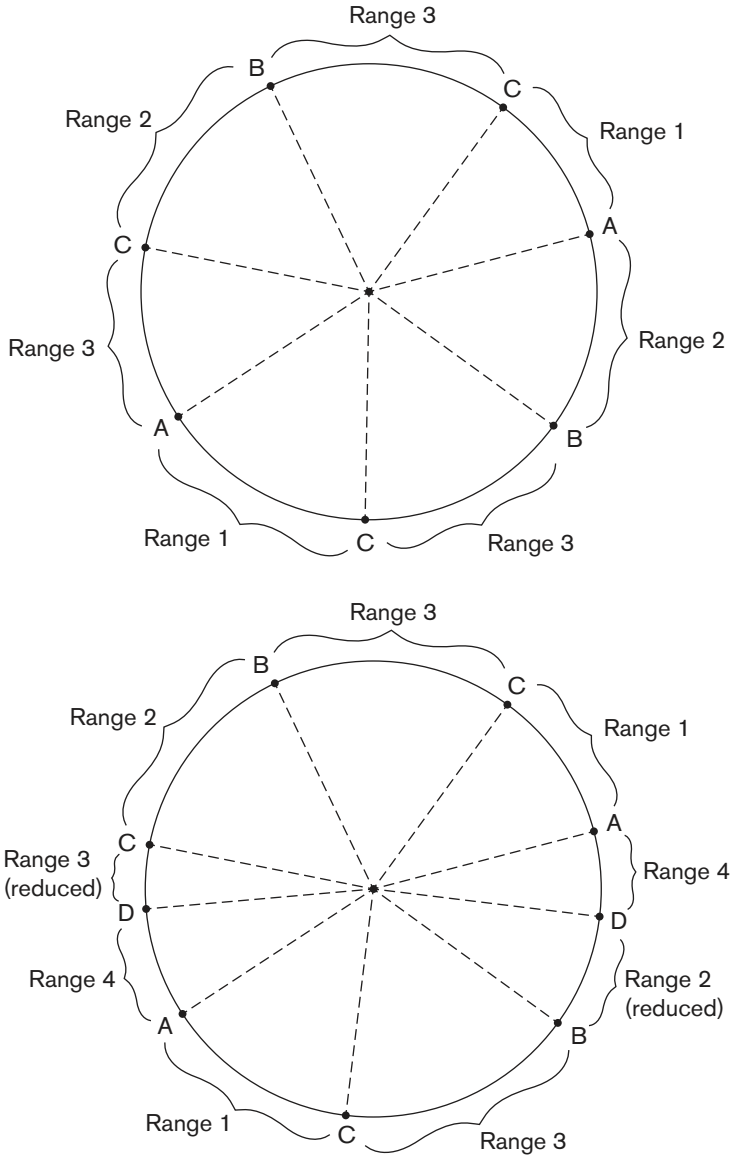

Figure 24.2
Example of consistent hashing. (a) Ring having three nodes A, B, and C, with C having greater capacity. The *h*(*K*) values that map to the circle points in *range 1* have their (k, v) items stored in node A, *range 2* in node B, *range 3* in node C. (b) Adding a node D to the ring. Items in *range 4* are moved to the node D from node B (*range 2* is reduced) and node C (*range 3* is reduced).

- The $h(k)$ values that fall in the parts of the circle marked as *range 1* in Figure 24.2(a) will have their (k, v) items stored in node A because that is the node whose label follows $h(k)$ on the ring in a clockwise direction; those in *range 2* are stored in node B; and those in *range 3* are stored in node C. This scheme allows *horizontal scalability* because when a new node is added to the distributed system, it can be added in one or more locations on the ring depending on the node capacity. Only a limited percentage of the (k, v) items will be reassigned to the new node from the existing nodes based on the consistent hashing placement algorithm. Also, those items assigned to the new node may not all come from only one of the existing nodes because the new node can have multiple locations on the ring. For example, if a node D is added and it has two placements on the ring as shown in Figure 24.2(b), then some of the items from nodes B and C would be moved to node D. The items whose keys hash to *range 4* on the circle (see Figure 24.2(b)) would be migrated to node D. This scheme also allows *replication* by placing the number of specified replicas of an item on successive nodes on the ring in a clockwise direction. The *sharding* is built into the method, and different items in the store (file) are located on different nodes in the distributed cluster, which means the items are horizontally partitioned (sharded) among the nodes in the distributed system. When a node fails, its load of data items can be distributed to the other existing nodes whose labels follow the labels of the failed node in the ring. And nodes with higher capacity can have more locations on the ring, as illustrated by node C in Figure 24.2(a), and thus store more items than smaller-capacity nodes.
- **Consistency and versioning.** Voldemort uses a method similar to the one developed for DynamoDB for consistency in the presence of replicas. Basically, concurrent write operations are allowed by different processes so there could exist two or more different values associated with the same key at different nodes when items are replicated. Consistency is achieved when the item is read by using a technique known as *versioning and read repair*. Concurrent writes are allowed, but each write is associated with a *vector clock* value. When a read occurs, it is possible that different versions of the same value (associated with the same key) are read from different nodes. If the system can reconcile to a single final value, it will pass that value to the read; otherwise, more than one version can be passed back to the application, which will reconcile the various versions into one version based on the application semantics and give this reconciled value back to the nodes.

24.4.3 Examples of Other Key-Value Stores

In this section, we briefly review three other key-value stores. It is important to note that there are many systems that can be classified in this category, and we can only mention a few of these systems.

Oracle key-value store. Oracle has one of the well-known SQL relational database systems, and Oracle also offers a system based on the key-value store concept; this system is called the **Oracle NoSQL Database**.

Redis key-value cache and store. **Redis** differs from the other systems discussed here because it caches its data in main memory to further improve performance. It offers master-slave replication and high availability, and it also offers persistence by backing up the cache to disk.

Apache Cassandra. **Cassandra** is a NOSQL system that is not easily categorized into one category; it is sometimes listed in the column-based NOSQL category (see Section 24.5) or in the key-value category. If offers features from several NOSQL categories and is used by Facebook as well as many other customers.

24.5 Column-Based or Wide Column NOSQL Systems

Another category of NOSQL systems is known as **column-based** or **wide column** systems. The Google distributed storage system for big data, known as **BigTable**, is a well-known example of this class of NOSQL systems, and it is used in many Google applications that require large amounts of data storage, such as Gmail. BigTable uses the **Google File System** (**GFS**) for data storage and distribution. An open source system known as **Apache Hbase** is somewhat similar to Google BigTable, but it typically uses **HDFS** (**Hadoop Distributed File System**) for data storage. HDFS is used in many cloud computing applications, as we shall discuss in Chapter 25. Hbase can also use Amazon's **Simple Storage System** (known as **S3**) for data storage. Another well-known example of column-based NOSQL systems is Cassandra, which we discussed briefly in Section 24.4.3 because it can also be characterized as a key-value store. We will focus on Hbase in this section as an example of this category of NOSQL systems.

BigTable (and Hbase) is sometimes described as a *sparse multidimensional distributed persistent sorted map*, where the word *map* means a *collection of (key, value) pairs* (the key is *mapped* to the value). One of the main differences that distinguish column-based systems from key-value stores (see Section 24.4) is the *nature of the key*. In column-based systems such as Hbase, the key is *multidimensional* and so has several components: typically, a combination of table name, row key, column, and timestamp. As we shall see, the column is typically composed of two components: column family and column qualifier. We discuss these concepts in more detail next as they are realized in Apache Hbase.

24.5.1 Hbase Data Model and Versioning

Hbase data model. The data model in Hbase organizes data using the concepts of *namespaces*, *tables*, *column families*, *column qualifiers*, *columns*, *rows*, and *data cells*. A column is identified by a combination of (column family:column qualifier). Data is stored in a self-describing form by associating columns with data values, where data values are strings. Hbase also stores *multiple versions* of a data item, with a *timestamp* associated with each version, so versions and timestamps are also

part of the Hbase data model (this is similar to the concept of attribute versioning in temporal databases, which we shall discuss in Section 26.2). As with other NOSQL systems, unique keys are associated with stored data items for fast access, but the keys identify *cells* in the storage system. Because the focus is on high performance when storing huge amounts of data, the data model includes some storage-related concepts. We discuss the Hbase data modeling concepts and define the terminology next. It is important to note that the use of the words *table*, *row*, and *column* is not identical to their use in relational databases, but the uses are related.

- **Tables and Rows.** Data in Hbase is stored in **tables**, and each table has a table name. Data in a table is stored as self-describing **rows**. Each row has a unique **row key**, and row keys are strings that must have the property that they can be lexicographically ordered, so characters that do not have a lexicographic order in the character set cannot be used as part of a row key.
- **Column Families, Column Qualifiers, and Columns.** A table is associated with one or more **column families**. Each column family will have a name, and the column families associated with a table *must be specified* when the table is created and cannot be changed later. Figure 24.3(a) shows how a table may be created; the table name is followed by the names of the column families associated with the table. When the data is loaded into a table, each column family can be associated with many **column qualifiers**, but the column qualifiers *are not specified* as part of creating a table. So the column qualifiers make the model a self-describing data model because the qualifiers can be dynamically specified as new rows are created and inserted into the table. A **column** is specified by a combination of ColumnFamily:ColumnQualifier. Basically, column families are a way of grouping together related columns (attributes in relational terminology) for storage purposes, except that the column qualifier names are not specified during table creation. Rather, they are specified when the data is created and stored in rows, so the data is *self-describing* since any column qualifier name can be used in a new row of data (see Figure 24.3(b)). However, it is important that the application programmers know which column qualifiers belong to each column family, even though they have the flexibility to create new column qualifiers on the fly when new data rows are created. The concept of column family is somewhat similar to *vertical partitioning* (see Section 23.2), because columns (attributes) that are accessed together because they belong to the same column family are stored in the same files. Each column family of a table is stored in its own files using the HDFS file system.
- **Versions and Timestamps.** Hbase can keep several **versions** of a data item, along with the **timestamp** associated with each version. The timestamp is a long integer number that represents the system time when the version was created, so newer versions have larger timestamp values. Hbase uses midnight 'January 1, 1970 UTC' as timestamp value zero, and uses a long integer that measures the number of milliseconds since that time as the system timestamp value (this is similar to the value returned by the Java utility java.util.Date.getTime() and is also used in MongoDB). It is also possible for

Figure 24.3
Examples in Hbase. (a) Creating a table called EMPLOYEE with three column families: Name, Address, and Details. (b) Inserting some in the EMPLOYEE table; different rows can have different self-describing column qualifiers (Fname, Lname, Nickname, Mname, Minit, Suffix, … for column family Name; Job, Review, Supervisor, Salary for column family Details). (c) Some CRUD operations of Hbase.

(a) creating a table:

```
create 'EMPLOYEE', 'Name', 'Address', 'Details'
```

(b) inserting some row data in the EMPLOYEE table:

```
put 'EMPLOYEE', 'row1', 'Name:Fname', 'John'
put 'EMPLOYEE', 'row1', 'Name:Lname', 'Smith'
put 'EMPLOYEE', 'row1', 'Name:Nickname', 'Johnny'
put 'EMPLOYEE', 'row1', 'Details:Job', 'Engineer'
put 'EMPLOYEE', 'row1', 'Details:Review', 'Good'
put 'EMPLOYEE', 'row2', 'Name:Fname', 'Alicia'
put 'EMPLOYEE', 'row2', 'Name:Lname', 'Zelaya'
put 'EMPLOYEE', 'row2', 'Name:MName', 'Jennifer'
put 'EMPLOYEE', 'row2', 'Details:Job', 'DBA'
put 'EMPLOYEE', 'row2', 'Details:Supervisor', 'James Borg'
put 'EMPLOYEE', 'row3', 'Name:Fname', 'James'
put 'EMPLOYEE', 'row3', 'Name:Minit', 'E'
put 'EMPLOYEE', 'row3', 'Name:Lname', 'Borg'
put 'EMPLOYEE', 'row3', 'Name:Suffix', 'Jr.'
put 'EMPLOYEE', 'row3', 'Details:Job', 'CEO'
put 'EMPLOYEE', 'row3', 'Details:Salary', '1,000,000'
```

(c) Some Hbase basic CRUD operations:

```
Creating a table: create <tablename>, <column family>, <column family>, …
Inserting Data: put <tablename>, <rowid>, <column family>:<column qualifier>, <value>
Reading Data (all data in a table): scan <tablename>
Retrieve Data (one item): get <tablename>,<rowid>
```

the user to define the timestamp value explicitly in a Date format rather than using the system-generated timestamp.

- **Cells.** A **cell** holds a basic data item in Hbase. The key (address) of a cell is specified by a combination of (table, rowid, columnfamily, columnqualifier, timestamp). If timestamp is left out, the latest version of the item is retrieved unless a default number of versions is specified, say the latest three versions. The default number of versions to be retrieved, as well as the default number of versions that the system needs to keep, are parameters that can be specified during table creation.
- **Namespaces.** A **namespace** is a collection of tables. A namespace basically specifies a collection of one or more tables that are typically used together by user applications, and it corresponds to a database that contains a collection of tables in relational terminology.

24.5.2 Hbase CRUD Operations

Hbase has low-level CRUD (create, read, update, delete) operations, as in many of the NOSQL systems. The formats of some of the basic CRUD operations in Hbase are shown in Figure 24.3(c).

Hbase only provides low-level CRUD operations. It is the responsibility of the application programs to implement more complex operations, such as joins between rows in different tables. The *create* operation creates a new table and specifies one or more column families associated with that table, but it does not specify the column qualifiers, as we discussed earlier. The *put* operation is used for inserting new data or new versions of existing data items. The get operation is for retrieving the data associated with a single row in a table, and the *scan* operation retrieves all the rows.

24.5.3 Hbase Storage and Distributed System Concepts

Each Hbase table is divided into a number of **regions**, where each region will hold a *range* of the row keys in the table; this is why the row keys must be lexicographically ordered. Each region will have a number of **stores**, where each column family is assigned to one store within the region. Regions are assigned to **region servers** (storage nodes) for storage. A **master server** (master node) is responsible for monitoring the region servers and for splitting a table into regions and assigning regions to region servers.

Hbase uses the **Apache Zookeeper** open source system for services related to managing the naming, distribution, and synchronization of the Hbase data on the distributed Hbase server nodes, as well as for coordination and replication services. Hbase also uses Apache HDFS (Hadoop Distributed File System) for distributed file services. So Hbase is built on top of both HDFS and Zookeeper. Zookeeper can itself have several replicas on several nodes for availability, and it keeps the data it needs in main memory to speed access to the master servers and region servers.

We will not cover the many additional details about the distributed system architecture and components of Hbase; a full discussion is outside the scope of our presentation. Full documentation of Hbase is available online (see the bibliographic notes).

24.6 NOSQL Graph Databases and Neo4j

Another category of NOSQL systems is known as **graph databases** or **graph-oriented NOSQL** systems. The data is represented as a graph, which is a collection of vertices (nodes) and edges. Both nodes and edges can be labeled to indicate the types of entities and relationships they represent, and it is generally possible to store data associated with both individual nodes and individual edges. Many systems can be categorized as graph databases. We will focus our discussion on one particular system, Neo4j, which is used in many applications. Neo4j is an open source system, and it is implemented in Java. We will discuss the Neo4j data model

in Section 24.6.1, and give an introduction to the Neo4j querying capabilities in Section 24.6.2. Section 24.6.3 gives an overview of the distributed systems and some other characteristics of Neo4j.

24.6.1 Neo4j Data Model

The data model in Neo4j organizes data using the concepts of **nodes** and **relationships**. Both nodes and relationships can have **properties**, which store the data items associated with nodes and relationships. Nodes can have **labels**; the nodes that have the *same label* are grouped into a collection that identifies a subset of the nodes in the database graph for querying purposes. A node can have zero, one, or several labels. Relationships are directed; each relationship has a *start node* and *end node* as well as a **relationship type**, which serves a similar role to a node label by identifying similar relationships that have the same relationship type. Properties can be specified via a **map pattern**, which is made of one or more "name : value" pairs enclosed in curly brackets; for example {Lname : 'Smith', Fname : 'John', Minit : 'B'}.

In conventional graph theory, nodes and relationships are generally called *vertices* and *edges*. The Neo4j graph data model somewhat resembles how data is represented in the ER and EER models (see Chapters 3 and 4) , but with some notable differences. Comparing the Neo4j graph model with ER/EER concepts, nodes correspond to *entities*, node labels correspond to *entity types and subclasses*, relationships correspond to *relationship instances*, relationship types correspond to *relationship types*, and properties correspond to *attributes*. One notable difference is that a relationship is *directed* in Neo4j, but is not in ER/EER. Another is that a node may have no label in Neo4j, which is not allowed in ER/EER because every entity must belong to an entity type. A third crucial difference is that the graph model of Neo4j is used as a basis for an actual high-performance distributed database system whereas the ER/EER model is mainly used for database design.

Figure 24.4(a) shows how a few nodes can be created in Neo4j. There are various ways in which nodes and relationships can be created; for example, by calling appropriate Neo4j operations from various Neo4j APIs. We will just show the high-level syntax for creating nodes and relationships; to do so, we will use the Neo4j CREATE command, which is part of the high-level declarative query language **Cypher**. Neo4j has many options and variations for creating nodes and relationships using various scripting interfaces, but a full discussion is outside the scope of our presentation.

- **Labels and properties.** When a node is created, the node label can be specified. It is also possible to create nodes without any labels. In Figure 24.4(a), the node labels are EMPLOYEE, DEPARTMENT, PROJECT, and LOCATION, and the created nodes correspond to some of the data from the COMPANY database in Figure 5.6 with a few modifications; for example, we use EmpId instead of SSN, and we only include a small subset of the data for illustration purposes. Properties are enclosed in curly brackets { … }. It is possible that some nodes have multiple labels; for example the same node can be labeled as PERSON and EMPLOYEE and MANAGER by listing all the labels separated by the colon symbol as follows: PERSON:EMPLOYEE:MANAGER. Having multiple labels is similar to an entity belonging to an entity type (PERSON)

plus some subclasses of PERSON (namely EMPLOYEE and MANAGER) in the EER model (see Chapter 4) but can also be used for other purposes.

- **Relationships and relationship types.** Figure 24.4(b) shows a few example relationships in Neo4j based on the COMPANY database in Figure 5.6. The → specifies the direction of the relationship, but the relationship can be traversed in either direction. The relationship types (labels) in Figure 24.4(b) are WorksFor, Manager, LocatedIn, and WorksOn; only relationships with the relationship type WorksOn have properties (Hours) in Figure 24.4(b).
- **Paths.** A **path** specifies a traversal of part of the graph. It is typically used as part of a query to specify a pattern, where the query will retrieve from the graph data that matches the pattern. A path is typically specified by a start node, followed by one or more relationships, leading to one or more end nodes that satisfy the pattern. It is somewhat similar to the concepts of path expressions that we discussed in Chapters 12 and 13 in the context of query languages for object databases (OQL) and XML (XPath and XQuery).
- **Optional Schema.** A **schema** is optional in Neo4j. Graphs can be created and used without a schema, but in Neo4j version 2.0, a few schema-related functions were added. The main features related to schema creation involve creating indexes and constraints based on the labels and properties. For example, it is possible to create the equivalent of a key constraint on a property of a label, so all nodes in the collection of nodes associated with the label must have unique values for that property.
- **Indexing and node identifiers.** When a node is created, the Neo4j system creates an internal unique system-defined identifier for each node. To retrieve individual nodes using other properties of the nodes efficiently, the user can create **indexes** for the collection of nodes that have a particular label. Typically, one or more of the properties of the nodes in that collection can be indexed. For example, Empid can be used to index nodes with the EMPLOYEE label, Dno to index the nodes with the DEPARTMENT label, and Pno to index the nodes with the PROJECT label.

24.6.2 The Cypher Query Language of Neo4j

Neo4j has a high-level query language, Cypher. There are declarative commands for creating nodes and relationships (see Figures 24.4(a) and (b)), as well as for finding nodes and relationships based on specifying patterns. Deletion and modification of data is also possible in Cypher. We introduced the CREATE command in the previous section, so we will now give a brief overview of some of the other features of Cypher.

A Cypher query is made up of *clauses*. When a query has several clauses, the result from one clause can be the input to the next clause in the query. We will give a flavor of the language by discussing some of the clauses using examples. Our presentation is not meant to be a detailed presentation on Cypher, just an introduction to some of the languages features. Figure 24.4(c) summarizes some of the main clauses that can be part of a Cyber query. The Cyber language can specify complex queries and updates on a graph database. We will give a few of examples to illustrate simple Cyber queries in Figure 24.4(d).

Figure 24.4
Examples in Neo4j using the Cypher language. (a) Creating some nodes. (b) Creating some relationships.

(a) creating some nodes for the COMPANY data (from Figure 5.6):

```
CREATE (e1: EMPLOYEE, {Empid: '1', Lname: 'Smith', Fname: 'John', Minit: 'B'})
CREATE (e2: EMPLOYEE, {Empid: '2', Lname: 'Wong', Fname: 'Franklin'})
CREATE (e3: EMPLOYEE, {Empid: '3', Lname: 'Zelaya', Fname: 'Alicia'})
CREATE (e4: EMPLOYEE, {Empid: '4', Lname: 'Wallace', Fname: 'Jennifer', Minit: 'S'})
...
CREATE (d1: DEPARTMENT, {Dno: '5', Dname: 'Research'})
CREATE (d2: DEPARTMENT, {Dno: '4', Dname: 'Administration'})
...
CREATE (p1: PROJECT, {Pno: '1', Pname: 'ProductX'})
CREATE (p2: PROJECT, {Pno: '2', Pname: 'ProductY'})
CREATE (p3: PROJECT, {Pno: '10', Pname: 'Computerization'})
CREATE (p4: PROJECT, {Pno: '20', Pname: 'Reorganization'})
...
CREATE (loc1: LOCATION, {Lname: 'Houston'})
CREATE (loc2: LOCATION, {Lname: 'Stafford'})
CREATE (loc3: LOCATION, {Lname: 'Bellaire'})
CREATE (loc4: LOCATION, {Lname: 'Sugarland'})
...
```

(b) creating some relationships for the COMPANY data (from Figure 5.6):

```
CREATE (e1) – [ : WorksFor ] –> (d1)
CREATE (e3) – [ : WorksFor ] –> (d2)
...
CREATE (d1) – [ : Manager ] –> (e2)
CREATE (d2) – [ : Manager ] –> (e4)
...
CREATE (d1) – [ : LocatedIn ] –> (loc1)
CREATE (d1) – [ : LocatedIn ] –> (loc3)
CREATE (d1) – [ : LocatedIn ] –> (loc4)
CREATE (d2) – [ : LocatedIn ] –> (loc2)
...
CREATE (e1) – [ : WorksOn, {Hours: '32.5'} ] –> (p1)
CREATE (e1) – [ : WorksOn, {Hours: '7.5'} ] –> (p2)
CREATE (e2) – [ : WorksOn, {Hours: '10.0'} ] –> (p1)
CREATE (e2) – [ : WorksOn, {Hours: 10.0} ] –> (p2)
CREATE (e2) – [ : WorksOn, {Hours: '10.0'} ] –> (p3)
CREATE (e2) – [ : WorksOn, {Hours: 10.0} ] –> (p4)
...
```

Figure 24.4 (continued)
Examples in Neo4j using the Cypher language. (c) Basic syntax of Cypher queries. (d) Examples of Cypher queries.

(c) Basic simplified syntax of some common Cypher clauses:

Finding nodes and relationships that match a pattern: MATCH <pattern>
Specifying aggregates and other query variables: WITH <specifications>
Specifying conditions on the data to be retrieved: WHERE <condition>
Specifying the data to be returned: RETURN <data>
Ordering the data to be returned: ORDER BY <data>
Limiting the number of returned data items: LIMIT <max number>
Creating nodes: CREATE <node, optional labels and properties>
Creating relationships: CREATE <relationship, relationship type and optional properties>
Deletion: DELETE <nodes or relationships>
Specifying property values and labels: SET <property values and labels>
Removing property values and labels: REMOVE <property values and labels>

(d) Examples of simple Cypher queries:

1. MATCH (d : DEPARTMENT {Dno: '5'}) – [: LocatedIn] → (loc)
   RETURN d.Dname , loc.Lname
2. MATCH (e: EMPLOYEE {Empid: '2'}) – [w: WorksOn] → (p)
   RETURN e.Ename , w.Hours, p.Pname
3. MATCH (e) – [w: WorksOn] → (p: PROJECT {Pno: 2})
   RETURN p.Pname, e.Ename , w.Hours
4. MATCH (e) – [w: WorksOn] → (p)
   RETURN e.Ename , w.Hours, p.Pname
   ORDER BY e.Ename
5. MATCH (e) – [w: WorksOn] → (p)
   RETURN e.Ename , w.Hours, p.Pname
   ORDER BY e.Ename
   LIMIT 10
6. MATCH (e) – [w: WorksOn] → (p)
   WITH e, COUNT(p) AS numOfprojs
   WHERE numOfprojs > 2
   RETURN e.Ename , numOfprojs
   ORDER BY numOfprojs
7. MATCH (e) – [w: WorksOn] → (p)
   RETURN e , w, p
   ORDER BY e.Ename
   LIMIT 10
8. MATCH (e: EMPLOYEE {Empid: '2'})
   SET e.Job = 'Engineer'

Query 1 in Figure 24.4(d) shows how to use the MATCH and RETURN clauses in a query, and the query retrieves the locations for department number 5. Match specifies the *pattern* and the *query variables* (*d* and *loc*) and RETURN specifies the query result to be retrieved by refering to the query variables. Query 2 has three variables (*e*, *w*, and *p*), and returns the projects and hours per week that the employee with

Empid = 2 works on. Query 3, on the other hand, returns the employees and hours per week who work on the project with Pno = 2. Query 4 illustrates the ORDER BY clause and returns all employees and the projects they work on, sorted by Ename. It is also possible to limit the number of returned results by using the LIMIT clause as in query 5, which only returns the first 10 answers.

Query 6 illustrates the use of WITH and aggregation, although the WITH clause can be used to separate clauses in a query even if there is no aggregation. Query 6 also illustrates the WHERE clause to specify additional conditions, and the query returns the employees who work on more than two projects, as well as the number of projects each employee works on. It is also common to return the nodes and relationships themselves in the query result, rather than the property values of the nodes as in the previous queries. Query 7 is similar to query 5 but returns the nodes and relationships only, and so the query result can be displayed as a graph using Neo4j's visualization tool. It is also possible to add or remove labels and properties from nodes. Query 8 shows how to add more properties to a node by adding a Job property to an employee node.

The above gives a brief flavor for the Cypher query language of Neo4j. The full language manual is available online (see the bibliographic notes).

24.6.3 Neo4j Interfaces and Distributed System Characteristics

Neo4j has other interfaces that can be used to create, retrieve, and update nodes and relationships in a graph database. It also has two main versions: the enterprise edition, which comes with additional capabilities, and the community edition. We discuss some of the additional features of Neo4j in this subsection.

- **Enterprise edition vs. community edition.** Both editions support the Neo4j graph data model and storage system, as well as the Cypher graph query language, and several other interfaces, including a high-performance native API, language drivers for several popular programming languages, such as Java, Python, PHP, and the REST (Representational State Transfer) API. In addition, both editions support ACID properties. The enterprise edition supports additional features for enhancing performance, such as caching and clustering of data and locking.
- **Graph visualization interface.** Neo4j has a graph visualization interface, so that a subset of the nodes and edges in a database graph can be displayed as a graph. This tool can be used to visualize query results in a graph representation.
- **Master-slave replication.** Neo4j can be configured on a cluster of distributed system nodes (computers), where one node is designated the master node. The data and indexes are fully replicated on each node in the cluster. Various ways of synchronizing the data between master and slave nodes can be configured in the distributed cluster.
- **Caching.** A main memory cache can be configured to store the graph data for improved performance.
- **Logical logs.** Logs can be maintained to recover from failures.

A full discussion of all the features and interfaces of Neo4j is outside the scope of our presentation. Full documentation of Neo4j is available online (see the bibliographic notes).

24.7 Summary

In this chapter, we discussed the class of database systems known as NOSQL systems, which focus on efficient storage and retrieval of large amounts of "big data." Applications that use these types of systems include social media, Web links, user profiles, marketing and sales, posts and tweets, road maps and spatial data, and e-mail. The term *NOSQL* is generally interpreted as Not Only SQL—rather than NO to SQL—and is meant to convey that many applications need systems other than traditional relational SQL systems to augment their data management needs. These systems are distributed databases or distributed storage systems, with a focus on semistructured data storage, high performance, availability, data replication, and scalability rather than an emphasis on immediate data consistency, powerful query languages, and structured data storage.

In Section 24.1, we started with an introduction to NOSQL systems, their characteristics, and how they differ from SQL systems. Four general categories of NOSQL systems are document-based, key-value stores, column-based, and graph-based. In Section 24.2, we discussed how NOSQL systems approach the issue of consistency among multiple replicas (copies) by using the paradigm known as eventual consistency. We discussed the CAP theorem, which can be used to understand the emphasis of NOSQL systems on availability. In Sections 24.3 through 24.6, we presented an overview of each of the four main categories of NOSQL systems—starting with document-based systems in Section 24.3, followed by key-value stores in Section 24.4, then column-based systems in Section 24.5, and finally graph-based systems in Section 24.6. We also noted that some NOSQL systems may not fall neatly into a single category but rather use techniques that span two or more categories.

Review Questions

24.1. For which types of applications were NOSQL systems developed?

24.2. What are the main categories of NOSQL systems? List a few of the NOSQL systems in each category.

24.3. What are the main characteristics of NOSQL systems in the areas related to data models and query languages?

24.4. What are the main characteristics of NOSQL systems in the areas related to distributed systems and distributed databases?

24.5. What is the CAP theorem? Which of the three properties (consistency, availability, partition tolerance) are most important in NOSQL systems?

24.6. What are the similarities and differences between using consistency in CAP versus using consistency in ACID?

24.7. What are the data modeling concepts used in MongoDB? What are the main CRUD operations of MongoDB?

24.8. Discuss how replication and sharding are done in MongoDB.

24.9. Discuss the data modeling concepts in DynamoDB.

24.10. Describe the consistent hashing schema for data distribution, replication, and sharding. How are consistency and versioning handled in Voldemort?

24.11. What are the data modeling concepts used in column-based NOSQL systems and Hbase?

24.12. What are the main CRUD operations in Hbase?

24.13. Discuss the storage and distributed system methods used in Hbase.

24.14. What are the data modeling concepts used in the graph-oriented NOSQL system Neo4j?

24.15. What is the query language for Neo4j?

24.16. Discuss the interfaces and distributed systems characteristics of Neo4j.

Selected Bibliography

The original paper that described the Google BigTable distributed storage system is Chang et al. (2006), and the original paper that described the Amazon Dynamo key-value store system is DeCandia et al. (2007). There are numerous papers that compare various NOSQL systems with SQl (relational systems); for example, Parker et al. (2013). Other papers compare NOSQL systems to other NOSQL systems; for example Cattell (2010), Hecht and Jablonski (2011), and Abramova and Bernardino (2013).

The documentation, user manuals, and tutorials for many NOSQL systems can be found on the Web. Here are a few examples:

MongoDB tutorials: docs.mongodb.org/manual/tutorial/

MongoDB manual: docs.mongodb.org/manual/

Voldemort documentation: docs.project-voldemort.com/voldemort/

Cassandra Web site: cassandra.apache.org

Hbase Web site: hbase.apache.org

Neo4j documentation: neo4j.com/docs/

In addition, numerous Web sites categorize NOSQL systems into additional subcategories based on purpose; nosql-database.org is one example of such a site.

chapter 25

Big Data Technologies Based on MapReduce and Hadoop[1]

The amount of data worldwide has been growing ever since the advent of the World Wide Web around 1994. The early search engines—namely, AltaVista (which was acquired by Yahoo in 2003 and which later became the Yahoo! search engine) and Lycos (which was also a search engine and a Web portal—were established soon after the Web came along. They were later overshadowed by the likes of Google and Bing. Then came an array of social networks such as Facebook, launched in 2004, and Twitter, founded in 2006. LinkedIn, a professional network launched in 2003, boasts over 250 million users worldwide. Facebook has over 1.3 billion users worldwide today; of these, about 800 million are active on Facebook daily. Twitter had an estimated 980 million users in early 2014 and it was reported to have reached the rate of 1 billion tweets per day in October 2012. These statistics are updated continually and are easily available on the Web.

One major implication of the establishment and exponential growth of the Web, which brought computing to laypeople worldwide, is that ordinary people started creating all types of transactions and content that generate new data. These users and consumers of multimedia data require systems to deliver user-specific data instantaneously from mammoth stores of data at the same time that they create huge amounts of data themselves. The result is an explosive growth in the amount of data generated and communicated over networks worldwide; in addition, businesses and governmental institutions electronically record every transaction of each customer, vendor, and supplier and thus have been accumulating data in so-called data warehouses (to be discussed in Chapter 29). Added to this mountain of data is the data

[1]We acknowledge the significant contribution of Harish Butani, member of the Hive Program Management Committee, and Balaji Palanisamy, University of Pittsburgh, to this chapter.

generated by sensors embedded in devices such as smartphones, energy smart meters, automobiles, and all kinds of gadgets and machinery that sense, create, and communicate data in the internet of things. And, of course, we must consider the data generated daily from satellite imagery and communication networks.

This phenomenal growth of data generation means that the amount of data in a single repository can be numbered in petabytes (10**15 bytes, which approximates to 2**50 bytes) or terabytes (e.g., 1,000 terabytes). The term *big data* has entered our common parlance and refers to such massive amounts of data. The McKinsey report[2] defines the term *big data* as datasets whose size exceeds the typical reach of a DBMS to capture, store, manage, and analyze that data. The meaning and implications of this data onslaught are reflected in some of the facts mentioned in the McKinsey report:

- A $600 disk can store all of the world's music today.
- Every month, 30 billion of items of content are stored on Facebook.
- More data is stored in 15 of the 17 sectors of the U.S. economy than is stored in the Library of Congress, which, as of 2011, stored 235 terabytes of data.
- There is currently a need for over 140,000 deep-data-analysis positions and over 1.5 million data-savvy managers in the United States. Deep data analysis involves more knowledge discovery type analyses.

Big data is everywhere, so every sector of the economy stands to benefit by harnessing it appropriately with technologies that will help data users and managers make better decisions based on historical evidence. According to the Mckinsey report,

> If the U.S. healthcare [system] could use the big data creatively and effectively to drive efficiency and quality, we estimate that the potential value from data in the sector could be more than $300 billion in value every year.

Big data has created countless opportunities to give consumers information in a timely manner—information that will prove useful in making decisions, discovering needs and improving performance, customizing products and services, giving decision makers more effective algorithmic tools, and creating value by innovations in terms of new products, services, and business models. IBM has corroborated this statement in a recent book,[3] which outlines why IBM has embarked on a worldwide mission of enterprise-wide big data analytics. The IBM book describes various types of analytics applications:

- **Descriptive and predictive analytics**: Descriptive analytics relates to reporting what has happened, analyzing the data that contributed to it to figure out why it happened, and monitoring new data to find out what is happening now. Predictive analytics uses statistical and data mining techniques (see Chapter 28) to make predictions about what will happen in the future.

---

[2]The introduction is largely based on the McKinsey (2012) report on big data from the McKinsey Global Institute.

[3]See IBM (2014): *Analytics Across the Enterprise: How IBM Realizes Business Value from Big Data and Analytics.*

- **Prescriptive analytics:** Refers to analytics that recommends actions.
- **Social media analytics:** Refers to doing a sentiment analysis to assess public opinion on topics or events. It also allows users to discover the behavior patterns and tastes of individuals, which can help industry target goods and services in a customized way.
- **Entity analytics:** This is a somewhat new area that groups data about entities of interest and learns more about them.
- **Cognitive computing:** Refers to an area of developing computing systems that will interact with people to give them better insight and advice.

In another book, Bill Franks of Teradata[4] voices a similar theme; he states that tapping big data for better analytics is essential for a competitive advantage in any industry today, and he shows how to develop a "big data advanced analytics ecosystem" in any organization to uncover new opportunities in business.

As we can see from all these industry-based publications by experts, big data is entering a new frontier in which big data will be harnessed to provide analytics-oriented applications that will lead to increased productivity, higher quality, and growth in all businesses. This chapter discusses the technology that has been created over the last decade to harness big data. We focus on those technologies that can be attributed to the MapReduce/Hadoop ecosystem, which covers most of the ground of open source projects for big data applications. We will not be able to get into the applications of the big data technology for analytics. That is a vast area by itself. Some of the basic data mining concepts are mentioned in Chapter 28; however, today's analytics offerings go way beyond the basic concepts we have outlined there.

In Section 25.1, we introduce the essential features of big data. In Section 25.2, we will give the historical background behind the MapReduce/Hadoop technology and comment on the various releases of Hadoop. Section 25.3 discusses the underlying file system called Hadoop Distributed File System for Hadoop. We discuss its architecture, the I/O operations it supports, and its scalability. Section 25.4 provides further details on MapReduce (MR), including its runtime environment and high-level interfaces called Pig and Hive. We also show the power of MapReduce in terms of the relational join implemented in various ways. Section 25.5 is devoted to the later development called Hadoop v2 or MRv2 or YARN, which separates resource management from job management. Its rationale is explained first, and then its architecture and other frameworks being developed on YARN are explained. In Section 25.6 we discuss some general issues related to the MapReduce/Hadoop technology. First we discuss this technology vis-à-vis the parallel DBMS technology. Then we discuss it in the context of cloud computing, and we mention the data locality issues for improving performance. YARN as a data service platform is discussed next, followed by the challenges for big data technology in general. We end this chapter in Section 25.7 by mentioning some ongoing projects and summarizing the chapter.

[4]See Franks (2013) : *Taming The Big Data Tidal Wave*.

25.1 What Is Big Data?

Big data is becoming a popular and even a fashionable term. People use this term whenever a large amount of data is involved with some analysis; they think that using this term will make the analysis look like an advanced application. However, the term *big data* legitimately refers to datasets whose size is beyond the ability of typical database software tools to capture, store, manage, and analyze. In today's environment, the size of datasets that may be considered as big data ranges from terabytes (10**12 bytes), or petabytes (10**15 bytes), to exabytes (10**18 bytes). The notion of what is Big data will depend on the industry, how data is used, how much historical data is involved and many other characteristics. The Gartner Group, a popular enterprise-level organization that industry looks up to for learning about trends, characterized big data in 2011 by the three V's: volume, velocity, and variety. Other characteristics, such as veracity and value, have been added to the definition by other researchers. Let us briefly see what these stand for.

Volume. The volume of data obviously refers to the size of data managed by the system. Data that is somewhat automatically generated tends to be voluminous. Examples include sensor data, such as the data in manufacturing or processing plants generated by sensors; data from scanning equipment, such as smart card and credit card readers; and data from measurement devices, such as smart meters or environmental recording devices.

The **industrial internet of things** (IIOT or IOT) is expected to bring about a revolution that will improve the operational efficiency of enterprises and open up new frontiers for harnessing intelligent technologies. The IOT will cause billions of devices to be connected to the Internet because these devices generate data continuously. For example, in gene sequencing, next generation sequencing (NGS) technology means that the volume of gene sequence data will be increased exponentially.

Many additional applications are being developed and are slowly becoming a reality. These applications include using remote sensing to detect underground sources of energy, environmental monitoring, traffic monitoring and regulation by automatic sensors mounted on vehicles and roads, remote monitoring of patients using special scanners and equipment, and tighter control and replenishment of inventories using radio-frequency identification (RFID) and other technologies. All these developments will have associated with them a large volume of data. Social networks such as Twitter and Facebook have hundreds of millions of subscribers worldwide who generate new data with every message they send or post they make. Twitter hit a half billion tweets daily in October 2012.[5] The amount of data required to store one second of high-definition video may equal 2,000 pages of text data. Thus, the multimedia data being uploaded on YouTube and similar video hosting platforms is significantly more voluminous than simple numeric or text data. In 2010, enterprises stored over 13 exabytes (10**18 bytes) of data, which amounts to over 50,000 times the amount of data stored by the Library of Congress.[6]

[5]See Terdiman (2012): http://www.cnet.com/news/report-twitter-hits-half-a-billion-tweets-a-day/

[6]From Jagadish et al. (2014).

Velocity. The definition of *big data* goes beyond the dimension of volume; it includes the types and frequency of data that are disruptive to traditional database management tools. The Mckinsey report on big data[7] described velocity as the speed at which data is created, accumulated, ingested, and processed. High velocity is attributed to data when we consider the typical speed of transactions on stock exchanges; this speed reaches billions of transactions per day on certain days. If we must process these transactions to detect potential fraud or we must process billions of call records on cell phones daily to detect malicious activity, we face the velocity dimension. Real-time data and streaming data are accumulated by the likes of Twitter and Facebook at a very high velocity. Velocity is helpful in detecting trends among people that are tweeting a million tweets every three minutes. Processing of streaming data for analysis also involves the velocity dimension.

Variety. Sources of data in traditional applications were mainly transactions involving financial, insurance, travel, healthcare, retail industries, and governmental and judicial processing. The types of sources have expanded dramatically and include Internet data (e.g., clickstream and social media), research data (e.g., surveys and industry reports), location data (e.g., mobile device data and geospatial data), images (e.g., surveillance, satellites and medical scanning), e-mails, supply chain data (e.g., EDI—electronic data interchange, vendor catalogs), signal data (e.g., sensors and RFID devices), and videos (YouTube enters hundreds of minutes of video every minute). Big data includes structured, semistructured, and unstructured data (see discussion in Chapter 26) in different proportions based on context.

Structured data feature a formally structured data model, such as the relational model, in which data are in the form of tables containing rows and columns, and a hierarchical database in IMS, which features record types as segments and fields within a record.

Unstructured data have no identifiable formal structure. We discussed systems like MongoDB (in Chapter 24), which stores unstructured document-oriented data, and Neo4j, which stores data in the form of a graph. Other forms of unstructured data include e-mails and blogs, PDF files, audio, video, images, clickstreams, and Web contents. The advent of the World Wide Web in 1993–1994 led to tremendous growth in unstructured data. Some forms of unstructured data may fit into a format that allows well-defined tags that separate semantic elements; this format may include the capability to enforce hierarchies within the data. XML is hierarchical in its descriptive mechanism, and various forms of XML have come about in many domains; for example, biology (bioML—biopolymer markup language), GIS (gML—geography markup language), and brewing (BeerXML—language for exchange of brewing data), to name a few. Unstructured data constitutes the major challenge in today's big data systems.

Veracity. The veracity dimension of big data is a more recent addition than the advent of the Internet. Veracity has two built-in features: the credibility of the source, and the suitability of data for its target audience. It is closely related to trust;

[7]See Mckinsey (2013).

listing veracity as one of the dimensions of big data amounts to saying that data coming into the so-called big data applications have a variety of trustworthiness, and therefore before we accept the data for analytical or other applications, it must go through some degree of quality testing and credibility analysis. Many sources of data generate data that is uncertain, incomplete, and inaccurate, therefore making its veracity questionable.

We now turn our attention to the technologies that are considered the pillars of big data technologies. It is anticipated that by 2016, more than half of the data in the world may be processed by Hadoop-related technologies. It is therefore important for us to trace the MapReduce/Hadoop revolution and understand how this technology is positioned today. The historical development starts with the programming paradigm called MapReduce programming.

25.2 Introduction to MapReduce and Hadoop

In this section, we will introduce the technology for big data analytics and data processing known as Hadoop, an open source implementation of the MapReduce programming model. The two core components of Hadoop are the MapReduce programming paradigm and HDFS, the Hadoop Distributed File System. We will briefly explain the background behind Hadoop and then MapReduce. Then we will make some brief remarks about the Hadoop ecosystem and the Hadoop releases.

25.2.1 Historical Background

Hadoop has originated from the quest for an open source search engine. The first attempt was made by the then Internet archive director Doug Cutting and University of Washington graduate student Mike Carafella. Cutting and Carafella developed a system called Nutch that could crawl and index hundreds of millions of Web pages. It is an open source Apache project.[8] After Google released the Google File System[9] paper in October 2003 and the MapReduce programming paradigm paper[10] in December 2004, Cutting and Carafella realized that a number of things they were doing could be improved based on the ideas in these two papers. They built an underlying file system and a processing framework that came to be known as Hadoop (which used Java as opposed to the C++ used in MapReduce) and ported Nutch on top of it. In 2006, Cutting joined Yahoo, where there was an effort under way to build open source technologies using ideas from the Google File System and the MapReduce programming paradigm. Yahoo wanted to enhance its search processing and build an open source infrastructure based on the Google File System and MapReduce. Yahoo spun off the storage engine and the processing parts of Nutch as **Hadoop** (named after the stuffed elephant toy of Cutting's son). The

[8]For documentation on Nutch, see http:nutch.apache.org

[9]Ghemawat, Gbioff, and Leung (2003).

[10]Dean and Ghemawat (2004).

initial requirements for Hadoop were to run batch processing using cases with a high degree of scalability. However, the circa 2006 Hadoop could only run on a handful of nodes. Later, Yahoo set up a research forum for the company's data scientists; doing so improved the search relevance and ad revenue of the search engine and at the same time helped to mature the Hadoop technology. In 2011, Yahoo spun off Hortonworks as a Hadoop-centered software company. By then, Yahoo's infrastructure contained hundreds of petabytes of storage and 42,000 nodes in the cluster. In the years since Hadoop became an open source Apache project, thousands of developers worldwide have contributed to it. A joint effort by Google, IBM, and NSF used a 2,000-node Hadoop cluster at a Seattle data center and helped further universities' research on Hadoop. Hadoop has seen tremendous growth since the 2008 launch of Cloudera as the first commercial Hadoop company and the subsequent mushrooming of a large number of startups. IDC, a software industry market analysis firm, predicts that the Hadoop market will surpass $800 million in 2016; IDC predicts that the big data market will hit $23 billion in 2016. For more details about the history of Hadoop, consult a four-part article by Harris.[11]

An integral part of Hadoop is the MapReduce programming framework. Before we go any further, let us try to understand what the MapReduce programming paradigm is all about. We defer a detailed discussion of the HDFS file system to Section 25.3.

25.2.2 MapReduce

The MapReduce programming model and runtime environment was first described by Jeffrey Dean and Sanjay Ghemawat (Dean & Ghemawat (2004)) based on their work at Google. Users write their programs in a functional style of *map* and *reduce* tasks, which are automatically parallelized and executed on large clusters of commodity hardware. The programming paradigm has existed as far back as the language LISP, which was designed by John McCarthy in late 1950s. However, the reincarnation of this way of doing parallel programming and the way this paradigm was implemented at Google gave rise to a new wave of thinking that contributed to the subsequent developments of technologies such as Hadoop. The runtime system handles many of the messy engineering aspects of parallelization, fault tolerance, data distribution, load balancing, and management of task communication. As long as users adhere to the **contracts** laid out by the MapReduce system, they can just focus on the logical aspects of this program; this allows programmers without distributed systems experience to perform analysis on very large datasets.

The motivation behind the MapReduce system was the years spent by the authors and others at Google implementing hundreds of special-purpose computations on large datasets (e.g., computing inverted indexes from Web content collected via Web crawling; building Web graphs; and extracting statistics from Web logs, such as frequency distribution of search requests by topic, by region, by type of user, etc.). Conceptually, these tasks are not difficult to express; however, given the scale

[11]Derreck Harris : 'The history of Hadoop: from 4 nodes to the future of data," at https://gigaom.com/2013/03/04/the-history-of-hadoop-from-4-nodes-to-the-future-of-data/

of data in billions of Web pages and with the data spread over thousands of machines, the execution task was nontrivial. Issues of program control and data management, data distribution, parallelization of computation, and handling of failures became critically important.

The MapReduce programming model and runtime environment was designed to cope with the above complexity. The abstraction is inspired by the map and reduce primitives present in LISP and many other functional languages. An underlying model of data is assumed; this model treats an object of interest in the form of a unique key that has associated content or value. This is the key-value pair. Surprisingly, many computations can be expressed as applying a map operation to each logical "record" that produces a set of intermediate key-value pairs and then applying a reduce operation to all the values that shared the same key (the purpose of sharing is to combine the derived data). This model allows the infrastructure to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance. The idea of providing a restricted programming model so that the runtime can parallelize computations automatically is not new. MapReduce is the enhancement of those existing ideas. As it is understood today, MapReduce is a fault-tolerant implementation and a runtime environment that scales to thousands of processors. The programmer is spared the worry of handling failures. In subsequent sections, we will abbreviate MapReduce as **MR**.

The MapReduce Programming Model In the following description, we use the formalism and description as it was originally described by Dean and Ghemawat (2010).[12] The map and reduce functions have the following general form:

map[K1,V1] which is (key, value) : List[K2,V2] and
reduce(K2, List[V2]) : List[K3,V3]

Map is a generic function that takes a key of type **K1** and a value of type **V1** and returns a list of key-value pairs of type **K2** and **V2**. **Reduce** is a generic function that takes a key of type **K2** and a list of values of type **V2** and returns pairs of type **(K3,V3)**. In general, the types K1, K2, K3, etc., are different, with the only requirement that the output types from the Map function must match the input type of the Reduce function.

The basic execution workflow of MapReduce is shown in Figure 25.1.

Assume that we have a document and we want to make a list of words in it with their corresponding frequencies. This ubiquitous *word count* example quoted directly from Dean and Ghemawat (2004) above goes as follows in pseudocode:

Map (String key, String value):
for each word w in value Emitintermediate (w, "1");

Here key is the document name, and value is the text content of the document.

[12]Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in OSDI (2004).

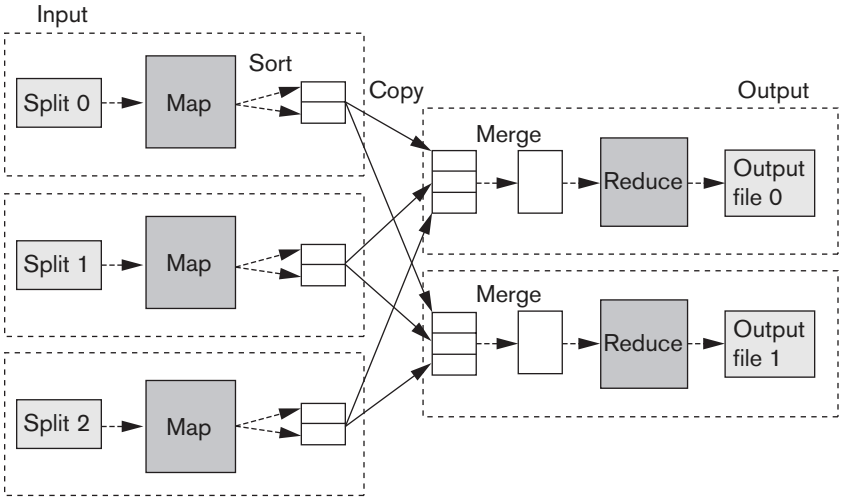

Figure 25.1
Overview of MapReduce execution. (Adapted from T. White, 2012)

Then the above lists of (word, 1) pairs are added up to output total counts of all words found in the document as follows:

```
Reduce (String key, Iterator values) : // here the key is a word and values are
lists of its counts //
    Int result =0;
    For each v in values :
        result += Parseint (v);
    Emit (key, Asstring (result));
```

The above example in MapReduce programming appears as:

```
map[LongWritable,Text](key, value) : List[Text, LongWritable] = {
    String[] words = split(value)
    for(word : words) {
        context.out(Text(word), LongWritable(1))
    }
}
reduce[Text, Iterable[LongWritable]](key, values) : List[Text, LongWritable] = {
    LongWritable c = 0
    for( v : values) {
        c += v
    }
    context.out(key,c)
    }
```

The data types used in the above example are LongWritable and Text. Each MapReduce job must register a Map and Reduce function. The Map function receives each key-value pair and on each call can output 0 or more key-value pairs. The signature of the Map function specifies the data types of its input and output

key-value pairs. The Reduce function receives a key and an iterator of values associated with that key. It can output one or more key-value pairs on each invocation. Again, the signature of the Reduce function indicates the data types of its inputs and outputs. The output type of the Map must match the input type of the Reduce function. In the wordcount example, the map function receives each line as a value, splits it into words, and emits (via the function context.out) a row for each word with frequency 1. Each invocation of the Reduce function receives for a given word the list of frequencies computed on the Map side. It adds these and emits each word and its frequency as output. The functions interact with a *context.* The context is used to interact with the framework. It is used by clients to send configuration information to tasks; and tasks can use it to get access to HDFS and read data directly from HDFS, to output key-value pairs, and to send status (e.g., task counters) back to the client.

The MapReduce way of implementing some other functions based on Dean and Ghemawat (2004) is as follows:

Distributed Grep

Grep looks for a given pattern in a file. The Map function emits a line if it matches a supplied pattern. The Reduce function is an identity function that copies the supplied intermediate data to the output. This is an example of a *Map only task*; there is no need to incur the cost of a **Shuffle**. We will provide more information when we explain the MapReduce runtime.

Reverse Web-Link Graph

The purpose here is to output (target URL, source URL) pairs for each link to a target page found in a page named source. The Reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair <target, list(source)>.

Inverted Index

The purpose is to build an inverted index based on all words present in a document repository. The Map function parses each document and emits a sequence of (word, document_id) pairs. The Reduce function takes all pairs for a given word, sorts them by document_id and emits a (word, list (document_id)) pair. The set of all these pairs forms an inverted index.

These illustrative applications give a sense of the MapReduce programming model's broad applicability and the ease of expressing the application's logic using the Map and Reduce phases.

A **Job** in MapReduce comprises the code for the Map and Reduce (usually packaged as a jar) phases, a set of artifacts needed to run the tasks (such as files, other jars, and archives) and, most importantly, a set of properties specified in a configuration. There are hundreds of properties that can be specified, but the core ones are as follows:

- the Map task
- the Reduce task

- the Input that the Job is to run on: typically specified as an HDFS path(s)
- the Format(Structure) of the Input
- the Output path
- the Output Structure
- the Reduce-side parallelism

A Job is submitted to the **JobTracker**, which then schedules and manages the execution of the Job. It provides a set of interfaces to monitor running Jobs. See the Hadoop Wiki[13] for further details about the workings of the JobTracker.

25.2.3 Hadoop Releases

Since the advent of Hadoop as a new distributed framework to run MapReduce programs, various releases have been produced:

The 1.x releases of Hadoop are a continuation of the original 0.20 code base. Subreleases with this line have added Security, additional HDFS and MapReduce improvements to support HBase, a better MR programming model, as well as other improvements.

The 2.x releases include the following major features:

- YARN (Yet Another Resource Navigator) is a general resource manager extracted out of the JobTracker from MR version1.
- A new MR runtime that runs on top of YARN.
- Improved HDFS that supports federation and increased availability.

At the time of this writing, Hadoop 2.0 has been around for about a year. The adoption is rapidly picking up; but a significant percentage of Hadoop deployments still run on Hadoop v1.

25.3 Hadoop Distributed File System (HDFS)

As we said earlier, in addition to MapReduce, the other core component of Hadoop is the underlying file system HDFS. In this section, we will first explain the architecture of HDFS, then describe the file input/output operations supported in HDFS, and finally comment on the scalability of HDFS.

25.3.1 HDFS Preliminaries

The Hadoop Distributed File System (HDFS) is the file system component of Hadoop and is designed to run on a cluster of commodity hardware. HDFS is patterned after the UNIX file system; however, it relaxes a few POSIX (portable operating system interface) requirements to enable streaming access to file system data. HDFS provides high-throughput access to large datasets. HDFS stores file system

[13]Hadoop Wiki is at http://hadoop.apache.org/

metadata and application data separately. Whereas the metadata is stored on a dedicated server, called the NameNode, the application data is stored on other servers, called DataNodes. All servers are fully connected and communicate with each other using TCP-based protocols. To make data durable, the file content is replicated on multiple DataNodes, as in the Google File System. This not only increases reliability, but it also multiplies the bandwidth for data transfer and enables colocation of computation with data. It was designed with the following assumptions and goals:

Hardware failure: Using commodity hardware, failure of hardware is the norm rather than an exception. Therefore, with thousands of nodes, automatic detection and recovery from failures becomes a must.

Batch processing: HDFS has been primarily designed for batch rather than interactive use. High throughput is emphasized over low latency of data access. Full scans of files are typical.

Large datasets: HDFS was designed to support huge files in the hundreds of gigabytes to terabytes range.

Simple coherency model: HDFS applications need a one writer and many reader access models for files. File content cannot be updated, but only appended. This model alleviates coherency issues among copies of data.

25.3.2 Architecture of HDFS

HDFS has a master-slave architecture. The master server, called the **NameNode**, manages the file system storage area or namespace**;** Clients access the namespace through the Namenode. The slaves called **DataNodes** run on a cluster of commodity machines, usually one per machine. They manage the storage attached to the node that they run on. The namespace itself comprises Files and Directories. The Namenodes maintain *inodes* (index nodes) about File and Directories with attributes like ownership, permissions, creation and access times, and disk space quotas. Using inodes, the mapping of File blocks to DataNodes is determined. DataNodes are responsible for serving read and write requests from clients. DataNodes perform block creation, deletion, and replication operations as instructed by the NameNode. A cluster can have thousands of DataNodes and tens of thousands of HDFS clients simultaneously connected.

To read a file, a client first connects to the NameNode and obtains the locations of the data blocks in the file it wants to access; it then connects directly with the DataNodes that house the blocks and reads the data.

The architecture of HDFS has the following highlights:

1. HDFS allows a decoupling of metadata from data operations. Metadata operations are fast whereas data transfers are much slower. If the location of metadata and transfer of data are not decoupled, speed suffers in a distributed environment because data transfer dominates and slows the response.

2. Replication is used to provide reliability and high availability. Each block is replicated (default is three copies) to a number of nodes in the cluster. The highly contentious files like MapReduce job libraries would have a higher number of replicas to reduce network traffic.
3. The network traffic is kept to a minimum. For reads, clients are directed to the closest DataNode. As far as possible, a local file system read is attempted and involves no network traffic; the next choice is a copy on a node on the same rack before going to another rack. For writes, to reduce network bandwidth utilization, the first copy is written to the same node as the client. For other copies, travel across racks is minimized.

NameNode. The NameNode maintains an **image** of the file system comprising *i*-nodes and corresponding block locations. Changes to the file system are maintained in a Write-ahead commit log (see the discussion of Write-ahead logs in Chapter 22) called the **Journal**. Checkpoints are taken for purposes of recovery; they represent a persistent record of the image without the dynamic information related to the block placement. Block placement information is obtained from the DataNodes periodically as described below. During Restart, the image is restored to the last checkpoint and the journal entries are applied to that image. A new checkpoint and empty journal are created so that the NameNode can start accepting new client requests. The startup time of a NameNode is proportional to the Journal file's size. Merging the checkpoint with the Journal periodically reduces restart time.

Note that with the above architecture, it is catastrophic to have any corruption of the Checkpoint or the Journal. To guard against corruption, both are written to multiple directories on different volumes.

Secondary NameNodes. These are additional NameNodes that can be created to perform either the checkpointing role or a backup role. A Checkpoint node periodically combines existing checkpoint and journal files. In backup mode, it acts like another storage location for the Journal for the primary NameNode. The backup NameNode remains up-to-date with the file system and can take over on failure. In Hadoop V1, this takeover must be done manually.

DataNodes: Blocks are stored on a DataNode in the node's native file system. The NameNode directs clients to the DataNodes that contain a copy of the block they want to read. Each block has its representation in two files in the native file system: a file containing the data and a second file containing the metadata, which includes the checksums for the block data and the block's generation stamp. DataNodes and NameNodes do not communicate directly but via a so-called **heartbeat mechanism**, which refers to a periodic reporting of the state by the DataNode to the NameNode; the report is called a Block Report. The report contains the block id, the generation stamp, and the length for each block. The block locations are not part of the namespace image. They must be obtained from the block reports, and they change as blocks are moved around. The MapReduce Job Tracker, along with the

NameNode, uses the latest block report information for scheduling purposes. In response to a heartbeat from the DataNode, the NameNode sends one of the following types of commands to the DataNode:

- Replicate a block to another node.
- Remove a block replica.
- Reregister the node or shut down the node.
- Send an immediate block report.

25.3.3 File I/O Operations and Replica Management in HDFS

HDFS provides a single-writer, multiple-reader model. Files cannot be updated, but only appended. A file consists of blocks. Data is written in 64-KB packets in a **write pipeline**, which is set up to minimize network utilization, as we described above. Data written to the last block becomes available only after an explicit hflush operation. Simultaneous reading by clients is possible while data is being written. A checksum is generated and stored for each block and is verified by the client to detect corruption of data. Upon detection of a corrupt block, the Namenode is notified; it initiates a process to replicate the block and instructs the Datanode to remove the corrupt block. During the read operation, an attempt is made to fetch a replica from as close a node as possible by ordering the nodes in ascending order of distance from the client. A read fails when the Datanode is unavailable, when the checksum test fails, or when the replica is no longer on the Datanode. HDFS has been optimized for batch processing similar to MapReduce.

Block Placement. Nodes of a Hadoop cluster are typically spread across many racks. They are normally organized such that nodes on a rack share a switch, and rack switches are connected to a high-speed switch at the upper level. For example, the rack level may have a 1-Gb switch, whereas at the top level there may be a 10-Gb switch. HDFS estimates the network bandwidth between Datanodes based on their distance. Datanodes on the same physical node have a distance of 0, on the same rack are distance 2 away, and on different racks are distance 4 away. The default HDFS block placement policy balances between minimizing the write cost and maximizing data reliability and availability as well as aggregate read bandwidth. Network bandwidth consumed is estimated based on distance among DataNodes. Thus, for DataNodes on the same physical node, the distance is 0, whereas on the same rack it is 2 and on a different rack it is 4. The ultimate goal of block placement is to minimize the write cost while maximizing data availability and reliability as well as available bandwidth for reading. Replicas are managed so that there is at least one on the original node of the client that created it, and others are distributed among other racks. Tasks are preferred to be run on nodes where the data resides; three replicas gives the scheduler enough leeway to place tasks where the data is.

Replica Management. Based on the block reports from the DataNodes, the NameNode tracks the number of replicas and the location of each block. A replication priority queue contains blocks that need to be replicated. A background thread

monitors this queue and instructs a DataNode to create replicas and distribute them across racks. NameNode prefers to have as many different racks as possible to host replicas of a block. Overreplicated blocks cause some replicas to be removed based on space utilization of the DataNodes.

25.3.4 HDFS Scalability

Since we are discussing big data technologies in this chapter, it is apropos to discuss some limits of scalability in HDFS. Hadoop program management committee member Shvachko commented that the Yahoo HDFS cluster had achieved the following levels as opposed to the intended targets (Shvachko, 2010). The numbers in parentheses are the targets he listed. Capacity: 14 petabytes (vs. 10 petabytes); number of nodes: 4,000 (vs. 10,000); clients:15,000 (vs. 100,000); and files: 60 million (vs. 100 million). Thus, Yahoo had come very close to its intended targets in 2010, with a smaller cluster of 4,000 nodes and fewer clients; but Yahoo had actually exceeded the target with respect to total amount of data handled.

Some of the observations made by Shvachko (2010) are worth mentioning. They are based on the HDFS configuration used at Yahoo in 2010. We present the actual and estimated numbers below to give the reader a sense of what is involved in these gigantic data processing environments.

- The blocksize used was 128K, and an average file contained 1.5 blocks. NameNode used about 200 bytes per block and an additional 200 bytes for an *i*-node. 100 million files referencing 200 million blocks would require RAM capacity exceeding 60 GB.
- For 100 million files with size of 200 million blocks and a replication factor of 3, the disk space required is 60 PB. Thus a rule of thumb was proposed that 1 GB of RAM in NameNode roughly corresponds to 1 PB of data storage based on the assumption of 128K blocksize and 1.5 blocks per file.
- In order to hold 60 PB of data on a 10,000-node cluster, each node needs a capacity of 6 TB. This can be achieved by having eight 0.75-TB drives.
- The internal workload for the NameNode is block reports. About 3 reports per second containing block information on 60K blocks per report were received by the NameNode.
- The external load on the NameNode consisted of external connections and tasks from MapReduce jobs. This resulted in tens of thousands of simultaneous connections.
- The Client Read consisted of performing a block lookup to get block locations from the NameNode, followed by accessing the nearest replica of the block. A typical client (the Map job from an MR task) would read data from 1,000 files with an average reading of half a file each, amounting to 96 MB of data. This was estimated to take 1.45 seconds. At that rate, 100,000 clients would send 68,750 block-location requests per second to the NameNode. This was considered to be well within the capacity of the NameNode, which was rated at handling 126K requests per second.

- The write workload: Given a write throughtput of 40 MB/sec, an average client writes 96 MB in 2.4 sec. That creates over 41K "create block" requests from 100,000 nodes at the NameNode. This was considered far above the NameNode capacity.

The above analysis assumed that there was only one task per node. In reality, there could be multiple tasks per node as in the real system at Yahoo, which ran 4 MapReduce (MR)tasks per node. The net result was a bottleneck at the NameNode. Issues such as these have been handled in Hadoop v2, which we discuss in the next section.

25.3.5 The Hadoop Ecosystem

Hadoop is best known for the MapReduce programming model, its runtime infrastructure, and the Hadoop Distributed File System (HDFS). However, the Hadoop ecosystem has a set of related projects that provide additional functionality on top of these core projects. Many of them are top-level open source Apache projects and have a very large contributing user community of their own. We list a few important ones here:

Pig and Hive: These provide a higher level interface for working with the Hadoop framework.

- Pig provides a dataflow language. A script written in PigScript translates into a directed acyclic graph (DAG) of MapReduce jobs.
- Hive provides an SQL interface on top of MapReduce. Hive's SQL support includes most of the SQL-92 features and many of the advanced analytics features from later SQL standards. Hive also defines the SerDe (Serialization/ Deserialization) abstraction, which defines a way of modeling the record structure on datasets in HDFS beyond just key-value pairs. We will discuss both of these in detail in Section 25.4.4.

Oozie: This is a service for scheduling and running workflows of Jobs; individual steps can be MR jobs, Hive queries, Pig scripts, and so on.

Sqoop: This is a library and a runtime environment for efficiently moving data between relational databases and HDFS.

HBase: This is a column-oriented key-value store that uses HDFS as its underlying store. (See Chapter 24 for a more detailed discussion of HBase.) It supports both batch processing using MR and key-based lookups. With proper design of the key-value scheme, a variety of applications are implemented using HBase. They include time series analysis, data warehousing, generation of cubes and multi-dimensional lookups, and data streaming.

25.4 MapReduce: Additional Details

We introduced the MapReduce paradigm in Section 25.2.2. We now elaborate further on it in terms of the MapReduce runtime. We discuss how the relational operation of join can be handled using MapReduce. We examine the high-level interfaces of Pig and Hive. Finally, we discuss the advantages of the combined MapReduce/Hadoop.

25.4.1 MapReduce Runtime

The purpose of this section is to give a broad overview of the MapReduce runtime environment. For a detailed description, the reader is encouraged to consult White (2012). MapReduce is a master-slave system that usually runs on the same cluster as HDFS. Typically, medium to large Hadoop clusters consist of a two- or three-level architecture built with rack-mounted servers.

JobTracker. The master process is called the *JobTracker*. It is responsible for managing the life cycle of Jobs and scheduling Tasks on the cluster. It is responsible for:

- Job submission, initializing a Job, providing Job status and state to both clients and TaskTrackers (the slaves), and Job completion.
- Scheduling Map and Reduce tasks on the cluster. It does this using a pluggable Scheduler.

TaskTracker. The slave process is called a *TaskTracker*. There is one running on all **Worker nodes** of the cluster. The Map-Reduce tasks run on Worker nodes. TaskTracker daemons running on these nodes register with the JobTracker on startup. They run tasks that the JobTracker assigns to them. Tasks are run in a separate process on the node; the life cycle of the process is managed by the TaskTracker. The TaskTracker creates the task process, monitors its execution, sends periodic status heartbeats to the JobTracker, and under failure conditions can kill the process at the request of the JobTracker. The TaskTracker provides services to the Tasks, the most important of which is the **Shuffle**, which we describe in a subsection below.

A. Overall flow of a MapReduce Job

A MapReduce job goes through the processes of Job Submission, Job Initialization, Task Assignment, Task Execution, and finally Job Completion. The Job Tracker and Task Tracker we described above are both involved in these. We briefly review them below.

Job submission A client submits a Job to the **JobTracker**. The Job package contains the executables (as a jar), any other components (files, jars archives) needed to execute the Job, and the InputSplits for the Job.

Job initialization The JobTracker accepts the Job and places it on a Job Queue. Based on the input splits, it creates map tasks for each split. A number of reduce tasks are created based on the Job configuration.

Task assignment The JobTracker's scheduler assigns Task to the TaskTracker from one of the running Jobs. In Hadoop v1, TaskTrackers have a fixed number of slots for map tasks and for reduce tasks. The Scheduler takes the location information of the input files into account when scheduling tasks on cluster nodes.

Task execution Once a task has been scheduled on a slot, the TaskTracker manages the execution of the task: making all Task artifacts available to the

Task process, launching the Task JVM, monitoring the process and coordinating with the JobTracker to perform management operations like cleanup on Task exit, and killing Tasks on failure conditions. The TaskTracker also provides the *Shuffle Service* to Tasks; we describe this when we discuss the Shuffle Procedure below.

Job completion Once the last Task in a Job is completed, the JobTracker runs the Job cleanup task (which is used to clean up intermediate files in both HDFS and the local file systems of TaskTrackers).

B. Fault Tolerance in MapReduce

There are three kinds of failures: failure of the Task, failure of the TaskTracker, and failure of the JobTracker.

Task failure This can occur if the Task code throws a Runtime exception, or if the Java Virtual Machine crashes unexpectedly. Another issue is when the TaskTracker does not receive any updates from the Task process for a while (the time period is configurable). In all these cases the TaskTracker notifies the JobTracker that the Task has failed. When the JobTracker is notified of the failure, it will reschedule execution of the task.

TaskTracker failure A TaskTracker process may crash or become disconnected from the JobTracker. Once the JobTracker marks a TaskTracker as failed, any map tasks completed by the TaskTracker are put back on the queue to be rescheduled. Similarly, any map task or reduce task in progress on a failed TaskTracker is also rescheduled.

JobTracker failure In Hadoop v1, JobTracker failure is not a recoverable failure. The JobTracker is a Single Point of Failure. The JobTracker has to be manually restarted. On restart all the running jobs have to be resubmitted. This is one of the drawbacks of Hadoop v1 that have been addressed by the next generation of Hadoop MapReduce called YARN.

Semantics in the presence of failure When the user-supplied map and reduce operators are deterministic functions of their input values, the MapReduce system produces the same output as would have been produced by a nonfaulting sequential execution of the entire program. Each task writes its output to a private task directory. If the JobTracker receives multiple completions for the same Task, it ignores all but the first one. When a Job is completed, Task outputs are moved to the Job output directory.

C. The Shuffle Procedure

A key feature of the MapReduce (MR) programming model is that the reducers get all the rows for a given key together. This is delivered by what is called the MR **shuffle**. The shuffle is divided into the Map, Copy, and Reduce phases.

Map phase: When rows are processed in Map tasks, they are initially held in an in-memory buffer, the size of which is configurable (the default is 100 MB). A

background thread partitions the buffered rows based on the number of Reducers in the job and the *Partitioner.* The *Partitioner* is a pluggable interface that is asked to choose a Reducer for a given Key value and the number of reducers in the Job. The partitioned rows are sorted on their key values. They can further be sorted on a provided *Comparator* so that rows with the same key have a stable sort order. This is used for Joins to ensure that for rows with the same key value, rows from the same table are bunched together. Another interface that can be plugged in is the *Combiner* interface. This is used to reduce the number of rows output per key from a mapper and is done by applying a reduce operation on each Mapper for all rows with the same key. During the Map phase, several iterations of partitioning, sorting, and combining may happen. The end result is a single local file per reducer that is sorted on the Key.

Copy phase: The Reducers pull their files from all the Mappers as they become available. These are provided by the JobTracker in Heartbeat responses. Each Mapper has a set of listener threads that service Reducer requests for these files.

Reduce phase: The Reducer reads all its files from the Mappers. All files are merged before streaming them to the Reduce function. There may be multiple stages of merging, depending on how the Mapper files become available. The Reducer will avoid unnecessary merges; for example, the last N files will be merged as the rows are being streamed to the Reduce function.

D. Job Scheduling

The **JobTracker** in MR 1.0 is responsible for scheduling work on cluster nodes. Clients' submitted jobs are added to the Job Queue of the JobTracker. The initial versions of Hadoop used a FIFO scheduler that scheduled jobs sequentially as they were submitted. At any given time, the cluster would run the tasks of a single Job. This caused undue delays for short jobs like ad-hoc hive queries if they had to wait for long-running machine learning–type jobs. The wait times would exceed runtimes, and the throughput on the cluster would suffer. Additionally, the cluster also would remain underutilized. We briefly describe two other types of schedulers, called the Fair Scheduler and Capacity Scheduler, that alleviate this situation.

Fair Scheduler: The goal of Fair Scheduler is to provide fast response time to small jobs in a Hadoop shared cluster. For this scheduler, jobs are grouped into Pools. The capacity of the cluster is evenly shared among the Pools. At any given time the resources of the cluster are evenly divided among the Pools, thereby utilizing the capacity of the cluster evenly. A typical way to set up Pools is to assign each user a Pool and assign certain Pools a minimum number of slots.

Capacity Scheduler: The Capacity Scheduler is geared to meet the needs of large Enterprise customers. It is designed to allow multiple tenants to share resources of a large Hadoop cluster by allocating resources in a timely manner under a given set of capacity constraints. In large enterprises, individual departments are apprehensive of using one centralized Hadoop cluster for concerns

that they may not be able to meet the service-level agreements (SLAs) of their applications. The Capacity Scheduler is designed to give each tenant guarantees about cluster capacity using the following provisions:

- There is support for multiple queues, with hard and soft limits in terms of fraction of resources.
- Access control lists (ACLs) are used that determine who can submit, view, and modify the Jobs in a queue.
- Excess capacity is evenly distributed among active Queues.
- Tenants have usage limits; such limits prevent tenants from monopolizing the cluster.

25.4.2 Example: Achieving Joins in MapReduce

To understand the power and utility of the MapReduce programming model, it is instructive to consider the most important operation of relational algebra, called Join, which we introduced in Chapter 6. We discussed its use via SQL queries (Chapters 7 and 8) and its optimization (Chapters 18 and 19). Let us consider the problem of joining two relations *R(A, B)* with *S(B, C)* with the join condition *R.A = S.B*. Assume both tables reside on HDFS. Here we list the many strategies that have been devised to do equi-joins in the MapReduce environment.

Sort-Merge Join. The broadest strategy for performing a join is to utilize the Shuffle to partition and sort the data and have the reducers merge and generate the output. We can set up an MR job that reads blocks from both tables in the Map phase. We set up a *Partitioner* to hash partition rows from *R* and *S* on the value of the *B* column. The key output from the Map phase includes a table *tag*. So the key has the form (tag, (key)). In MR, we can configure a custom Sort for the Job's shuffle; the custom Sort sorts the rows that have the same key. In this case, we Sort rows with the same *B* value based on the tag. We give the smaller table a tag of *0* and the larger table a tag of *1*. So a reducer will see all rows with the same *B* value in the order: smaller table rows first, then larger table rows. The Reducer can buffer smaller table rows; once it starts to receive large table rows, it can do an in-memory cross-product with the buffered small table rows to generate the join output. The cost of this strategy is dominated by the shuffle cost, which will write and read each row multiple times.

Map-Side Hash Join. For the case when one of *R* or *S* is a small table that can be loaded in the memory of each task, we can have the Map phase operate only on the large table splits. Each Map task can read the entire small table and create an in-memory hash map based on *B* as the hash key. Then it can perform a hash join. This is similar to Hash Joins in databases. The cost of this task is roughly the cost of reading the large table.

Partition Join. Assume that both *R* and *S* are stored in such a way that they are partitioned on the join keys. Then all rows in each Split belong to a certain identifiable range of the domain of the join field, which is *B* in our example. Assume both *R* and *S* are stored as *p* files. Suppose file (*i*) contains rows such that (Value *B*)mod

$p = i$. Then we only need to join the ith file of \(R\) R with the corresponding ith file of S. One way to do this is to perform a variation of the Map-Side join we discussed above: have the Mapper handling the ith partition of the larger table read the ith partition from the smaller table. This strategy can be expanded to work even when the two tables do not have the same number of partitions. It is sufficient for one to be a multiple of the other. For example, if table A is divided into two partitions and table B is divided into four partitions, then partition 1 from table A needs to join with partitions 1 and 3 of B, and partition 2 of A needs to join with partitions 2 and 4 of B. The opportunity to perform Bucketed Join (see below) is also common: for example, assume R and S are outputs of previous sort-merge joins. The output of the sort-merge join is partitioned in the joining expressions. Further joining this dataset allows us to avoid a shuffle.

Bucket Joins. This is a combination of Map-Side and Partition Joins. In this case only one relation, say the right side relation, is Partitioned. We can then run Mappers on the left side relation and perform a Map Join against each Partition from the right side.

***N*-Way Map-Side Joins.** A join on $R(A, B, C, D)$, $S(B, E)$, and $T(C, F)$ can be achieved in one MR job provided the rows for a key for all small tables can be buffered in memory. The join is typical in Data Warehouses (see Chapter 29), where R is a fact table and S and T are dimension tables whose keys are B and C, respectively. Typically, in a Data Warehouse query filters are specified on Dimensional Attributes. Hence each Map task has enough memory to hold the hash map of several small Dimensional tables. As Fact table rows are being read into the Map task, they can be hash joined with all the dimensional tables that the Map task has read into memory.

Simple *N*-Way Joins. A join on $R(A, B)$, $S(B, C)$, and $T(B, D)$ can be achieved in one MR job provided the rows for a key for all small tables can be buffered in memory. Suppose R is a large table and S and T are relatively smaller tables. Then it is typically the case that for any given key value B, the number of rows in S or T will fit in a Task's memory. Then, by giving the large table the largest tag, it is easy to generalize the Sort-Merge join to an N-way join where the joining expressions are the same. In a Reducer for a key value of B, the reducer will first receive the S rows, then the T rows, and finally the R rows. Since the assumption is that there aren't a large number of S and T rows, the reducer can cache them. As it receives R rows, it can do a cross product with the cached S and T rows and output the result of join.

In addition to the above strategies for performing joins using the MapReduce paradigm, algorithms have been proposed for other types joins (e.g., the general multi-way natural join with special cases of chain-join or star-join in data warehouses have been shown to be handled as a single MR job).[14] Similarly, algorithms have been proposed to deal with skew in the join attributes (e.g., in a sales fact table, certain days may have a disproportionate number of transactions). For joins on attributes with skew, a modified algorithm would let the *Partitioner* assign unique values to the

[14]See Afrati and Ullman (2010).

data having a large number of entries and let them be handled by Reduce tasks, whereas the rest of the values may undergo hash partitioning as usual.

This discussion should provide the reader with a good sense of the many possibilities of implementing Join strategies on top of MapReduce. There are other factors affecting performance, such as row versus columnar storage and pushing predicates down to storage handlers. These are beyond our scope of discussion here. Interested readers will find ongoing research publications in this area that are similar to Afrati and Ullman (2010).

The purpose of this section is to highlight two major developments that have impacted the big data community by providing high-level interfaces on top of the core technology of Hadoop and MapReduce. We will give a brief overview of the language Pig Latin and the system Hive.

Apache Pig. Pig[15] was a system that was designed at Yahoo Research to bridge the gap between declarative-style interfaces such as SQL, which we studied in the context of the relational model, and the more rigid low-level procedural-style programming style required by MapReduce that we described in Section 25.2.2. Whereas it is possible to express very complex analysis in MR, the user must express programs as a one-input, two-stage (map and reduce) process. Furthermore, MR provides no methods for describing a complex data flow that applies a sequence of transformations on the input. There is no standard way to do common data transformation operations like Projections, Filtering, Grouping, and Joining. We saw all these operations being expressed declaratively in SQL in Chapters 7 and 8. However, there is a community of users and programmers that thinks more procedurally. So the developers of Pig invented the language Pig Latin to fill in the "sweet spot" between SQL and MR. We show an example of a simple Group By query expressed in Pig Latin in Olston et al. (2008):

There is a table of urls: (url,category.pagerank).

We wish to find, for categories having a large number of URLs, the average pagerank of the high-pagerank URLs in that category. This requires a grouping of URLs by category. The SQL query that expresses this requirement may look like:

```
SELECT category, AVG(pagerank)
FROM urls WHERE pagerank > 0.2
GROUP BY category HAVING COUNT(*) > 10**6
```

The same query in Pig Latin is written as:

```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls)> 10**6;
output = FOREACH big_groups GENERATE
         category, AVG(good_urls.pagerank);
```

[15]See Olston et al. (2008).

As shown by this example, a Pigscript written using the scripting language Pig Latin is a sequence of data transformation steps. On each step, a basic transformation like Filter, Group By, or Projection is expressed. The script resembles a query plan for the SQL query similar to the plans we discussed in Chapter 19. The language supports operating on nested data structures like JSON (Java Script Object Notation) and XML. It has an extensive and extendible function library, and also an ability to bind schema to data very late or not at all.

Pig was designed to solve problems such as ad hoc analyses of Web logs and clickstreams. The logs and clickstreams typically require custom processing at row level as well as at an aggregate level. Pig accommodates user-defined functions (UDFs) extensively. It also supports a nested data model with the following four types:

Atoms: Simple atomic values such as a number or a string

Tuples: A sequence of fields, each of which can be of any permissible type

Bag: A collection of tuples with possible duplicates

Map: A collection of data items where each item has a key that allows direct access to it

Olston et al. (2008) demonstrates interesting applications on logs using Pig. An example is analysis of activity logs for a search engine over any time period (day, week, month, etc.) to calculate frequency of search terms by a user's geographic location. Here the functions needed include mapping IP addresses to geo-locations and using *n*-gram extraction. Another application involves co-grouping search queries of one period with those of another period in the past based on search terms.

Pig was architected so that it could run on different execution environments. In implementing Pig, Pig Latin was compiled into physical plans that were translated into a series of MR jobs and run in Hadoop. Pig has been a useful tool for enhancing programmers' productivity in the Hadoop environment.

25.4.3 Apache Hive

Hive was developed at Facebook[16] with a similar intent—to provide a higher level interface to Hadoop using SQL-like queries and to support the processing of aggregate analytical queries that are typical in data warehouses (see Chapter 29). Hive remains a primary interface for accessing data in Hadoop at Facebook; it has been adopted widely in the open source community and is undergoing continuous improvements. Hive went beyond Pig Latin in that it provided not only a high-level language interface to Hadoop, but a layer that makes Hadoop look like a DBMS with DDL, metadata repository, JDBC/ODBC access, and an SQL compiler. The architecture and components of Hive are shown in Figure 25.2.

Figure 25.2 shows Apache Thrift as interface in Hive. Apache Thrift defines an Interface Definition Language (IDL) and Communication Protocol used to develop

[16]See Thusoo et al. (2010).

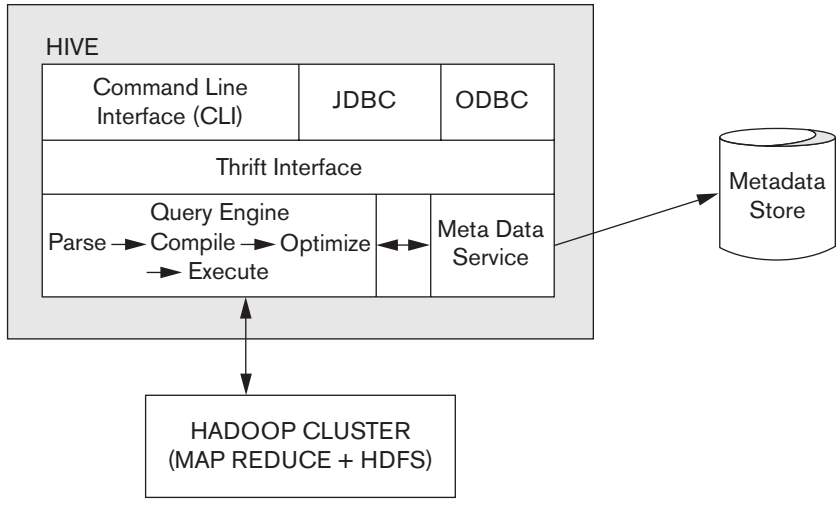

Figure 25.2
Hive system architecture and components.

remote services. It comes with a runtime and code generation engine that can be used to develop remote services in many languages, including Java, C++, Python, and Ruby. Apache Thrift supports JSON-based and binary protocols; it supports http, socket, and file transports.

The Hive query language HiveQL includes a subset of SQL that includes all types of joins, Group By operations, as well as useful functions related to primitive and complex data types. We comment below on some of the highlights of the Hive system.

Interfacing with HDFS:

- Tables in Hive are linked to directories in HDFS. Users can define partitions within tables. For example, a Web log table can be partitioned by day and within day by the hour. Each partition level introduces a level of directories in HDFS. A table may also be stored as *bucketed* on a set of columns. This means that the stored data is physically partitioned by the column(s). For example, within an hour directory, the data may be *bucketed* by Userid; this means that each hour's data is stored in a set of files, each file represents a bucket of Users, and the bucket is based on the hashing of the Userid column. Users can specify how many buckets the data should be divided into.
- The SerDe (Serialization/Deserialization) plugin architecture lets users specify how data in native file formats is exposed as rows to Hive SQL operators. Hive comes with a rich set of SerDe functions and supported File formats (e.g., CSV, JSON, SequenceFile); columnar formats (e.g., RCFile, ORCFile, Parquet); and support for Avro—another data serialization system. The different *StorageHandlers* expand on the SerDe mechanism to allow pluggable behavior for how data is read/written and the ability to push *predicates* down to the Storage Handler for early evaluation. For

example, the *JDBC StorageHandler* allows a Hive user to define a table that is in fact stored in some relational DBMS and accessed using the JDBC protocol (see Chapter 10) during query execution.

Support of SQL and Optimizations in Hive: Hive incorporated the concepts of Logical and Physical Optimizations similar to those used in optimization of SQL queries, which we discussed in Chapters 18 and 19. Early on, there was support for logical optimizations such as pruning unneeded columns and pushing selection predicates down into the query tree. Physical optimizations of converting sort-merge joins to Map-side joins based on user hints and data file sizes have also been incorporated. Hive started with support for a subset of SQL-92 that included SELECT, JOIN, GROUP BY, and filters based on conditions in the WHERE clause. Hive users can express complex SQL commands in Hive. Early in its development, Hive was able to run the 22 TPCH benchmark queries (Transaction Processing Performance Council benchmark for decision support), although with considerable manual rewriting.

Significant strides have been made in language support and in optimizer and runtime techniques. Here is a sampling of those improvements:

- Hive SQL has added many analytic features of SQL, such as subquery predicates, Common Table expressions (this is the WITH clause in SQL that allows users to name common subquery blocks and reference them multiple times in the query; these expressions can be considered query-level views), aggregates over a certain window within the data, Rollups (which refer to higher aggregation levels), and Grouping sets (this capability allows you to express multiple levels of aggregation in one Group By level). Consider, for example, Group By Grouping Sets ((year, month), (dayofweek)); this expresses aggregates both at the (Year, Month) level and also by DayOfWeek. A full set of SQL data types, including varchars, numeric types, and dates, is now supported. Hive also supports the common Change Data Capture ETL flow via Insert and Update statements. In a Data Warehouse, the process of delivering slowly changing Dimensions (e.g., customers in a Retail Data Warehouse) requires a complex dataflow of identifying new and updated records in that Dimension. This is called the Change Data Capture (CDC) process. By adding Insert and Update statements in Hive, it is possible to model and execute CDC processes in Hive SQL.
- Hive now has a greatly expanded set of DDLs for expressing grants and privileges in terms of discretionary access control (see Section 30.2).
- Several standard database optimizations have been incorporated, including Partition pruning, Join reordering, Index rewrite, and Reducing the number of MR jobs. Very large tables, like Fact tables in Data Warehouses, are typically partitioned. Time is probably the most common attribute used for partitioning. With HDFS being used as the storage layer, users tend to retain data for long time periods. But a typical Warehouse will only include the most current time periods (e.g., the last quarter or current year). The time periods are specified as filters in the Query. Partition Pruning is the technique of extracting relevant predicates from the Query filters and translating them to a list of

Table partitions that need to be read. Obviously, this has a huge impact on performance and cluster utilization: Instead of scanning all partitions retained for the last *N* years, only the partitions from the last few weeks/months are scanned. Work in progress includes collecting column- and table-level statistics and generating plans based on a cost model that uses these statistics (similar to what we considered for RDBMSs in Chapter 19).

- Hive now supports Tez as a runtime environment that has significant advantages over MR, including that there is no need to write to disk between jobs; and there is no restriction on one-input, two-stage processes. There is also active work to support Hive on Spark, a new technology that we briefly mention in Section 25.6.

25.4.4 Advantages of the Hadoop/MapReduce Technology

Hadoop version 1 was optimized for batch processing on very large datasets. Various factors contribute to its success:

1. The disk seek rate is a limiting factor when we deal with petabyte-level workloads. Seek is limited by the disk mechanical structure, whereas the transfer speed is an electronic feature and increasing steadily. (See Section 16.2 for a discussion of disk drives.) The MapReduce model of scanning datasets in parallel alleviates this situation. For instance, scanning a 100-TB dataset sequentially using 1 machine at a rate of 50 Mbps will take about 24 days to complete. On the other hand, scanning the same data using 1,000 machines in parallel will just take 35 minutes. Hadoop recommends very large block sizes, 64 MB or higher. So when scanning datasets, the percentage of time spent on disk seeks is negligible. Unlimited disk seek rates combined with processing large datasets in chunks and in parallel is what drives the scalability and speed of the MapReduce model.
2. The MapReduce model allows handling of semistructured data and key-value datasets more easily compared to traditional RDBMSs, which require a predefined schema. Files such as very large logfiles present a particular problem in RDBMSs because they need to be parsed in multiple ways before they can be analyzed.
3. The MapReduce model has linear scalability in that resources can be added to improve job latency and throughput in a linear fashion. The failure model is simple, and individual failed jobs can be rerun without a major impact on the whole job.

25.5 Hadoop v2 alias YARN

In previous sections, we discussed Hadoop development in detail. Our discussion included the core concepts of the MapReduce paradigm for programming and the HDFS underlying storage infrastructure. We also discussed high-level interfaces like Pig and Hive that are making it possible to do SQL-like, high level data processing on top of the Hadoop framework. Now we turn our attention to subsequent developments, which are broadly called Hadoop v2 or MRv2 or YARN (Yet Another

Resource Negotiator). First, we point out the shortcomings of the Hadoop v1 platform and the rationale behind YARN.

25.5.1 Rationale behind YARN

Despite the success of Hadoop v1, user experience with Hadoop v1 in enterprise applications highlighted some shortcomings and suggested that an upgrade of Hadoop v1 might be necessary:

- As cluster sizes and the number of users grew, the JobTracker became a bottleneck. It was always known to be the Single Point of Failure.
- With a static allocation of resources to map and reduce functions, utilization of the cluster of nodes was less than desirable
- HDFS was regarded as a single storage system for data in the enterprise. Users wanted to run different types of applications that would not easily fit into the MR model. Users tended to get around this limitation by running Map-only Jobs, but this only compounded scheduling and utilization issues.
- On large clusters, it became problematic to keep up with new open source versions of Hadoop, which were released every few months.

The above reasons explain the rationale for developing version 2 of Hadoop. Some of the points mentioned in the previous list warrant a more detailed discussion, which we provide next.

Multitenancy: Multitenancy refers to accommodating multiple tenants/users concurrently so that they can share resources. As the cluster sizes grew and the number of users increased, several communities of users shared the Hadoop cluster. At Yahoo, the original solution to this problem was **Hadoop on Demand**, which was based on the Torque resource manager and Maui scheduler. Users could set up a separate cluster for each Job or set of Jobs. This had several advantages:

- Each cluster could run its own version of Hadoop.
- JobTracker failures were isolated to a single cluster.
- Each user/organization could make independent decisions on the size and configuration of its cluster depending on expected workloads.

But Yahoo abandoned Hadoop on Demand for the following reasons:

- Resource allocation was not based on data locality. So most reads and writes from HDFS were remote accesses, which negated one of the key benefits of the MR model of mostly local data accesses.
- The allocation of a cluster was static. This meant large parts of a cluster were mostly idle:
  - Within an MR job, the reduce slots were not usable during the Map phase and the map slots were not usable during the Reduce phase. When using higher level languages like Pig and Hive, each script or query spawned multiple Jobs. Since cluster allocation was static, the maximum nodes needed in any Job had to be acquired upfront.

- Even with the use of Fair or Capacity scheduling (see our discussion in Section 25.4.2), dividing the cluster into fixed map and reduce slots meant the cluster was underutilized.

- The latency involved in acquiring a cluster was high—a cluster would be granted only when enough nodes were available. Users started extending the lifetime of clusters and holding the clusters longer than they needed. This affected cluster utilization negatively.

JobTracker Scalability. As the cluster sizes increased beyond 4,000 nodes, issues with memory management and locking made it difficult to enhance JobTracker to handle the workload. Multiple options were considered, such as holding data about Jobs in memory, limiting the number of tasks per Job, limiting the number of Jobs submitted per user, and limiting the number of concurrently running jobs. None of these seemed to fully satisfy all users; JobTracker often ran out of memory.

A related issue concerned completed Jobs. Completed jobs were held in JobTracker and took up memory. Many schemes attempted to reduce the number and memory footprint of completed Jobs. Eventually, a viable solution was to offload this function to a separate Job History daemon.

As the number of TaskTrackers grew, the latencies for heartbeats (signals from TaskTracker to JobTracker) were almost 200 ms. This meant that heartbeat intervals for TaskTrackers could be 40 seconds or more when there were more than 200 task trackers in the cluster. Efforts were made to fix this but were eventually abandoned.

JobTracker: Single Point of Failure. The recovery model of Hadoop v1 was very weak. A failure of JobTracker would bring down the entire cluster. In this event, the state of running Jobs was lost, and all jobs would have to be resubmitted and JobTracker restarted. Efforts to make the information about completed jobs persist did not succeed. A related issue was to deploy new versions of the software. This required scheduling a cluster downtime, which resulted in backlogs of jobs and a subsequent strain on JobTracker upon restart.

Misuse of the MapReduce Programming Model. MR runtime was not a great fit for iterative processing; this was particularly true for machine learning algorithms in analytical workloads. Each iteration is treated as an MR job. Graph algorithms are better expressed using a bulk synchronous parallel (BSP) model, which uses message passing as opposed to the Map and Reduce primitives. Users got around these impediments by inefficient alternatives such as implementing machine learning algorithms as long-running Map-only jobs. These types of jobs initially read data from HDFS and executed the first pass in parallel; but then exchanged data with each other outside the control of the framework. Also, the fault tolerance was lost. The JobTracker was not aware of how these jobs operated; this lack of awareness led to poor utilization and instability in the cluster.

Resource Model Issues. In Hadoop v1, a node is divided into a fixed number of Map and Reduce slots. This led to cluster underutilization because idle slots could

not be used. Jobs other than MR could not run easily on the nodes because the node capacity remained unpredictable.

The aforementioned issues illustrate why Hadoop v1 needed upgrading. Although attempts were made to fix in Hadoop v1 many of the issues listed above, it became clear that a redesign was needed. The goals of the new design were set as follows:

- To carry forward the scalibility and locality awareness of Hadoop v1.
- To have multitenancy and high cluster utilization.
- To have no single point of failure and to be highly available.
- To support more than just MapReduce jobs. The cluster resources should not be modeled as static map and reduce slots.
- To be backward compatible, so existing jobs should run as they are and possibly without any recompilation.

The outcome of these was YARN or Hadoop v2, which we discuss in the next section.

25.5.2 YARN Architecture

Overview. Having provided the motivation behind upgrading Hadoop v1, we now discuss the detailed architecture of the next generation of Hadoop, which is popularly known as MRv2, MapReduce 2.0, Hadoop v2, or YARN.[17] The central idea of YARN is the separation of cluster Resource Management from Jobs management. Additionally, YARN introduces the notion of an *ApplicationMaster*, which is now responsible for managing work (task data flows, task lifecycles, task failover, etc.). MapReduce is now available as a service/application provided by the *MapReduce ApplicationMaster*. The implications of these two decisions are far-reaching and are central to the notion of a data service operating system. Figure 25.3 shows a high-level schematic diagram of Hadoop v1 and Hadoop v2 side by side.

The *ResourceManager* and the per worker node *NodeManager* together form the platform on which any Application can be hosted on YARN. The ResourceManager manages the cluster, doling out Resources based on a pluggable scheduling policy (such as a fairness policy or optimizing cluster utilization policy). It is also responsible for the lifecycle of nodes in the cluster in that it will track when nodes go down, when nodes become unreachable, or when new nodes join. Node failures are reported to the ApplicationMasters that had containers on the failed node. New nodes become available for use by ApplicationMasters.

ApplicationMasters send ResourceRequests to the ResourceManager which then responds with cluster Container leases. A **Container** is a lease by the ResourceManager to the ApplicationManager to use certain amount of resources on a node

[17] See the Apache website: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html for up-to-date documentation on YARN.

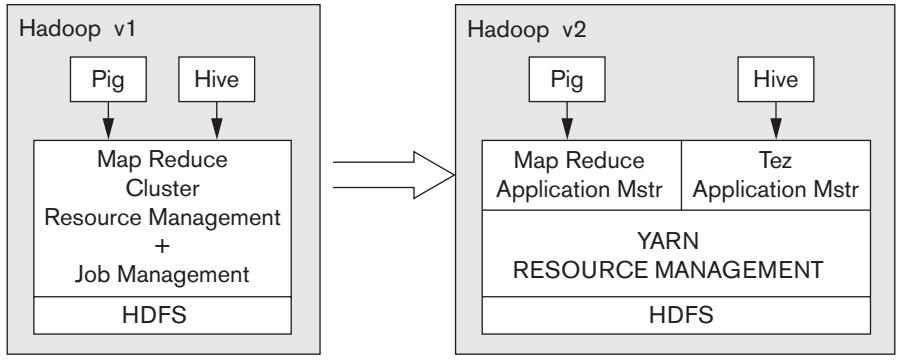

Figure 25.3
The Hadoop v1 vs. Hadoop v2 schematic.

of the cluster. The ApplicationMaster presents a Container Launch Context to the NodeManager for the node that this lease references. The Launch Context, in addition to containing the lease, also specifies how to run the process for the task and how to get any resources like jars, libs for the process, environment variables, and security tokens. A node has a certain processing power in terms of number of cores, memory, network bandwidth, etc. Currently, YARN only considers memory. Based on its processing power, a node can be divided into an interchangeable set of containers. Once an ApplicationMaster receives a container lease, it is free to schedule work on it as it pleases. ApplicationMasters, based on their workload, can continuously change their Resource requirements. The ResourceManager bases its scheduling decisions purely on these requests, on the state of the cluster, and on the cluster's scheduling policy. It is not aware of the actual tasks being carried out on the nodes. The responsibility of managing and analyzing the actual work is left to ApplicationMasters.

The *NodeManager* is responsible for managing Containers on their nodes. Containers are responsible for reporting on the node health. They also handle the procedure for nodes joining the cluster. Containers provide the Container Launch service to ApplicationMasters. Other services available include a Local cache, which could be User level, Application level, or Container level. Containers also can be configured to provide other services to Tasks running on them. For example, for MR tasks, the shuffle is now provided as a Node-level service.

The *ApplicationMaster* is now responsible for running jobs on the cluster. Based on their job(s) the clusters negotiate for Resources with the ResourceManager. The *ApplicationMaster* itself runs on the cluster; at startup time a client submits an Application to the ResourceManager, which then allocates a container for the ApplicationMaster and launches it in that container. In the case of MR, the ApplicationMaster takes over most of the tasks of the JobTracker: it launches Map and Reduce tasks, makes decisions on their placement, manages failover of tasks, maintains counters similar to Job state counters, and provides a monitoring interface for running Jobs. The management and interface for completed jobs has been moved to a separate Job History Server.

The following advantages accrue from the separation of Resource Management from Application Management in the YARN architecture:

- A rich diversity of Data Services is available to utilize the cluster. Each of these can expose its own programming model.
- Application Masters are free to negotiate resources in patterns that are optimized for their work: for example, machine learning Apps may hold Containers for long durations.
- The Resource and Container model allows nodes to be utilized in a dynamic manner, which increases the overall utilization of the cluster.
- The ResourceManager does only one thing—manage resources; hence it is highly scalable to tens of thousands of nodes.
- With ApplicationMasters managing Jobs, it is possible to have multiple versions of an Application running on the cluster. There is no need for a global cluster update, which would require that all Jobs be stopped.

Failure of an ApplicationMaster affects only Jobs managed by it. The ResourceManager provides some degree of management of ApplicationMasters. Let us briefly consider each of the components of the YARN environment.

Resource Manager (RM). The Resource Manager is only concerned with allocating resources to Applications, and not with optimizing the processing within Applications. The policy of resource allocation is pluggable. Application Masters are supposed to request resources that would optimize their workload.

The Resource Manager exposes the following interfaces:

1. An API for clients to start ApplicationMasters
2. A protocol for ApplicationMasters to negotiate for cluster resources
3. A protocol for NodeManagers to report on node resources and be managed by the Resource Manager

The scheduler in the ResourceManager matches the Resource Requirements submitted by Applications against the global state of the cluster resources. The allocation is based on the policies of the pluggable Scheduler (such as capacity or fairness). Resources are requested by ApplicationMasters as **Resource Requests**. A Resource Request specifies:

- The number of containers needed
- The physical resources (CPU, memory) needed per container
- The locality preferences (physical node, rack) of the containers
- The priority of the request for the Application

The scheduler satisfies these requests based on the state of the cluster as reported by the NodeManager heartbeats. The locality and priority guides the scheduler toward alternatives: for example, if a requested node is busy, the next best alternative is another node on the same rack.

The scheduler also has the ability to request resources back from an Application if needed and can even take back the resources forcibly. Applications, in returning a container, can migrate the work to another container, or checkpoint the state and restore it on another container. It is important to point out what the Resource manager is *not* responsible for: handling the execution of tasks within an application, providing any status information about applications, providing history of finished jobs, and providing any recovery for failed tasks.

ApplicationMaster (AM). The ApplicationMaster is responsible for coordinating the execution of an Application on the cluster. An Application can be a set of processes like an MR Job, or it can be a long-running service like a Hadoop on demand (HOD) cluster serving multiple MR jobs. This is left to the Application Writer.

The ApplicationMaster will periodically notify the ResourceManager of its current Resource Requirements through a heartbeat mechanism. Resources are handed to the ApplicationMaster as Container leases. Resources used by an Application are dynamic: they are based on the progress of the application and the state of the cluster. Consider an example: the MR ApplicationMaster running an MR job will ask for a container on each of the *m* nodes where an InputSplit resides. If it gets a container on one of the nodes, the ApplicationMaster will either remove the request for containers on the rest of the *m-1* nodes or at least reduce their priority. On the other hand, if the map task fails, it is AM that tracks this failure and requests containers on other nodes that have a replica of the same InputSplit.

NodeManager. A NodeManager runs on every worker node of the cluster. It manages Containers and provides pluggable services for Containers. Based on a detailed *Container Launch Context* specification, a NodeManager can launch a process on its node with the environment and local directories set up. It also monitors to make sure the resource utilization does not exceed specifications. It also periodically reports on the state of the Containers and the node health. A NodeManager provides local services to all Containers running on it. The *Log Aggregation* service is used to upload each task's standard output and standard error (stdout and stderr) to HDFS. A NodeManager may be configured to run a set of pluggable *auxillary services*. For example, the MR Shuffle is provided as a NodeManager service. A Container running a Map task produces the Map output and writes to local disk.The output is made available to Reducers of the Job via the Shuffle service running on the Node.

Fault tolerance and availability. The RM remains the single point of failure in YARN. On restart, the RM can recover its state from a persistent store. It kills all containers in the cluster and restarts each ApplicationMaster. There is currently a push to provide an active/passive mode for RMs. The failure of an ApplicationMaster is not a catastrophic event; it only affects one Application. It is responsible for recovering the state of its Application. For example, the MR ApplicationMaster will recover its completed task and rerun any running tasks.

Failure of a Container because of issues with the Node or because of Application code is tracked by the framework and reported to the ApplicationMaster. It is the responsibility of the ApplicationMaster to recover from the failure.

25.5.3 Other Frameworks on YARN

The YARN architecture described above has made it possible for other application frameworks to be developed as well as other programming models to be supported that can provide additional services on the shared Hadoop cluster. Here we list some of the Frameworks that have become available in YARN at the time this text was written.

Apache Tez. Tez is an extensible framework being developed at Hortonworks for building high-performance applications in YARN; these applications will handle large datasets up to petabytes. Tez allows users to express their workflow as a directed acyclic graph (DAG) of tasks. Jobs are modeled as DAGs, where Vertices are tasks or operations and Edges represent interoperation dependencies or flows of data. Tez supports the standard dataflow patterns like pipeline, scatter-gather, and broadcast. Users can specify the concurrency in a DAG, as well as the failover characteristics, such as whether to store task output in persistent storage or to recompute it. The DAG can be changed at runtime based on job and cluster state. The DAG model is a more natural fit (than executing as one or more MapReduce jobs) for Pig scripts and SQL physical plans. Both Hive and Pig now provide a mode in which they run on Tez. Both have benefitted in terms of simpler plans and significant performance improvements. An often cited performance optimization is the Map-Reduce-Reduce pattern; an SQL query that has a Join followed by a Group-By normally is translated to two MR jobs: one for the Join and one for the Group-By. In the first MR stage, the output of the join will be written to HDFS and read back in the Map phase of the second MR for the Group-By Job. In Tez, this extra write and read to/from HDFS can be avoided by having the Join Vertex of the DAG stream resulting rows to the Group-By Vertex.

Apache Giraph. Apache Giraph is the open source implementation of Google's Pregel system,[18] which was a large-scale graph processing system used to calculate Page-Rank. (See Section 27.7.3 for a definition of Page-Rank.) Pregel was based on the bulk synchronous processing (BSP) model of computation.[19] Giraph added several features to Pregel, including sharded aggregators (sharding, as defined in Chapter 24, refers to a form of partitioning) and edge-oriented input. The Hadoop v1 version of Giraph ran as MR jobs, which was not a very good fit. It did this by running long-running Map-only Jobs. On YARN, the Giraph implementation exposes an iterative processing model. Giraph is currently used at Facebook to analyze the social network users' graph, which has users as nodes and their connections as edges; the current number of users is approximately 1.3 billion.

Hoya: HBase on YARN. The Hortonworks Hoya (HBase on YARN) project provides for elastic HBase clusters running on YARN with the goal of more flexibility and improved utilization of the cluster. We discussed HBase in Section 24.5 as a

[18]Pregel is described in Malewicz et al. (2010).

[19]BSP is a model for designing parallel algorithms and was originally proposed by Valiant (1990).

distributed, open source, nonrelational database that manages tables with billions of rows and millions of columns. HBase is patterned after BigTable from Google[20] but is implemented using Hadoop and HDFS. Hoya is being developed to address the need for creating on-demand clusters of HBase, with possibly different versions of HBase running on the same cluster. Each of the HBase instances can be individually configured. The Hoya ApplicationMaster launches the HBase Master locally. The Hoya AM also asks the YARN RM for a set of containers to launch HBase RegionServers on the cluster. HBase RegionServers are the worker processes of Hbase; each ColumnFamily (which is like a set of Columns in a relational table) is distributed across a set of RegionServers. This can be used to start one or more HBase instances on the cluster, on demand. The clusters are elastic and can grow or shrink based on demand.

The above three examples of the applications developed on YARN should give the reader a sense of the possibilities that have been opened up by the decoupling of Resource Management from Application Management in the overall Hadoop/MapReduce architecture by YARN.

25.6 General Discussion

So far, we have discussed the big data technology development that has occurred roughly in the 2004–2014 time frame, and we have emphasized Hadoop v1 and YARN (also referred to as Hadoop v2 or MRv2). In this section, we must first state the following disclaimer: there are a number of ongoing projects under Apache open source banner as well as in companies devoted to developing products in this area (e.g., Hortonworks, Cloudera, MapR) as well as many private startup companies. Similarly, the Amplab at University of California and other academic institutions are contributing heavily to developing technology that we have not been able to cover in detail. There is also a series of issues associated with the cloud concept, with running MapReduce in the cloud environment, and with data warehousing in the cloud that we have not discussed. Given this background, we now cover a few general topics that are worth mentioning in the context of the elaborate descriptions we presented so far in this chapter. We present issues related to the tussle between the traditional approach to high performance applications in parallel RDBMS implementations vis-à-vis Hadoop- and YARN-based technologies. Then we present a few points related to how big data and cloud technologies will be complementary in nature. We outline issues related to the locality of data and the optimization issues inherent in the storage clouds and the compute clouds. We also discuss YARN as a data services platform and the ongoing movement to harness big data for analytics. Finally, we present some current challenges facing the entire big data movement.

25.6.1 Hadoop/MapReduce vs. Parallel RDBMS

A team of data experts, including Abadi, DeWitt, Madden, and Stonebracker, have done a methodological study comparing a couple of parallel database systems with

[20]BigTable is described in Chang et al. (2006).

the open source version of Hadoop/MR (see, for example, Pavlo et al. (2009)). These experts measure the performance of these two approaches on the same benchmark using a 100-node cluster. They admit that the parallel database took longer to load and tune compared to MR, but the performance of parallel DBMSs was "strikingly better." We list the areas the experts compared in the study and attempt to show the progress made in both DBMSs and Hadoop since then.

Performance. In their paper, Pavlo et al. concluded that parallel DBMSs were three to six times faster than MR. The paper lists many reasons why the DBMSs gave better performance. Among the reasons given are the following: (i) indexing with B^+-trees, which expedites selection and filtering; (ii) novel storage orientation (e.g., column-based storage has certain advantages); (iii) techniques that allow operations on compressed data directly; and (iv) parallel query optimization techniques common in parallel DBMSs.

Since the time of Pavlo et al.'s comparison, which involved Hadoop version 0.19, huge strides have been made in the MR runtime, the storage formats, and the planning capabilities for job scheduling and for optimizing complex data flows in the Hadoop ecosystem. *ORC* and *Parquet* file formats are sophisticated Columnar file formats that have the same aggressive compression techniques, the ability to push predicates to the storage layer, and the ability to answer aggregate queries without scanning data. We will briefly talk about the improvements in HDFS and MR; Apache Hive has made huge strides in both the runtime and Cost-based optimizations of complex SQLs. In their move to transform Hadoop from batch into real-time and interactive query mode, Hortonworks (2014) reports orders-of-magnitude gains in performance of queries on a TPC-DS (decision support)–style benchmark. Cloudera's Impala product, as reported in Cloudera (2014), uses Parquet (the open source columnar data format) and is claimed to perform comparably to traditional RDBMSs.

Upfront Cost advantage. Hadoop has maintained its cost advantage. With few exceptions, Hadoop continues to be primarily an open source platform. YARN, Hive, and Spark are all developed as Apache projects and are available as freely downloadable packages.

Handling Unstructured/Semistructured data. MR reads data by applying the schema definition to it; doing so allows it to handle semistructured datasets like CSVS, JSON, and XML documents. The loading process is relatively inexpensive for the Hadoop/MR systems. However, the support for unstructured data is definitely on the rise in RDBMSs. PostgreSQL now supports key-value stores and json; most RDBMSs have a support for XML. On the other hand, one of the reasons for the performance gains on the Hadoop side has been the use of specialized data formats like ORC (Optimized Row Columnar) and Parquet (another open source columnar format). The latter may not remain a strongly differentiating feature among RDBMSs and Hadoop-based systems for too long because RDBMSs may also incorporate special data formats.

Higher level language support. SQL was a distinguishing feature that was in favor for RDBMSs for writing complex analytical queries. However, Hive has incorporated a large number of SQL features in HiveQL, including grouping and aggregation as well as nested subqueries and multiple functions that are useful in data warehouses, as we discussed previously. Hive 0.13 is able to execute about 50 queries from the TPC-DS benchmark without any manual rewriting. New machine learning–oriented function libraries are emerging (e.g., the function library at madlib.net supports traditional RDBMSs like PostgreSql as well as the Pivotal distribution of Hadoop database (PHD)). Pivotal's HAWQ claims to be the latest and most powerful parallel SQL engine combining the advantages of SQL and Hadoop. Furthermore, the YARN plugin architecture that we discussed simplifies the process of extending the fabric with new components and new functions. Pig and Hive have extendibility with UDFs (user-defined functions). Several data services are now available on YARN, such as Revolution R and Apache Mahout for machine learning and Giraph for graph processing. Many traditional DBMSs now run on the YARN platform; for example, the Vortex analytic platform from Actian[21] and BigSQL 3.0 from IBM.[22]

Fault tolerance. Fault tolerance remains a decided advantage of MR-based systems. The panel of authors in Pavlo et al. (2009) also acknowledged that "MR does a superior job of minimizing the amount of work lost when a hardware failure occurs." As pointed out by these authors, this capability comes at the cost of materializing intermediate files between Map and Reduce phases. But as Hadoop begins to handle very complex data flows (such as in Apache Tez) and as the need for latencies decreases, users can trade off performance for fault tolerance. For example, in Apache Spark one can configure an intermediate Resilient Distributed Dataset (RDD)[23] to be either materialized on disk or in memory, or even to be recomputed from its input.

As we can see from this discussion, even though MR started with a goal of supporting batch-oriented workloads, it could not keep up with traditional parallel RDBMSs in terms of interactive query workloads, as exemplified by Pavlo et al. (2009). However, the two camps have moved much closer to each other in capabilities. Market forces, such as the need for venture capital for new startups, require an SQL engine for new applications that largely deal with very large semistructured datasets; and the research community's interest and involvement have brought about substantial improvements in Hadoop's capability to handle traditional analytical workloads. But there is still significant catching up to be done in all the areas pointed out in Pavlo et al. (2009): runtime, planning and optimization, and analytic feature-sets.

---

[21] See http://www.actian.com/about-us/blog/sql-hadoop-real-deal/ for a current description.

[22] See Presentation at http://www.slideshare.net/Hadoop_Summit/w-325p230-azubirigrayatv4 for a current description.

[23] See Zaharia et al. (2012).

25.6.2 Big Data in Cloud Computing

The cloud computing movement and the big data movement have been proceeding concurrently for more than a decade. It is not possible to address the details of cloud computing issues in the present context. However, we state some compelling reasons why big data technology is in some sense dependent on cloud technology not only for its further expansion, but for its continued existence.

- The cloud model affords a high degree of flexibility in terms of management of resources: "scaling out," which refers to adding more nodes or resources; "scaling up," which refers to adding more resources to a node in the system; or even downgrading are easily handled almost instantaneously.
- The resources are interchangeable; this fact, coupled with the design of distributed software, creates a good ecosystem where failure can be absorbed easily and where virtual computing instances can be left unperturbed. For the cost of a few hundred dollars, it is possible to perform data mining operations that involve complete scans of terabyte databases, and to crawl huge Web sites that contain millions of pages.
- It is not uncommon for big data projects to exhibit unpredictable or peak computing power and storage needs. These projects are faced with the challenge of providing for this peak demand on an as-needed and not necessarily continuous basis. At the same time, business stakeholders expect swift, inexpensive, and dependable products and project outcomes. To meet with these conflicting requirements, cloud services offer an ideal solution.
- A common situation in which cloud services and big data go hand-in-hand is as follows: Data is transferred to or collected in a cloud data storage system, like Amazon's S3, for the purpose of collecting log files or exporting text-formatted data. Alternatively, database adapters can be utilized to access data from databases in the cloud. Data processing frameworks like Pig, Hive, and MapReduce, which we described above in Section 25.4, are used to analyze raw data (which may have originated in the cloud).
- Big data projects and startup companies benefit a great deal from using a cloud storage service. They can trade capital expenditure for operational expenditure; this is an excellent trade because it requires no capital outlay or risk. Cloud storage provides reliable and scalable storage solutions of a quality otherwise unachievable.
- Cloud services and resources are globally distributed. They ensure high availability and durability unattainable by most but the largest organizations.

The Netflix Case for Marrying Cloud and Big Data.[24] Netflix is a large organization characterized by a very profitable business model and an extremely inexpensive and reliable service for consumers. Netflix provides video streaming services to millions of customers today thanks to a highly efficient information

[24]Based on http://techblog.netflix.com/2013/01/hadoop-platform-as-service-in-cloud.html

system and data warehouse. Netflix uses Amazon S3 rather than HDFS as the data processing and analysis platform for several reasons. Netflix presently uses Amazon's Elastic MapReduce (EMR) distribution of Hadoop. Netflix cites the main reason for its choice as the following: S3 is designed for 99.999999999% durability and 99.99% availability of objects over a given year, and S3 can sustain concurrent loss of data in two facilities. S3 provides bucket versioning, which allows Netflix to recover inadvertently deleted data. The elasticity of S3 has allowed Netflix a practically unlimited storage capacity; this capacity has enabled Netflix to grow its storage from a few hundred terabytes to petabytes without any difficulty or prior planning. Using S3 as the data warehouse enables Netflix to run multiple Hadoop clusters that are fault-tolerant and can sustain excess load. Netflix executives claim that they have no concerns about data redistribution or loss during expansion or shrinking of the warehouse. Although Netflix's production and query clusters are long-running clusters in the cloud, they can be essentially treated as completely transient. If a cluster goes down, Netflix can simply substitute with another identically sized cluster, possibly in a different geographic zone, in a few minutes and not sustain any data loss.

25.6.3 Data Locality Issues and Resource Optimization for Big Data Applications in a Cloud

The increasing interest in cloud computing combined with the demands of big data technology means that data centers must be increasingly cost-effective and consumer-driven. Also, many cloud infrastructures are not intrinsically designed to handle the scale of data required for present-day data analytics. Cloud service providers are faced with daunting challenges in terms of resource management and capacity planning to provide for big data technology applications.

The network load of many big data applications, including Hadoop/MapReduce, is of special concern in a data center because large amounts of data can be generated during job execution. For instance, in a MapReduce job, each reduce task needs to read the output of all map tasks, and a sudden explosion of network traffic can significantly deteriorate cloud performance. Also, when data is located in one infrastructure (say, in a storage cloud like Amazon S3) and processed in a compute cloud (such as Amazon EC2), job performance suffers significant delays due to data loading.

Research projects have proposed[25] a self-configurable, locality-based data and virtual machine management framework based on the storage-compute model. This framework enables MapReduce jobs to access most of their data either locally or from close-by nodes, including all input, output, and intermediate data generated during map and reduce phases of the jobs. Such frameworks categorize jobs using a data-size sensitive classifier into four classes based on a data size–based footprint. Then they provision virtual MapReduce clusters in a locality-aware manner, which enables efficient pairing and allocation of MapReduce virtual machines (VMs) to reduce the network distance between storage and compute nodes for both map and reduce processing.

[25]See Palanisamy et al. (2011).

Recently, caching techniques have been shown to improve the performance of MapReduce jobs for various workloads.[26] The PACMan framework provides support for in-memory caching, and the MixApart system provides support for disk-based caching when the data is stored in an enterprise storage server within the same site. Caching techniques allow flexibility in that data is stored in a separate storage infrastructure that allows prefetching and caching of the most essential data. Recent work[27] has addressed the big data caching problem in the context of privacy-conscious scenarios, wherein data stored in encrypted form in a public cloud must be processed in a separate, secure enterprise site.

In addition to the data locality problem, one of the most challenging goals for cloud providers is to optimally provision virtual clusters for jobs while minimizing the overall consumption cost of the cloud data center.

An important focus of cloud resource optimization is to optimize globally across all jobs in the cloud as opposed to per-job resource optimizations. A good example of a globally optimized cloud- managed system is the recent Google BigQuery system,[28] which allows Google to run SQL-like queries against very large datasets with potentially billions of rows using an Excel-like interface. In the BigQuery service, customers only submit the queries to be processed on the large datasets, and the cloud system intelligently manages the resources for the SQL-like queries. Similarly, the Cura resource optimization model[29] proposed for MapReduce in a cloud achieves global resource optimization by minimizing the overall resource utilization in the cloud as opposed to per-job or per-customer resource optimization.

25.6.4 YARN as a Data Service Platform

The separation of resource management from application management has taken Hadoop to another level as a platform. Hadoop v1 was all about MapReduce. In Hadoop v2, MapReduce is one of the many application frameworks that can run on the cluster. As we discussed in Section 25.5, this has opened the door for many services (with their own programming models) to be provided on YARN. There is no need to translate all data processing techniques and algorithms into a set of MapReduce jobs. MapReduce is presently being used only for batch-oriented processing such as the ETL (extract, transform, load) process in data warehouses (see Chapter 29). The emerging trend is to see Hadoop as a **data lake**, where a significant portion of enterprise data resides and where processing happens. Traditionally, HDFS has been where an enterprise's historical data resides because HDFS can handle the scale of such data. Most new sources of data, which in today's search and social networking applications come from Web and machine logs, clickstream data, message data (as in Twitter) and sensor data, also is being stored largely in HDFS.

---

[26]See the PACMAN framework by Ananthanarayanan et al. (2012) and the MixApart system by Mihailescu et al. (2013).

[27]See Palanisamy et al. (2014a).

[28]For the Google BigQuery system, see https://developers.google.com/bigquery/

[29]Palanisamy et al. (2014b).

The Hadoop v1 model was the **federation** model: although HDFS was the storage layer for the enterprise, processing was a mixture of MapReduce and other engines. One alternative was to extract data from HDFS store to engines running outside the cluster in their own silos; such data was moved to graph engines, machine learning analytical applications, and so forth. The same machines as those used for the Hadoop cluster were being used for entirely different applications, such as stream processing outside of Hadoop. This scenario was far from ideal since physical resources had to be divvied up in a static manner and it was difficult to migrate and upgrade to new versions when multiple frameworks ran on the same machines. With YARN, the above issues are addressed. Traditional services are taking advantage of the YARN ResourceManager and are providing their service on the same Hadoop cluster where the data resides.

Whereas support for SQL in Hadoop was promised by multiple vendors, the actual support has been less than completely desirable. Some vendors required the HDFS data to be moved out to another database to run SQL; some required wrappers to read the HDFS data before an SQL query ran on it. A new trend among RDBMSs and traditional database systems considers a YARN cluster as a viable platform. One example is Actian's analytics platform, which provides SQL in Hadoop[30] and which is claimed to be a complete and robust implementation of SQL using the Actian Vectorwise columnar database (which runs as a YARN application). IBM's Big SQL 3.0[31] is a project that makes an existing IBM shared-nothing DBMS run on a YARN cluster.

Apache Storm is a distributed scalable streaming engine that allows users to process real-time data feeds. It is widely used by Twitter. Storm on YARN (http://hortonworks.com/labs/storm/) and SAS on YARN (http://hortonworks.com/partner/sas/) are applications that treat Storm (a distributed stream processing application) and SAS (statistical analysis software) as applications on the YARN platform. As we discussed previously, Giraph and HBase Hoya are ongoing efforts that are rapidly adopting YARN. A wide range of application systems uses the Hadoop cluster for storage; examples include services like streaming, machine learning/statistics, graph processing, OLAP, and key-value stores. These services go well beyond MapReduce. The goal/promise of YARN is for these services to coexist on the same cluster and take advantage of the locality of data in HDFS while YARN orchestrates their use of cluster resources.

25.6.5 Challenges Faced by Big Data Technologies

In a recent article,[32] several database experts voiced their concerns about the impending challenges faced by big data technologies when such technologies

[30]Current documentation is available at http://www.actian.com/about-us/blog/sql-hadoop-real-deal/

[31]Current information is available at: http://www.slideshare.net/Hadoop_Summit/w-325p230-azubirigrayatv4

[32]See Jagadish et al. (2014).

are used primarily for analytics applications. These concerns include the following:

- **Heterogeneity of information:** Heterogeneity in terms of data types, data formats, data representation, and semantics is unavoidable when it comes to sources of data. One of the phases in the big data life cycle involves integration of this data. The cost of doing a clean job of integration to bring all data into a single structure is prohibitive for most applications, such as healthcare, energy, transportation, urban planning, and environmental modeling. Most machine learning algorithms expect data to be fed into them in a uniform structure. The data provenance (which refers to the information about the origin and ownership of data) is typically not maintained in most analytics applications. Proper interpretation of data analysis results requires large amounts of metadata.
- **Privacy and confidentiality:** Regulations and laws regarding protection of confidential information are not always available and hence not applied strictly during big data analysis. Enforcement of HIPAA regulations in the healthcare environment is one of few instances where privacy and confidentiality are strictly enforced. Location-based applications (such as on smart phones and other GPS-equipped devices), logs of user transactions, and clickstreams that capture user behavior all reveal confidential information. User movement and buying patterns can be tracked to reveal personal identity. Because it is now possible to harness and analyze billions of users' records via the technologies described in this chapter, there is widespread concern about personal information being compromised (e.g., data about individuals could be leaked from social data networks that are in some way linked to other data networks). Data about customers, cardholders, and employees is held by organizations and thus is subject to breaches of confidentiality. Jagadish et al. (2014) voiced a need for stricter control over digital rights management of data similar to the control exercised in the music industry.
- **Need for visualization and better human interfaces:** Huge volumes of data are crunched by big data systems, and the results of analyses must be interpreted and understood by humans. Human preferences must be accounted for and data must be presented in a properly digestible form. Humans are experts at detecting patterns and have great intuition about data they are familiar with. Machines cannot match humans in this regard. It should be possible to bring together multiple human experts to share and interpret results of analysis and thereby increase understanding of those results. Multiple modes of visual exploration must be possible to make the best use of data and to properly interpret results that are out of range and thus are classified as outlier values.
- **Inconsistent and incomplete information:** This has been a perennial problem in data collection and management. Future big data systems will allow multiple sources to be handled by multiple coexisting applications, so problems due to missing data, erroneous data, and uncertain data will be compounded. The large volume and built-in redundancy of data in fault-tolerant

systems may compensate to some extent for the missing values, conflicting values, hidden relationships, and the like. There is an inherent uncertainty about data collected from regular users using normal devices when such data comes in multiple forms (e.g., images, rates of speed, direction of travel). There is still a lot to be learned about how to use crowdsourcing data to generate effective decision making.

The aforementioned issues are not new to information systems. However, the large volume and wide variety of information inherent in big data systems compounds these issues.

25.6.6 Moving Forward

YARN makes it feasible for enterprises to run and manage many services on one cluster. But building data solutions on Hadoop is still a daunting challenge. A solution may involve assembling ETL (extract, transform, load) processing, machine learning, graph processing, and/or report creation. Although these different functional engines all run on the same cluster, their programming models and metadata are not unified. Analytics application developers must try to integrate all these services into a coherent solution.

On current hardware, each node contains a significant amount of main memory and flash memory storage. The cluster thus becomes a vast resource of main memory and flash storage. Significant innovation has demonstrated the performance gains of **in-memory data engines**; for example, SAP HANA is an in-memory, columnar scale-out RDBMS that is gaining a wide following.[33]

The Spark platform from Databricks (https://databricks.com/), which is an offshoot of the Berkeley Data Analytics Stack from AMPLabs at Berkeley,[34]addresses both of the advances mentioned above—namely, the ability to house diverse applications in one cluster and the ability to use vast amounts of main memory for faster response. Matei Zaharia developed the Resilient Distributed Datasets (RDD) concept[35] as a part of his Ph.D. work at the University of California–Berkeley that gave rise to the Spark system. The concept is generic enough to be used across all Spark's engines: Spark core (data flow), Spark-SQL, GraphX, (graph processing), MLLib (machine learning), and Spark-Streaming (stream processing). For example, it is possible to write a script in Spark that expresses a data flow that reads data from HDFS, reshapes the data using a Spark-SQL query, passes that information to an MLLib function for machine learning–type analysis, and then stores the result back in HDFS.[36]

---

[33]See http://www.saphana.com/welcome for a variety of documentation on SAP's HANA system.

[34]See https://amplab.cs.berkeley.edu/software/ for projects at Amplab from the University of California–Berkeley.

[35]The RDD concept was first proposed in Zaharia et al. (2012).

[36]See an example of the use of Spark at https://databricks.com/blog/2014/03/26/spark-sql-manipulating-structured-data-using-spark-2.html

RDDs are built on the capabilities of Scala language collections[37] that are able to re-create themselves from their input. RDDs can be configured based on how their data is distributed and how their data is represented: it can be always re-created from input, and it can be cached on disk or in memory. In-memory representations vary from serialized Java objects to highly optimized columnar formats that have all the advantages of columnar databases (e.g., speed, footprint, operating in serialized form).

The capabilities of a unified programming model and in-memory datasets will likely be incorporated into the Hadoop ecosystem. Spark is already available as a service in YARN (http://spark.apache.org/docs/1.0.0/running-on-yarn.html). Detailed discussion of Spark and related technologies in the Berkeley Data Analysis Stack is beyond our scope here. Agneeswaran (2014) discusses the potential of Spark and related products; interested readers should consult that source.

25.7 Summary

In this chapter, we discussed big data technologies. Reports from IBM, Mckinsey, and Tearadata scientist Bill Franks all predict a vibrant future for this technology, which will be at the center of future data analytics and machine learning applications and which is predicted to save businesses billions of dollars in the coming years.

We began our discussion by focusing on developments at Google with the Google file system and MapReduce (MR), a programming paradigm for distributed processing that is scalable to huge quantities of data reaching into the petabytes. After giving a historical development of the technology and mentioning the Hadoop ecosystem, which spans a large number of currently active Apache projects, we discussed the Hadoop distributed file system (HDFS) by outlining its architecture and its handling of file operations; we also touched on the scalability studies done on HDFS. We then gave details of the MapReduce runtime environment. We provided examples of how the MapReduce paradigm can be applied to a variety of contexts; we gave a detailed example of its application to optimizing various relational join algorithms. We then presented briefly the developments of Pig and Hive, the systems that provide an SQL-like interface with Pig Latin and HiveQL on top of the low-level MapReduce programming. We also mentioned the advantages of the joint Hadoop/MapReduce technology.

Hadoop/MapReduce is undergoing further development and is being repositioned as version 2, known as MRv2 or YARN; version 2 separates resource management from task/job management. We discussed the rationale behind YARN, its architecture, and other ongoing frameworks based on YARN, including Apache Tez, a workflow modeling environment; Apache Giraph, a large-scale graph processing system based on Pregel of Google; and Hoya, a Hortonworks rendering of HBase elastic clusters on YARN.

[37] See http://docs.scala-lang.org/overviews/core/architecture-of-scala-collections.html for more information on Scala Collections.

Finally, we presented a general discussion of some issues related to MapReduce/Hadoop technology. We briefly commented on the study done for this architecture vis-à-vis parallel DBMSs. There are circumstances where one is superior over the other, and claims about the superiority of parallel DBMSs for batch jobs are becoming less relevant due to architectural advancements in the form of YARN-related developments. We discussed the relationship between big data and cloud technologies and the work being done to address data locality issues in cloud storage for big data analytics. We stated that YARN is being considered as a generic data services platform, and we listed the challenges for this technology as outlined in a paper authored by a group of database experts. We concluded with a summary of ongoing projects in the field of big data.

Review Questions

25.1. What is data analytics and what is its role in science and industry?

25.2. How will the big data movement support data analytics?

25.3. What are the important points made in the McKinsey Global Institute report of 2012?

25.4. How do you define big data?

25.5. What are the various types of analytics mentioned in the IBM (2014) book?

25.6. What are the four major characteristics of big data? Provide examples drawn from current practice of each characteristic.

25.7. What is meant by *veracity of data*?

25.8. Give the chronological history of the development of MapReduce/Hadoop technology.

25.9. Describe the execution workflow of the MapReduce programming environment.

25.10. Give some examples of MapReduce applications.

25.11. What are the core properties of a job in MapReduce?

25.12. What is the function of JobTracker?

25.13. What are the different releases of Hadoop?

25.14. Describe the architecture of Hadoop in your own words.

25.15. What is the function of the NameNode and secondary NameNode in HDFS?

25.16. What does the Journal in HDFS refer to? What data is kept in it?

25.17. Describe the heartbeat mechanism in HDFS.

25.18. How are copies of data (replicas) managed in HDFS?

25.19. Shvachko (2012) reported on HDFS performance. What did he find? Can you list some of his results?

25.20. What other projects are included in the open source Hadoop ecosystem?

25.21. Describe the workings of the JobTracker and TaskTracker in MapReduce.

25.22. Describe the overall flow of the job in MapReduce.

25.23. What are the different ways in which MapReduce provides fault tolerance?

25.24. What is the Shuffle procedure in MapReduce?

25.25. Describe how the various job schedulers for MapReduce work.

25.26. What are the different types of joins that can be optimized using MapReduce?

25.27. Describe the MapReduce join procedures for Sort-Merge join, Partition Join, *N*-way Map-side join, and Simple *N*-way join.

25.28. What is Apache Pig, and what is Pig Latin? Give an example of a query in Pig Latin.

25.29. What are the main features of Apache Hive? What is its high-level query language?

25.30. What is the SERDE architecture in Hive?

25.31. List some of the optimizations in Hive and its support of SQL.

25.32. Name some advantages of the MapReduce/Hadoop technology.

25.33. Give the rationale in moving from Hadoop v1 to Hadoop v2 (YARN).

25.34. Give an overview of the YARN architecture.

25.35. How does Resource Manager work in YARN?

25.36. What are Apache Tez, Apache Giraph, and Hoya?

25.37. Compare parallel relational DBMSs and the MapReduce/Hadoop systems.

25.38. In what way are big data and cloud technology complementary to one another?

25.39. What are the data locality issues related to big data applications in cloud storage?

25.40. What services can YARN offer beyond MapReduce?

25.41. What are some of the challenges faced by big data technologies today?

25.42. Discuss the concept of RDDs (resilient distributed datasets).

25.43. Find out more about ongoing projects such as Spark, Mesos, Shark, and BlinkDB as they relate to the Berkeley Data Analysis Stack.

Selected Bibliography

The technologies for big data discussed in this chapter have mostly sprung up in the last ten years or so. The origin of this wave is traced back to the seminal papers from Google, including the Google file system (Ghemawat, Gobioff, & Leung, 2003) and the MapReduce programming paradigm (Dean & Ghemawat, 2004). The Nutch system with follow-on work at Yahoo is a precursor of the Hadoop technology and continues as an Apache open source project (nutch.apache.org). The BigTable system from Google (Fay Chang et al., 2006) describes a distributed scalable storage system for managing structured data in the petabytes range over thousands of commodity servers.

It is not possible to name a specific single publication as "the" Hadoop paper. Many studies related to MapReduce and Hadoop have been published in the past decade. We will list only a few landmark developments here. Schvachko (2012) outlines the limitations of the HDFS file system. Afrati and Ullman (2010) is a good example of using MapReduce programming in various contexts and applications; they demonstrate how to optimize relational join operations in MapReduce. Olston et al. (2008) describe the Pig system and introduce Pig Latin as a high-level programming language. Thusoo et al. (2010) describe Hive as a petabyte- scale data warehouse on top of Hadoop. A system for large-scale graph processing called Pregel at Google is described in Malewicz et al. (2010). It uses the bulk synchronous parallel (BSP) model of parallel computation originally proposed by Valiant (1990). In Pavlo et al. (2009), a number of database technology experts compared two parallel RDBMSs with Hadoop/MapReduce and showed how the parallel DBMS can actually perform better under certain conditions. The results of this study must not be considered definitive because of the significant performance improvements achieved in Hadoop v2 (YARN). The approach of resilient distributed datasets (RDDs) for in-memory cluster computing is at the heart of the Berkeley's Spark system, developed by Zaharia et al. (2013). A recent paper by Jagadish et al. (2014) gives the collective opinion of a number of database experts about the challenges faced by the current big data technologies.

The definitive resource for Hadoop application developers is the book *Hadoop: The Definitive Guide*, by Tom White (2012), which is in its third edition. A book by YARN project founder Arun Murthy with Vavilapalli (2014) describes how YARN increases scalability and cluster utilization, enables new programming models and services, and extends applicability beyond batch applications and Java. Agneeswaran (2014) has written about going beyond Hadoop, and he describes the Berkeley Data Analysis Stack (BDAS) for real-time analytics and machine learning; the Stack includes Spark, Mesos, and Shark. He also describes Storm, a complex event-processing engine from Twitter widely used in industry today for real-time computing and analytics.

The Hadoop wiki is at Hadoop.apache.org. There are many open source, big data projects under Apache, such as Hive, Pig, Oozie, Sqoop, Storm, and HBase. Up-to-date information about these projects can be found in the documentation at the projects' Apache Web sites and wikis. The companies Cloudera, MapR, and Hor-

tonworks include on their Web sites documentation about their own distributions of MapReduce/Hadoop-related technologies. The Berkeley Amplab (https://amplab.cs.berkeley.edu/) provides documentation about the Berkeley Data Analysis Stack (BDAS), including ongoing projects such as GraphX, MLbase, and BlinkDB.

There are some good references that outline the promise of big data technology and large scale data management. Bill Franks (2012) talks about how to leverage big data technologies for advanced analytics and provides insights that will help practitioners make better decisions. Schmarzo (2013) discusses how the big data analytics can empower businesses. Dietrich et al. (2014) describe how IBM has applied the power of big data analytics across the enterprise in applications worldwide. A book published by McKinsey Global Institute (2012) gives a strategic angle on big data technologies by focusing on productivity, competitiveness, and growth.

There has been a parallel development in the cloud technologies that we have not been able to discuss in detail in this chapter. We refer the reader to recent books on cloud computing. Erl et al. (2013) discusses models, architectures, and business practices and desccribes how this technology has matured in practice. Kavis (2014) presents the various service models, including software as a service (SaaS), platform as a service (PaaS), and infrastructure as a service (IaaS). Bahga and Madisetti (2013) offer a practical, hands-on introduction to cloud computing. They describe how to develop cloud applications on various cloud platforms, such as Amazon Web Service (AWS), Google Cloud, and Microsoft's Windows Azure.