CHAPTER

Cloud applications 11

Over the years the users of large-scale computing systems in data centers discovered how difficult it was to develop efficient data-intensive and computationally intensive applications. It was equally difficult to locate systems best-suited to run an application, to determine when an application was able to run on these systems, and to estimate the time when the results could be expected. Porting an application from one system to another frequently constituted a challenging endeavor. Often, an application optimized for one system performed poorly on other systems.

There were also formidable challenges for the providers of service because system resources could not be effectively managed, and it was not feasible to provide QoS guarantees. Accommodating a dynamic load and supporting security and rapid recovery after a system-wide failure were daunting tasks due to the scale of the system. Any economic advantage offered by resource concentration was offset by the relatively low utilization of costly resources.

Cloud computing has changed the views and perceptions of users and providers of service on how to compute more efficiently and at a lower cost. Most of the challenges faced by application developers and service providers have either disappeared, or are significantly diminished. Cloud application developers are now able to work in a familiar environment and enjoy the advantages of a *just-in-time infrastructure*; they are free to design an application without being concerned where the application will run.

Cloud elasticity allows applications to seamlessly absorb additional workload. Cloud users also benefit from the speedup due to parallelization. When the workload can be partitioned in n segments, the application can spawn n instances of itself and run them concurrently, resulting in dramatic speedups. This is particularly useful for computer-aided design and for modeling complex systems when multiple design alternatives and multiple models of a physical system can be evaluated at the same time.

Cloud computing is focused on enterprise computing which clearly differentiates it from the grid computing effort largely focused on scientific and engineering applications. An important advantage of cloud computing over grid computing is that the resources offered by a cloud service provider are in one administrative domain.

Cloud computing is beneficial for the providers of computing cycles because it typically leads to more efficient resource utilization. It soon became obvious that a significant percentage of the typical workloads are dominated by frameworks such as MapReduce and that multiple frameworks must share the large computer clusters populating the cloud infrastructure.

The future success of cloud computing rests on the ability of companies promoting utility computing to convince an increasingly larger segment of the user population of the advantages of network-centric computing and network-centric content. This translates into the ability to provide satisfactory solutions to critical aspects of security, scalability, reliability, quality of service, and the requirements agreed upon in SLAs.

Cloud Computing. https://doi.org/10.1016/B978-0-32-385277-7.00018-X

Sections 11.1, 11.2, and 11.3 cover cloud application developments and provide insights into workflow management. Coordination based on a state machine model is presented in Section 11.4, followed by the in-depth discussion of the MapReduce programming model and one of its applications in Sections 11.5 and 11.6. Hadoop, Yarn, and Tez are covered in Section 11.7, while systems such a Pig, Hive, and Impala are discussed in Section 11.8.

An overview of current cloud applications and new application opportunities is presented in Section 11.9, followed by a discussion of cloud applications in science and engineering and in biology research in Sections 11.10 and 11.11, respectively. Social computing and software fault isolations are the subjects of Sections 11.12 and 11.13.

11.1 Cloud application development and architectural styles

Web services, database services, and transaction-based services are ideal applications for cloud computing. The cost-performance profiles of such applications benefit from an elastic environment where resources are available when needed and where a cloud user pays only for the resources consumed by her application.

Not all applications are suitable for cloud computing. Applications where the workload cannot be arbitrarily partitioned, or require intensive communication among concurrent instances are unlikely to perform well on a cloud. An application with a complex workflow and multiple dependencies, as is often the case in high-performance computing, could require longer execution times and higher costs on a cloud. Benchmarks for high-performance computing, discussed in Section 11.10, show that communication and memory-intensive applications may not exhibit the performance levels shown when running on supercomputers with low-latency and high-bandwidth interconnects.

Cloud application development challenges. The development of efficient cloud applications faces challenges posed by the inherent imbalance between computing, I/O, and the communication bandwidth of processors. These challenges are greatly amplified due to the scale of the cloud infrastructure, its distributed nature, and by the very nature of data-intensive applications. Though cloud computing infrastructures attempt to automatically distribute and balance the workload, application developers are still left with the responsibility to identify optimal storage for the data, exploit spatial and temporal data and code locality, and minimize communication among running treads and instances.

A main attraction of cloud computing is the ability to use as many servers as necessary to optimally respond to the cost and the timing constraints of an application. This is only possible if the workload can be partitioned in segments of arbitrary size and can be processed in parallel by the servers available on the cloud. The *arbitrarily divisible load sharing model* describes workloads that can be partitioned into an arbitrarily large number of units and can be processed concurrently by multiple cloud instances. Applications most suitable for cloud computing enjoy this model.

The shared infrastructure, a defining quality of cloud computing, has side effects. Performance isolation, discussed in Section 5.2, is nearly impossible under real conditions, especially when a system is heavily loaded. The performance of virtual machines fluctuates based on the workload and the environment. Security isolation is challenging on multitenant systems.

Reliability of the cloud infrastructure is also a major concern. The frequent failures of servers built with off-the-shelf components is a consequence of the sheer number of servers and communication

systems. Choosing an optimal instance from those offered by the cloud infrastructure is another critical factor to be considered. Instances differ in terms of performance isolation, reliability, and security. Cost considerations also play a role in the choice of the instance type.

Many applications consist of multiple stages. In turn, each stage may involve multiple instances running in parallel on cloud servers and communicating among them. Thus, efficiency, consistency, and communication scalability are major concerns for an application developer. A cloud infrastructure exhibits latency and bandwidth fluctuations affecting the performance of all applications and, in particular, of data-intensive ones.

Data storage plays a critical role in the performance of any data-intensive application. The organization and the location of data storage, as well as the storage bandwidth, must be carefully analyzed to ensure optimal application performance. Clouds support many storage options to set up a file system similar to the Hadoop file system discussed in Section 11.7. Among them are off-instance cloud storage, e.g. S3, and mountable off-instance block storage, e.g., EBS, as well as storage persistent for the lifetime of the instance.

Many data-intensive applications use metadata associated with individual data records. For example, the metadata for an MPEG audio file may include the title of the song, the name of the singer, recording information, etc. Metadata should be stored for easy access; the storage should be scalable, and reliable.

Another important consideration for the application developer is logging. Performance considerations limit the amount of data logging, while the ability to identify the source of unexpected results and errors is helped by frequent logging. Logging is typically done using instance storage preserved only for the lifetime of the instance; thus measures to preserve the logs for a post-mortem analysis must be taken. Another challenge waiting resolution is related to software licensing, discussed in Section 2.11.

Cloud application architectural styles. Cloud computing is based on the client–server paradigm discussed in Section 3.12. The vast majority of cloud applications take advantage of request–response communication between clients and stateless servers. A *stateless server* does not require a client to first establish a connection to the server; instead, it views a client request as an independent transaction and responds to it.

The advantages of stateless servers are obvious. Recovering from a server failure requires a considerable overhead for a server that maintains the state of all its connections, but, in case of a stateless server, a client is not affected while a server goes down and then comes back up between two consecutive requests. A stateless system is simpler, more robust, and scalable; a client does not have to be concerned with the state of the server; if the client receives a response to a request, this means that the server is up and running; if not, it should resend the request later. A connection-based service must reserve space to maintain the state of each connection with a client; therefore, such a system is not scalable, and the number of clients a server could interact with at any given time is limited by the storage space available to the server.

The Hypertext Transfer Protocol used by a browser to communicate with a web server is a request–response application protocol. HTTP uses TCP, a connection-oriented and reliable transport protocol. While TCP ensures reliable delivery of large objects, it exposes web servers to denial of service attacks. In such attacks, malicious clients fake attempts to establish a TCP connection and force the server to allocate space for the connection. A basic web server is stateless; it responds to an HTTP request without maintaining a history of past interactions with the client. The client, a browser, is also stateless since it sends requests and waits for responses.

A critical aspect of the development of networked applications is how processes and threads running on systems with different architectures, possibly compiled from different programming languages, can *communicate structured information with one another*. First, the internal representation of the two structures at the two sites may be different; one system may use *Big-Endian* and the other *Little-Endian* representation. The character representations also may be different. A communication channel transmits a sequence of bits/bytes and the data structure must be serialized at the sending site and reconstructed at the receiving site.

Several considerations including neutrality, extensibility, and independence, must be analyzed before choosing the architectural style of an application. *Neutrality* refers to application-level protocol ability to use different transport protocols such as TCP and UDP and to run on top of a different protocol stack. For example, SOAP can use not only TCP but also UDP, SMTP, and Java Message Service as transport vehicles. *Extensibility* refers to the ability to incorporate additional functions such as security. *Independence* refers to the ability to accommodate different programming styles.

Very often application clients and the servers running on the cloud communicate using RPCs, discussed in Section 3.12, yet other communication styles are possible. RPC-based applications use *stubs* to convert the parameters involved in an RPC call. A stub performs two functions, marshaling the data structures and serialization.

A more general concept is that of an Object Request Broker (ORB), middleware facilitating communication of networked applications. The sending site ORB transforms data structures used internally and transmits a byte sequence over the network. The receiving site ORB maps the byte sequence to data structures used internally by the receiving process.

Common Object Request Broker Architecture (CORBA) was developed in early 1990s to enable networked applications developed using different programming languages and running on systems with different architecture and system software to work with one another. At the heart of the system is the Interface Definition Language (IDL) used to specify the interface of an object. An IDL representation is then mapped to the set of programming languages including: C, C++, Java, Smalltalk, Ruby, Lisp, and Python. Networked applications pass CORBA by reference and pass data by value.

Simple Object Access Protocol (SOAP) was developed in 1998 for web applications. SOAP message format is based on the Extensible Markup Language (XML). SOAP uses TCP and more recently, UDP transport protocols, but it can also be stacked above other application layer protocols such as HTTP, SMTP, or JMS. The processing model of SOAP is based on a network consisting of senders, receivers, intermediaries, message originators, ultimate receivers, and message paths. SOAP is an underlying layer of Web Services.

Web Services Description Language (WSDL) was introduced in 2001 as an XML-based grammar to describe communication between end points of a networked application. The abstract definitions of the elements involved include: *services*, collection of endpoints of communication; *types*, containers for data type definitions; *operations*, description of actions supported by a service; *port types*, operations supported by endpoints; *bindings*, protocols and data format supported by a particular port type; and *port*, an endpoint as a combination of a binding and a network address. These abstractions are mapped to concrete message formats and network protocols to define endpoints and services.

Representational State Transfer (REST) is a software architecture for distributed hypermedia systems supporting client communication with stateless servers. It is platform- and language-independent, supports data caching, and can be used in the presence of firewalls. REST almost always uses HTTP

for all four CRUD (*Create/Read/Update/Delete*) operations; it uses *GET*, *PUT*, and *DELETE* to read, write, and delete the data, respectively.

REST is an easier to use alternative to RPC, CORBA, or web services such as SOAP or WSDL. For example, to retrieve the address of an individual from a database, a REST system sends an URL specifying the network address of the database, the name of the individual, and the specific attribute in the record that the client application wants to retrieve, in this case the address. The corresponding SOAP version of such a request consists of ten lines or more of XML. The REST server responds with the address of the individual. This justifies the statement that REST is a lightweight protocol. As far as usability is concerned, REST is easier to build from scratch and debug, but SOAP is supported by tools that use self-documentation, e.g., WSDL to generate the code to connect.

11.2 Coordination of multiple activities

Many applications require the completion of multiple interdependent tasks [529]. The description of a complex activity involving such an ensemble of tasks is known as a *workflow*. In this section, we discuss workflow models, the lifecycle of a workflow, and the desirable properties of a workflow description. Workflow patterns, reachability of the goal state of a workflow, dynamic workflows, and a parallel between traditional transaction systems and cloud workflows are covered in Section 11.3.

Basic concepts. Workflow models are abstractions revealing the most important properties of the entities participating in a workflow management system. *Task* is the central concept in workflow modeling. A task is a unit of work to be performed on the cloud, and it is characterized by several attributes, such as: (i) name—a string of characters uniquely identifying the task; (ii) description—a natural language description of the task; (iii) actions—an action is a modification of the environment caused by the execution of the task; (iv) preconditions—Boolean expressions that must be true before the action(s) of the task can take place; (v) postconditions—Boolean expressions that must be true after the action(s) of the task do take place; (vi) attributes—provide indications of the type and quantity of resources necessary for the execution of the task, the actors in charge of the tasks, the security requirements, whether the task is reversible or not, and other task characteristics; and (vii) exceptions—provide information on how to handle abnormal events. The exceptions supported by a task consist of a list of `<event, action>` pairs. The exceptions included in the task exception list are called *anticipated exceptions*, as opposed to unanticipated exceptions. Events not included in the exception list trigger replanning. *Replanning* means restructuring of a process, redefinition of the relationship among various tasks.

A *composite task* is a structure describing a subset of tasks and the order of their execution. A *primitive task* is one that cannot be decomposed into simpler tasks. A composite task inherits some properties from workflows; it consists of tasks, has one start symbol, and possibly several end symbols. At the same time, a composite task inherits some properties from tasks; it has a name, preconditions, and postconditions.

A *routing task* is a special-purpose task connecting two tasks in a workflow description. The task that has just completed execution is called the *predecessor* task, the one to be initiated next is called the *successor task*. A routing task could trigger the sequential, concurrent, or iterative execution. Two types of routing tasks exist:

- A *fork routing task* triggers execution of several successor tasks. Several semantics for this construct are possible: (i) all successor tasks are enabled; (ii) each successor task is associated with a condition, the conditions for all tasks are evaluated and only the tasks with a `true` condition are enabled; (iii) each successor task is associated with a condition, the conditions for all tasks are evaluated, but the conditions are mutually exclusive, and only one condition may be `true`, thus only one task is enabled; and (iv) nondeterministic—k out of $n > k$ successors are selected at random to be enabled.
- A *join routing task* waits for completion of its predecessor tasks. There are several semantics for the join routing task: (i) the successor is enabled after all predecessors end; (ii) the successor is enabled after k out of $n > k$ predecessors end; and (iii) iterative—the tasks between the fork and the join are executed repeatedly.

Process descriptions and cases. A *process description*, also called a workflow schema, is a structure describing the *tasks* or *activities* to be executed and the order of their execution; a process description contains one start symbol and one end symbol. A process description can be provided in a Workflow Definition Language (WFDL), supporting constructs for choice, concurrent execution, the classical *fork, join* constructs, and iterative execution. Workflow description resembles a *flowchart*, a concept we are familiar with from programming.

The phases in the life-cycle of a workflow are creation, definition, verification, and enactment. There is a striking similarity between the life-cycle of a workflow and that of a traditional computer program, namely, creation, compilation, and execution; see Fig. 11.1. The workflow specification by means of a workflow description language is analogous to writing a program. Planning is equivalent to automatic program generation. Workflow verification corresponds to syntactic verification of a program, and workflow enactment mirrors the execution of a compiled program.

A *case* is an instance of a process description. The start and stop symbols in the workflow description enable the creation and the termination of a case. An *enactment model* describes the steps taken to process a case. When all tasks required by a workflow are executed by a computer, the enactment can be performed by a program called an *enactment engine*.

The *state of a case* at time t is defined in terms of tasks already completed at that time. Events cause transitions between states. Identifying the states of a case consisting of concurrent activities is considerably more difficult than identifying the states of a strictly sequential process. Indeed, when several activities could proceed concurrently, the state has to reflect the progress made on each independent activity.

An alternative description of a workflow can be provided by a transition system describing the possible paths from the current state to a goal state. Sometimes, instead of providing a process description, we may specify only the goal state and expect the system to generate a workflow description that could lead to that state through a set of actions. In this case, the new workflow description is generated automatically, knowing a set of tasks and the preconditions and postconditions for each one of them. In AI, this activity is known as *planning*.

The state space of a process includes one initial state and one goal state. A transition system identifies all possible paths from the initial to the goal state. A case corresponds to a particular path in the transition system. The state of a case tracks the progress made during the enactment of that case.

Safety and liveness are the most desirable properties of a process description. Informally, *safety* means that nothing "bad" ever happens, while *liveness* means that something "good" will eventually take place, should a case based on the process be enacted. Not all processes are safe and live. For example, the process description in Fig. 11.2(a) violates the liveness requirement. As long as task C is

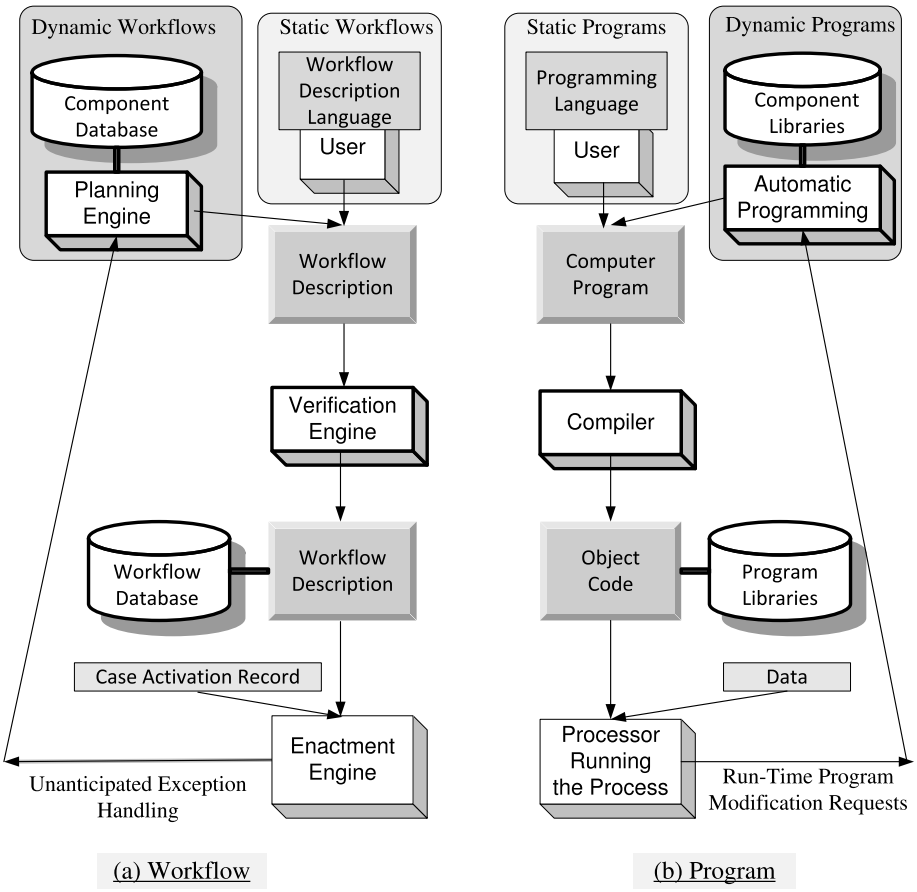

FIGURE 11.1

Workflows and programs. (a) The life-cycle of a workflow. (b) The life-cycle of a computer program. The workflow definition is analogous to writing a program. Planning is analogous to automatic program generation. Verification corresponds to syntactic verification of a program. Workflow enactment mirrors the execution of a program. A static workflow corresponds to a static program and a dynamic workflow to a dynamic program.

chosen after completion of B, the process will terminate. However, if D is chosen, then F will never be instantiated because it requires the completion of both C and E. The process will never terminate because G requires completion of both D and F.

A process description language should be unambiguous and should allow a verification of the process description before the enactment of a case. It is entirely possible that a process description may be enacted correctly in some cases but could fail for others. Such enactment failures may be very costly and should be prevented by a thorough verification at the process definition time. To avoid enactment

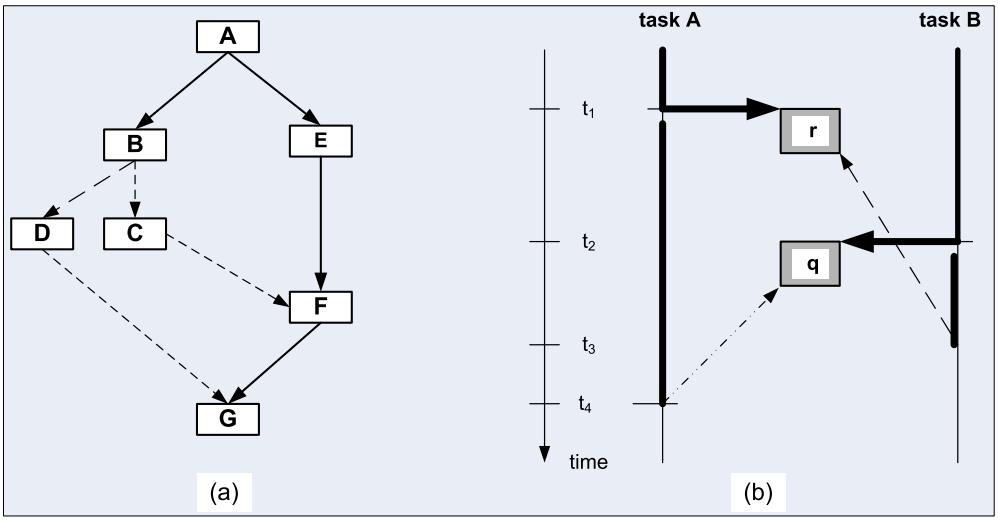

FIGURE 11.2

(a) A process description that violates the liveness requirement; if task C is chosen after completion of B, the process will terminate after executing task G; if D is chosen, then F will never be instantiated because it requires the completion of both C and E. The process will never terminate because G requires completion of both D and F. (b) Tasks A and B need exclusive access to two resources r and q, and a deadlock may occur if the following sequence of events occur: at time t_1, task A acquires r, at time t_2, task B acquires q and continues to run; then, at time t_3, task B attempts to acquire r, and it blocks because r is under the control of A; task A continues to run at time t_4, attempts to acquire q, and blocks because q is under the control of B.

errors, we need to verify process description and check for desirable properties such as safety and liveness. Some process description methods are more suitable for verification than others.

A note of caution: Although the original description of a process could be live, the actual enactment of a case may be affected by deadlocks due to resource allocation. To illustrate this situation, consider two tasks, A and B, running concurrently; each of them needs exclusive access to resources r and q for a period of time. Two scenarios are possible:

(1) either A or B acquires both resources and then releases them, and allows the other task to do the same;

(2) we face the undesirable situation in Fig. 11.2(b) when at time t_1 task A acquires r and continues its execution; then, at time t_2, task B acquires q and continues to run. Then, at time t_3, task B attempts to acquire r, and it blocks because r is under the control of A. Task A continues to run, and at time t_4 attempts to acquire q and it blocks because q is under the control of B.

The deadlock illustrated in Fig. 11.2(b) can be avoided by requesting each task to acquire all resources at the same time; the price to pay is underutilization of resources; indeed, the idle time of each resource increases under this scheme.

11.3 Workflow patterns

The term *workflow pattern* refers to the temporal relationships among the tasks of a process. The workflow description languages and the mechanisms to control the enactment of a case must have provisions to support these temporal relationships. Workflow patterns are analyzed in [1], [325], and [531]. These patterns are classified in several categories: basic, advanced branching and synchronization, structural, state-based, cancelation, and patterns involving multiple instances. The basic workflow patterns in Fig. 11.3 are:

Sequence—occurs when several tasks have to be scheduled one after the completion of the other, Fig. 11.3(a).
AND split—requires several tasks to be executed concurrently. Both tasks B and C are activated when task A terminates; see Fig. 11.3(b). In case of an *explicit AND split,* the activity graph has a routing node, and all activities connected to the routing node are activated as soon as the flow of control reaches the routing node. In the case of an *implicit AND split,* activities are connected directly and conditions can be associated with branches linking an activity with the next ones. The tasks are activated only when the conditions associated with a branch are true.
Synchronization—an activity can start only after several concurrent activities finish execution; task C can start only after both tasks *A* and *B* terminate, Fig. 11.3(c).
XOR split—requires a decision; after the completion of task A, either B or C can be activated, Fig. 11.3(d).
XOR join—several alternatives are merged into one; task *C* is enabled when either *A* or *B* terminates; see Fig. 11.3(e).
OR split—choose multiple alternatives out of a set; after completion of task *A*, one could activate either *B* or *C*, or both; see Fig. 11.3(f).
Multiple merge—allows multiple activations of a task and does not require synchronization after the execution of concurrent tasks; once A terminates, tasks *B* and *C* execute concurrently; see Fig. 11.3(g). When the first of them, say *B*, terminates, then task *D* is activated; then, when *C* terminates, *D* is activated again.
Discriminator—waits for a number of incoming branches to complete before activating the subsequent activity; see Fig. 11.3(h); then, it waits for the remaining branches to finish without taking any action until all of them have terminated; next, it resets itself.
N out of M join—provides a barrier synchronization; assuming $M > N$ tasks run concurrently, N of them have to reach the barrier before the next task is enabled; in our example, any two out of the three tasks A, B, and C have to finish before E is enabled; see Fig. 11.3(i).
Deferred choice—similar to XOR split but the choice is not made explicitly and the run-time environment decides what branch to take; see Fig. 11.3(j).

Next, we discuss the reachability of the goal state, and we consider the following elements:

- A system Σ, an initial state of the system, $\sigma_{initial}$, and a goal state, σ_{goal}.
- A process group, $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$; each process p_i in the process group is characterized by a set of preconditions, $pre(p_i)$, postconditions, $post(p_i)$, and attributes, $atr(p_i)$.
- A workflow described by a directed activity graph $\mathcal{A}$ or by a procedure Π capable to construct $\mathcal{A}$ given the tuple $< \mathcal{P}, \sigma_{initial}, \sigma_{goal} >$. The nodes of $\mathcal{A}$ are processes in $\mathcal{P}$, and the edges define precedence relationships among processes. $P_i \rightarrow P_j$ implies that $pre(p_j) \subset post(p_i)$.

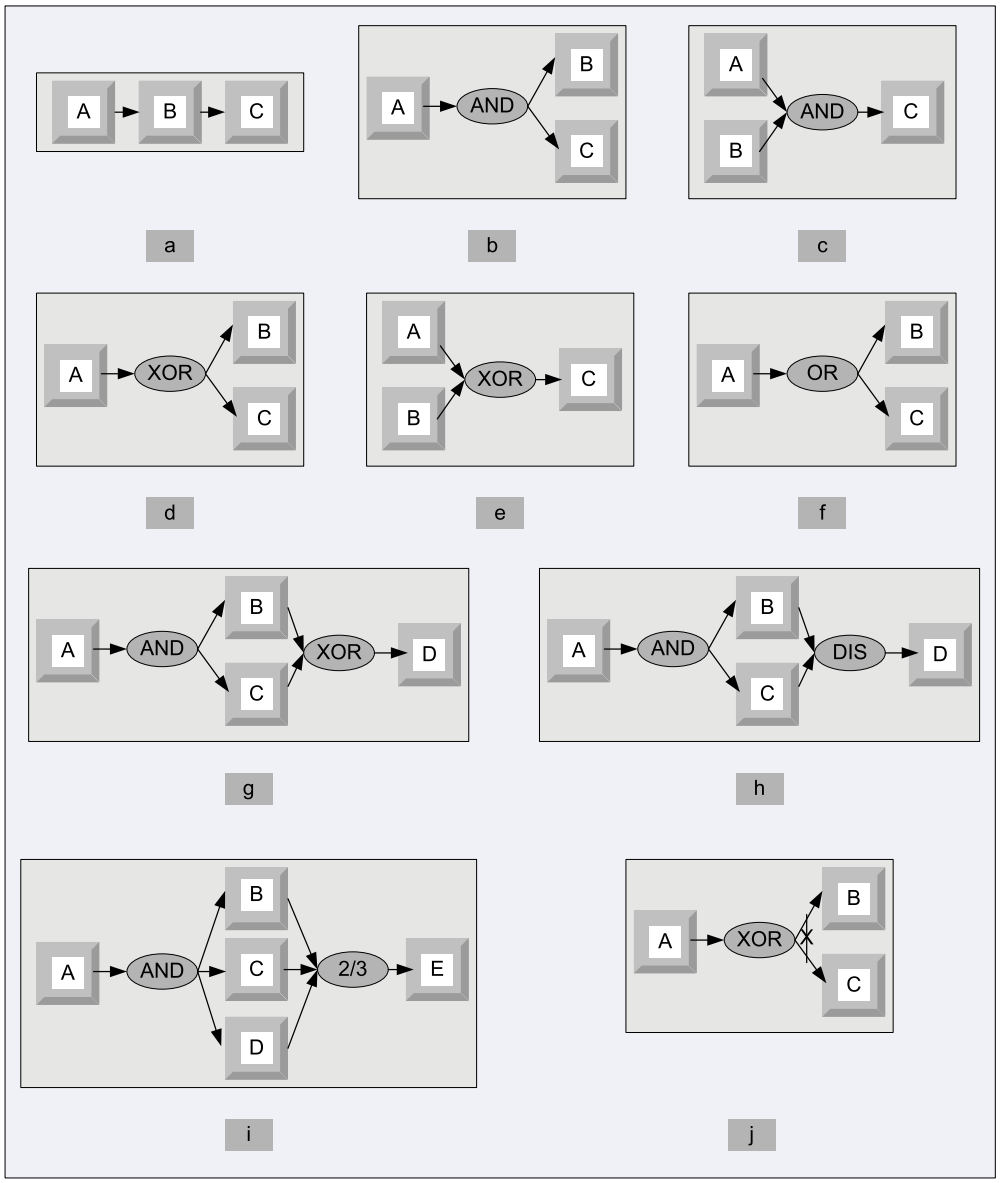

FIGURE 11.3

Basic workflow patterns. (a) Sequence; (b) AND split; (c) Synchronization; (d) XOR split; (e) XOR merge; (f) OR split; (g) Multiple Merge; (h) Discriminator; (i) N out of M join; (j) Deferred Choice.

- A set of constraints, $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$.

The coordination problem for system Σ in state $\sigma_{initial}$ is to reach state σ_{goal}, as a result of postconditions of some process $P_{final} \in \mathcal{P}$ subject to constraints $C_i \in \mathcal{C}$. Here, $\sigma_{initial}$ enables the preconditions of some process $P_{initial} \in \mathcal{P}$. Informally, this means that a chain of processes exists such that the postconditions of one process are preconditions of the next process in the chain.

The *preconditions* of a process are either the conditions and/or the events that trigger the execution of the process or the data the process expects as input. The *postconditions* are the results produced by that process. The attributes of a process describe special requirements or properties of the process.

Some workflows are static, and the activity graph does not change during the enactment of a case. *Dynamic workflows* are those that allow the activity graph to be modified during the enactment of a case. Some of the more difficult questions encountered in dynamic workflow management refer to: (i) how to integrate workflow and resource management and guarantee optimality or near optimality of cost functions for individual cases; (ii) how to guarantee consistency after a change in a workflow; and (iii) how to create a dynamic workflow. Static workflows can be described in WFDL (the workflow definition language), but dynamic workflows need a more flexible approach.

We distinguish two basic models for the mechanics of workflow enactment: (i) *strong coordination models* in which the process group $\mathcal{P}$ executes under the supervision of a *coordinator* process and a coordinator process acts as an enactment engine and ensures a seamless transition from one process to another in the activity graph; and (ii) *Weak coordination models* in which there is no supervisory process.

In the first case, we may deploy a *hierarchical coordination scheme* with several levels of coordinators. A supervisor at level i in a hierarchical scheme with $i + 1$ levels coordinates a subset of processes in the process group. A supervisor at level $i - 1$ coordinates a number of supervisors at level i, and the root provides global coordination. Such a hierarchical coordination scheme may be used to reduce the communication overhead; a coordinator and the processes it supervises may be colocated.

An important feature of this coordination model is its ability of supporting dynamic workflows. The coordinator or the global coordinator may respond to a request to modify the workflow by first stopping all the threads of control in a consistent state, then investigating the feasibility of the requested changes, and finally implementing feasible changes.

Weak coordination models are based on peer-to-peer communication between processes in the process group by means of a societal service, such as a *tuple space*. Once a process $p_i \in \mathcal{P}$ finishes, it deposits a token including possibly a subset of its postconditions, $post(p_i)$, in a tuple space. The consumer process p_j is expected to visit at some point in time the tuple space, examine the tokens left by its ancestors in the activity graph, and, if its preconditions $pre(p_j)$ are satisfied, commence the execution. This approach requires individual processes to either have a copy of the activity graph or some timetable to visit the tuple space. An alternative approach is using an *active space*, a tuple space augmented with the ability to generate an event awakening the consumer of a token.

There are similarities and some differences between workflows of traditional transaction-oriented systems and cloud workflows; the similarities are mostly at the modeling level, whereas the differences affect the mechanisms used to implement workflow management systems. Some of the more subtle differences between them are:

- The emphasis in a transactional model is placed on the contractual aspect of a transaction; in a workflow the enactment of a case is sometimes based on a "best-effort" model where the agents involved will do their best to attain the goal state, but there is no guarantee of success.

- A critical aspect of the transactional model in database applications is maintaining a consistent state of the database; a cloud is an open system; thus its state is considerably more difficult to define.
- Database transactions are typically short-lived; the tasks of a cloud workflow could be long lasting.
- A database transaction consists of a set of well-defined actions that are unlikely to be altered during the execution of the transaction. However, the process description of a cloud workflow may change during the lifetime of a case.
- The individual tasks of a cloud workflow may not exhibit the traditional properties of database transactions. For example, consider durability, which means that, at any instance of time before reaching the goal state, a workflow may roll back to some previously encountered state and continue from there on an entirely different path. A task of a workflow could be either reversible or irreversible. Sometimes, paying a penalty for reversing an action is more profitable in the long run than continuing on a wrong path.
- Resource allocation is a critical aspect of the workflow enactment on a cloud without an immediate correspondent for database transactions.

The relatively simple coordination model discussed next is often used in cloud computing.

11.4 Coordination based on a state machine model—zookeeper

Cloud computing elasticity requires the ability to distribute computations and data across multiple systems. In a distributed computing environment coordination among these systems is a critical function. The coordination model depends on the specific task, e.g., coordination of data storage, orchestration of multiple activities, blocking an activity until an event occurs, reaching consensus on the next action, or recovery after an error.

The entities to be coordinated could be processes running on a set of cloud servers or even running on multiple clouds. Servers running critical task are often replicated; when one primary server fails, a backup automatically continues the execution. This is only possible if the backup is in a *hot standby* mode, in other words, if the primary server shares its state with the backup at all times.

For example, in the distributed data store model discussed in Section 2.6, the access to data is mitigated by a proxy. An architecture with multiple proxies is desirable, as a proxy is a single point of failure. All proxies should be in the same state, so, whenever one of them fails, the client could seamlessly continue to access the data from another proxy.

Consider now an advertising service involving a large number of cloud servers. The advertising service runs on a number of servers specialized for tasks such as: database access, monitoring, accounting, event logging, installers, customer dashboards,[1] advertising campaign planners, scenario testing, etc.

These activities can be coordinated using a configuration file shared by all systems. When the service starts, or after a system failure, all servers use the configuration file to coordinate their actions. This solution is static; any change requires an update and re-distribution of the configuration file. Moreover, in case of a system failure the configuration file does not allow recovery from the state each server was in prior to the system crash.

[1] A customer dashboard provides access to key customer information, such as contact name and account number, in an area of the screen that remains persistent as the user navigates trough multiple web pages.

A solution for the coordination problem is to implement a proxy as a deterministic finite-state machine transitioning from one state to the next in response to client commands. When P proxies are involved, all must be synchronized and execute the same sequence of state transitions upon receiving client commands. This scenario can be ensured when all proxies implement a version of the Paxos consensus algorithm described in Section 10.13.

Zookeeper is a distributed coordination service based on this model. The high throughput and low latency service is used for coordination in large-scale distributed systems. The open-source software is written in Java and has bindings for Java and C. The information about the project is available at http://zookeeper.apache.org/.

Zookeeper software must first be downloaded and installed on several servers. Then clients can connect to any server and access the coordination service. The service is available as long as the majority of servers in the pack are available.

The servers in the pack communicate with one another and elect a *leader*. A database is replicated on each of them, and the consistency of the replicas is maintained. Fig. 11.4(a) shows that the service provides a single system image; a client can connect to any server of the pack. A client uses TCP to connect to one server, then sends requests, receives responses, and watches events. A client synchronizes its clock with the server it is connected to. When a server fails, the TCP connections of all clients connected to it time-out, and the clients detect the failure and connect to other servers.

Figs. 11.4(b) and (c) show that a READ operation directed to any server in the pack returns the same result, while processing of a WRITE operation is more involved. The servers elect a *leader,* and any *follower* server forwards to the leader requests from the clients connected to it. The leader uses atomic broadcast to reach consensus. When the leader fails, the servers elect a new leader.

The system is organized as a shared hierarchical namespace similar to the organization of a file system. A name is a sequence of path elements separated by a backslash. Every name in Zookeeper's name space is identified by a unique path; see Fig. 11.5.

A *znode* of Zookeeper, the equivalent of an *inode* of UFS, may have data associated with it. The system is designed to store state information; the data in each node includes version numbers for the data, changes of ACLs,[2] and time stamps. A client can set a watch on a *znode* and receive a notification when the *znode* changes. This organization allows coordinated updates. The data retrieved by a client contains also a version number. Each update is stamped with a number that reflects the order of the transition.

Data stored in each node is read and written atomically, and a READ returns all data stored in a *znode*, while a WRITE replaces all data in the *znode*. Unlike a file system, the Zookeeper data, the image of the state, is stored in the server memory. Updates are logged to disk for recoverability, and WRITEs are serialized to disk before they are applied to the in-memory database containing the entire tree. Zookeeper service guarantees:

1. Atomicity—a transaction either completes or fails.
2. Updates sequential consistency—updates are applied strictly in the order in which they are received.
3. Single system image for the clients—a client receives the same response regardless of the server it connects to.

[2] An Access Control List (ACL) is a list of pairs (subject, value) that define the list of access rights to an object; for example, read, write, execute permissions for a file.

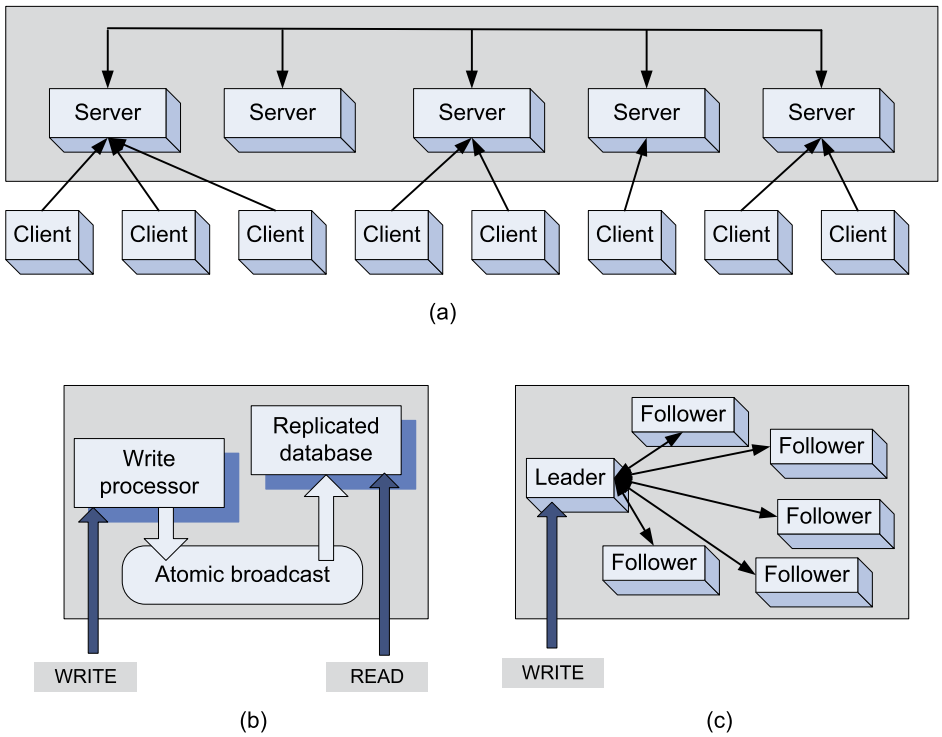

FIGURE 11.4

Zookeeper coordination service. (a) The service provides a single system image; clients can connect to any server in the pack. (b) The functional model of Zookeeper service; the replicated database is accessed directly by READ commands. (c) Processing a WRITE command: (i) a server receiving a command from a client, forwards it to the leader; (ii) the leader uses atomic broadcast to reach consensus among all followers.

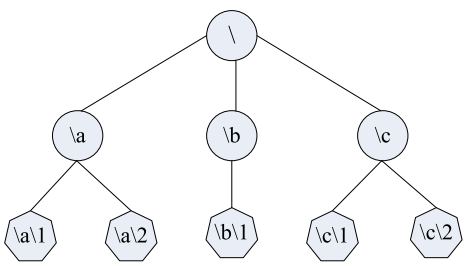

FIGURE 11.5

The Zookeeper is organized as a shared hierarchical namespace; a name is a sequence of path elements separated by a backslash.

4. Update persistence—once applied, an update persists until is overwritten by a client.
5. Reliability—the system is guaranteed to function correctly as long as the majority of servers function correctly.

READ requests are serviced from the local replica of the server connected to the client to reduce the response time. When the leader receives a WRITE request, it determines the state of the system where the WRITE will be applied, and then it transforms the state into a transaction capturing the new state.

The messaging layer is responsible for the election of a new leader when the current leader fails. The messaging protocol uses: *packets*—sequence of bytes sent through a FIFO channel, *proposals*—units of agreement, and *messages*—sequence of bytes atomically broadcast to all servers. A message is included into a proposal, and it is agreed upon before it is delivered. Proposals are agreed upon by exchanging packets with a quorum of servers as required by the Paxos algorithm.

An atomic messaging system keeps all of the servers in pack in sync. The messaging system guarantees: (a) Reliable delivery: if a message m is delivered to one server, it will be eventually delivered to all servers; (b) Total order: if message m is delivered before message n to one server, it will be delivered before n to all servers; and (c) Causal order: if message n is sent after m has been delivered by the sender of n, then m must be ordered before n.

Zookeeper Application Programming Interface (API) consists of seven operations: *create*—add a node at a given location on the tree; *delete*—delete a node; *get data*—read data from a node; *set data*—write data to a node; *get children*—retrieve a list of the children of the node; and *sync*—wait for the data to propagate. The system also supports the creation of *ephemeral* nodes, nodes created when a session starts and deleted when the session ends.

This brief description shows that the Zookeeper service supports the finite-state machine model of coordination where a *znode* stores the state. The Zookeeper service can be used to implement higher-level operations such as group membership, synchronization, and so on. Yahoo's Message Broker and many other applications use the Zookeeper service.

11.5 MapReduce programming model

A main advantage of cloud computing is elasticity, the ability to use as many servers as necessary to optimally respond to the cost and the timing constraints of an application. Typically, a front end of a transaction processing system distributes incoming transactions to a number of back-end systems and attempts to balance the workload. As the workload increases, new back-end systems are added to the pool. Many realistic applications in physics, biology, and other areas of computational science and engineering obey the arbitrarily divisible load sharing model; the workload can be partitioned into an arbitrarily large number of smaller workloads of equal, or nearly so, size. Yet, partitioning the workload of data-intensive applications is not always trivial.

MapReduce. MapReduce is based on a very simple idea for parallel processing of data-intensive applications supporting arbitrarily divisible load sharing; see Fig. 11.6. First, split the data into blocks, then assign each block to an instance/process and run these instances in parallel. Once all the instances have finished the computations assigned to them, start the second phase and merge the partial results produced by individual instances. The Same Program Multiple Data (SPMD) paradigm, used since the early days of parallel computing, is based on the same idea, but it assumes that a *Master* instance partitions the data and gathers the partial results.

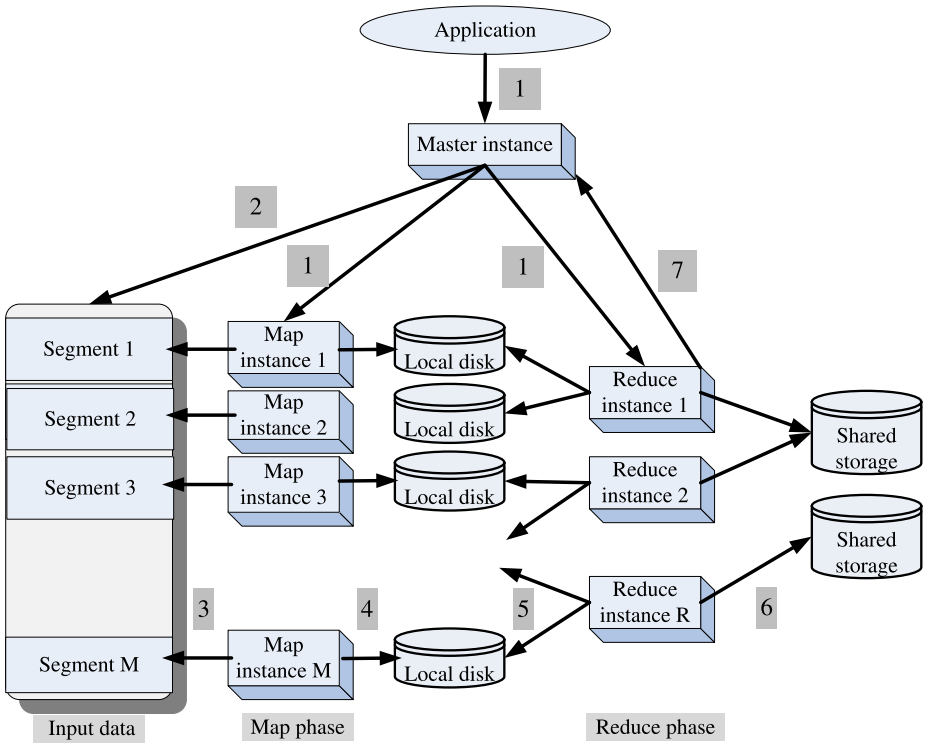

FIGURE 11.6

MapReduce philosophy. 1. An application starts a Master instance and M worker instances for the Map phase and later R worker instances for the Reduce phase. 2. The Master partitions the input data in M segments. 3. Each Map instance reads its input data segment and processes the data. 4. The results of the processing are stored on the local disks of the servers where the Map instances run. 5. When all Map instances have finished processing, their data R Reduce instances read the results of the first phase and merge the partial results. 6. The final results are written by Reduce instances to a shared storage server. 7. The Master instance monitors the Reduce instances, and when all of them report task completion, the application is terminated.

MapReduce is a programming model inspired by the *map* and the *reduce* primitives of the Lisp programming language. It was conceived for processing and generating large data sets on computing clusters [129]. As a result of the computation, a set of input $< key, value >$ pairs is transformed into a set of output $< key, value >$ pairs.

Numerous applications can be easily implemented using this model. For example, one can process logs of web page requests and count the URL access frequency; the Map and Reduce functions produce the pairs $< URL, 1 >$ and $< URL, totalcount >$, respectively. Another trivial example is *distributed sort* when the map function extracts the key from each record and produces a $< key, record >$ pair, and the Reduce function outputs these pairs unchanged. The following example [129] shows the two

user-defined functions for an application that counts the number of occurrences of each word in a set of documents.

```
map(String key, String value):
  // key: document name;  value: document contents
  for each word w in value:
  EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word;  values: a list of counts
  int result = 0;
  for each v in values:
  result += ParseInt(v);
  Emit(AsString(result));
```

Call M and R the number of Map and Reduce tasks, respectively, and N the number of systems used by the MapReduce. When a user program invokes the MapReduce function, the following sequence of actions take place:

- The run-time library splits the input files into M *splits* of 16 to 64 MB each, identifies a number N of systems to run, and starts multiple copies of the program, one of the systems being a Master and the others Workers. The Master assigns to each idle system either a *map* or a *reduce* task. The Master makes $\mathcal{O}(M + R)$ scheduling decisions and keeps $\mathcal{O}(M \times R)$ worker state vectors in memory. These considerations limit the size of M and R; at the same time, efficiency considerations require that $M, R \gg N$.
- A Worker being assigned a Map task reads the corresponding input split, parses $< key, value >$ pairs, and passes each pair to a user-defined Map function. The intermediate $< key, value >$ pairs produced by the Map function are buffered in memory before being written to a local disk, partitioned into R regions by the partitioning function.
- The locations of these buffered pairs on the local disk are passed back to the Master, which is responsible for forwarding these locations to the Reduce Workers. A Reduce Worker uses remote procedure calls to read the buffered data from the local disks of the Map Workers; after reading all the intermediate data, it sorts it by the intermediate keys. For each unique intermediate key, the key and the corresponding set of intermediate values are passed to a user-defined Reduce function. The output of the Reduce function is appended to a final output file.
- The Master wakes up the user program when all Map and Reduce tasks finish.

The system is fault-tolerant; for each Map and Reduce task, the Master stores the state (idle, in-progress, or completed) and the identity of the worker machine. The Master pings every worker periodically and marks the worker as failed if it does not respond; a task in progress on a failed worker is reset to idle and becomes eligible for rescheduling. The Master writes periodic checkpoints of its control data structures, and, if the task fails, it can be restarted from the last checkpoint. The data is stored using GFS, the Google File System discussed in Section 7.6.

An environment for experimenting with MapReduce is described in [129]: the computers are typically dual-processor *x86* processors running Linux, with 2–4 GB of memory per machine and commodity networking hardware typically 100–1 000 Mbps. A cluster consists of hundreds or thousands

of machines. Data is stored on IDE[3] disks attached directly to individual machines. The file system uses replication to provide availability and reliability with unreliable hardware. To minimize network bandwidth, the input data is stored on the local disks of each system.

MapReduce with FlumeJava. The Java library discussed in Section 10.15 supports a new operation called MapShuffleCombineReduce. This operation combines *ParallelDo, GroupByKey, CombineValues*, and *Flatten* operations into a single MapReduce [90]. This generalization of MapReduce supports multiple reducers and combiners and enables each reducer to produce multiple outputs, rather than enforcing the requirement that the reducer must produce outputs with the same key as its input. This solution enables the FlumeJava optimizer to produce better results.

M input channels, each performing a Map operation, feed into R output channels, each optionally performing a shuffle, an optional combine, and a Reduce operation. The *executor* of FlumeJava will run an operation locally if the input is relatively small. It will run parallel MapReduce remotely, though the overhead of launching a remote execution is larger, but the advantages for data larger inputs are significant. Temporary files for the outputs of all operations are created automatically and deleted when no longer necessary for later operations of the pipeline.

The system supports a *cached* execution mode when it first attempts to reuse the result of an operation from the previous run if it was saved in a (internal or user-visible) file and if it was not changed since. A result is unchanged if the inputs and the operation's code and saved state have not changed. This execution mode is useful for debugging an extended pipeline. Reference [90] reports that the largest pipeline had 820 unoptimized stages and 149 optimized stages.

11.6 Case study: the GrepTheWeb application

Many applications process massive amounts of data using the MapReduce programming model. An application, GrepTheWeb [487], now in production at Amazon, illustrates the power and the appeal of cloud computing. The application allows a user to define a regular expression and search the web for records that match it. GrepTheWeb is analogous to the *grep* Unix command used to search a file for a given regular expression.

This application performs a search of a very large set of records attempting to identify records that satisfy a regular expression. The source of this search is a collection of document URLs produced by Alexa Web Search, a software system that crawls the web every night. The inputs to the applications are: (i) the large data set produced by the web-crawling software, and (ii) a regular expression. The output is the set of records that satisfy the regular expression. The user is able to interact with the application and get the current status; see Fig. 11.7(a).

The application uses message passing to trigger the activities of multiple controller threads that launch the application, initiate processing, shutdown the system, and create billing records. GrepTheWeb uses Hadoop MapReduce, an open-source software package that splits a large data set into chunks, distributes them across multiple systems, launches the processing, and, when the processing is complete, aggregates the outputs from various systems into a final result. Apache Hadoop is a

[3] IDE (Integrated Drive Electronics) is an interface for connecting disk drives; the drive controller is integrated into the drive, as opposed to a separate controller on, or connected to, the motherboard.

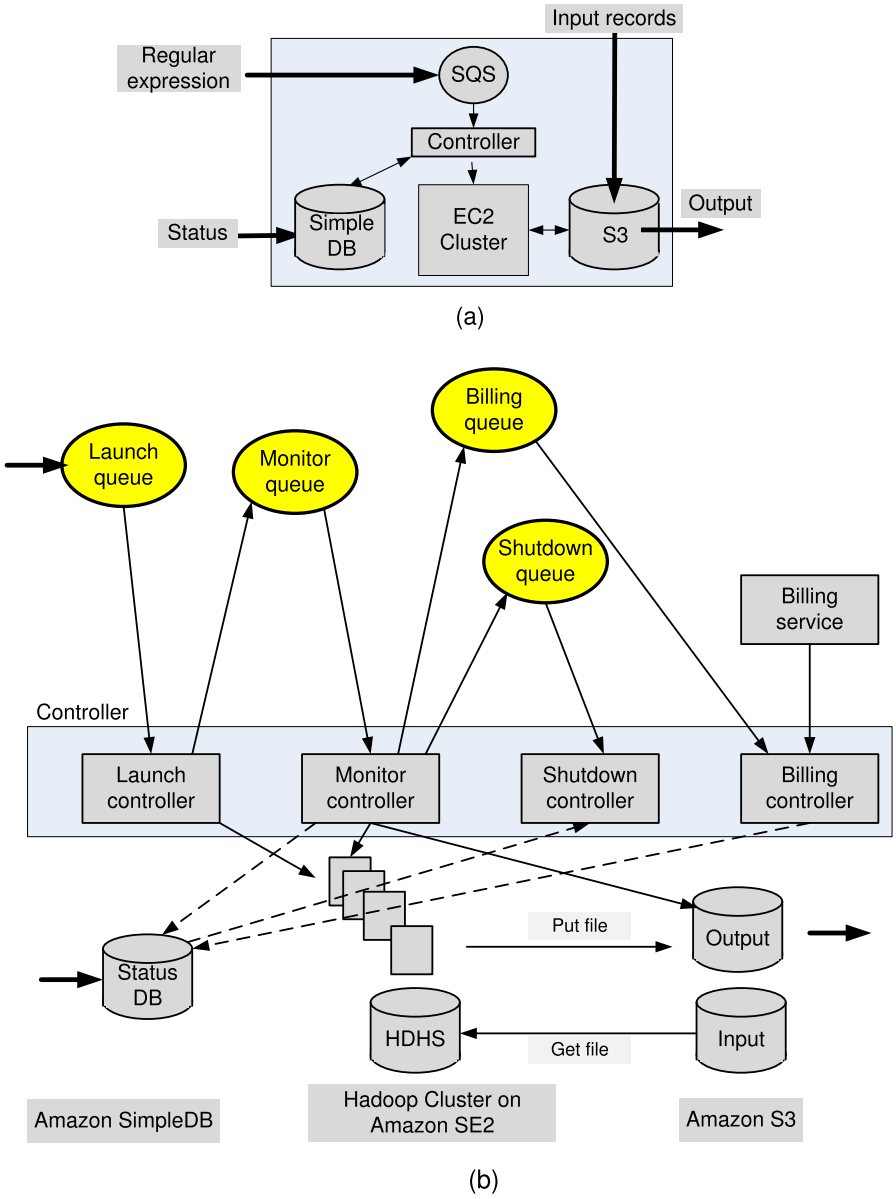

FIGURE 11.7

The organization of the GrepTheWeb application. The application uses the Hadoop MapReduce software and four Amazon services: EC2, Simple DB, S3, and SQS. (a) The simplified workflow showing the two inputs, the regular expression and the input records generated by the web crawler; a third type of input are the user commands to report the current status and to terminate the processing. (b) The detailed workflow; the system is based on message passing between several queues; four controller threads periodically poll their associated input queues, retrieve messages, and carry out the required actions.

software library for distributed processing of large data sets across clusters of computers using a simple programming model.

GrepTheWeb workflow, illustrated in Fig. 11.7(b), consists of the following steps [487]:

1. *The start-up phase:* create several queues to launch, monitor, billing, and shutdown queues; start the corresponding controller threads. Each thread polls periodically its input queue and when a message is available, retrieves the message, parses it, and takes the required actions.
2. *The processing phase:* it is triggered by a *StartGrep* user request; then a launch message is enqueued in the launch queue. The launch controller thread picks up the message and executes the launch task; then, it updates the status and time stamps in the Amazon *Simple DB* domain. Lastly, it enqueues a message in the monitor queue and deletes the message from the launch queue. The processing phase consists of the following steps:
   a. The launch task starts Amazon EC2 instances: it uses a Java Runtime Environment preinstalled Amazon Machine Image (AMI), deploys required Hadoop libraries, and starts a Hadoop job (run Map/Reduce tasks).
   b. Hadoop runs Map tasks on EC2 slave nodes in parallel: a Map task takes files from S3, runs a regular expression, and writes locally the match results along with a description of up to five matches; then, the Combine/Reduce task combines and sorts the results and consolidates the output.
   c. Final results are stored on Amazon S3 in the output bucket.
3. *The monitoring phase:* the monitor controller thread retrieves the message left at the beginning of the processing phase, validates the status/error in Simple DB, and executes the monitor task; it updates the status in the Simple DB domain and enqueues messages in the shutdown and the billing queues. The monitor task checks for the Hadoop status periodically and updates the Simple DB items with status/error and the S3 output file. Finally, it deletes the message from the monitor queue when the processing is completed.
4. *The shutdown phase:* the shutdown controller thread retrieves the message from the shutdown queue and executes the shutdown task that updates the status and time stamps in the Simple DB domain; finally, it deletes the message from the shutdown queue after processing. The shutdown phase consists of the following steps:
   a. The shutdown task kills the Hadoop processes, terminates the EC2 instances after getting EC2 topology information from Simple DB, and disposes of the infrastructure.
   b. The billing task gets the EC2 topology information, Simple DB usage, S3 file and query input, calculates the charges, and passes the information to the billing service.
5. *The cleanup phase:* archives the Simple DB data with user info.
6. *User interactions with the system:* get the status and output results. The *GetStatus* is applied to the service endpoint to obtain the status of the overall system (all controllers and Hadoop) and download the filtered results from S3 after completion.

Multiple S3 files are bundled up and stored as S3 objects to optimize the end-to-end transfer rates in the S3 storage system. Another performance optimization is to run a script and sort the keys, the URL pointers, and upload them in sorted order in S3. Multiple fetch threads are started to fetch the objects. This application illustrates the means to create an on-demand infrastructure and run it on a massively distributed system in a manner that enables it to run in parallel and scale up and down, based on the number of users and the problem size.

11.7 Hadoop, Yarn, and Tez

A wide range of data-intensive applications, such as marketing analytics, image processing, machine learning, and web crawling, use the Apache Hadoop, an open-source, Java-based software system.[4] Hadoop supports distributed applications handling extremely large volumes of data. Many members of the community contributed to the development and optimization of Hadoop and of several related Apache projects such as Hive and HBase.

Hadoop is used by many organization from industry, government, and research. The long list of Hadoop users includes major IT companies, e.g., Apple, IBM, HP, Microsoft, Yahoo, and Amazon, media companies, e.g., *The New York Times* and Fox, social networks including, Twitter, Facebook, and Linkedln, and government agencies such as the Federal Reserve. In 2012, the Facebook Hadoop cluster had a capacity of 100 petabytes and was growing at a rate of 0.5 petabytes a day. In 2013, more than half of Fortune 500 companies were using Hadoop. Azure HDInsight service deploys Hadoop on Microsoft Azure.

Hadoop. Apache Hadoop is an open-source software framework for distributed storage and distributed processing based on the MapReduce programming model. Recall that in MapReduce the Map stage processes the raw input data, one data item at a time, and produces a stream of data items annotated with keys. Next, a local sort stage orders the data produced during the Map stage by key. The locally ordered data is then passed to an (optional) combiner stage for partial aggregation by key. The shuffle stage then redistributes data among machines to achieve a global organization of data by key.

A Hadoop system has two components, a MapReduce engine and a database; see Fig. 11.8. The database could be the Hadoop File System (HDFS), Amazon S3, or CloudStore, an implementation of GFS, discussed in Section 7.6. HDFS is a highly performant distributed file system written in Java. HDFS is portable but cannot be directly mounted on an existing operating system; it is not fully POSIX compliant.

The Hadoop engine on the master of a multinode cluster consists of a *job tracker* and a *task tracker*, while the engine on a slave has only a *task tracker*. The *job tracker* receives a MapReduce job from a client and dispatches the work to the *task trackers* running on the nodes of a cluster. To increase efficiency, the *job tracker* attempts to dispatch the tasks to the available slaves closest to the place where the task data was stored. The *task tracker* supervises the execution of the work allocated to the node; several scheduling algorithms have been implemented in Hadoop engines, including Facebook's fair scheduler and Yahoo's capacity schedulers.

HDFS replicates data on multiple nodes; the default is three replicas, and a large dataset is distributed over many nodes. The *name node* running on the master manages the data distribution and data replication and communicates with *data nodes* running on all cluster nodes; it shares with the *job tracker* information about the data placement to minimize communication between the nodes where data is located and the ones where it is needed. Although HDFS can be used for applications other than those based on the MapReduce model, its performance for such applications is not on par with the ones it was originally designed for.

Hadoop brings computations to the data on clusters built with off-the-shelf components. This strategy is pushed further by Spark, which stores data in processor's memory instead of the disk. Data

[4] Hadoop requires JRE (Java Runtime Environment) 1.6 or higher.

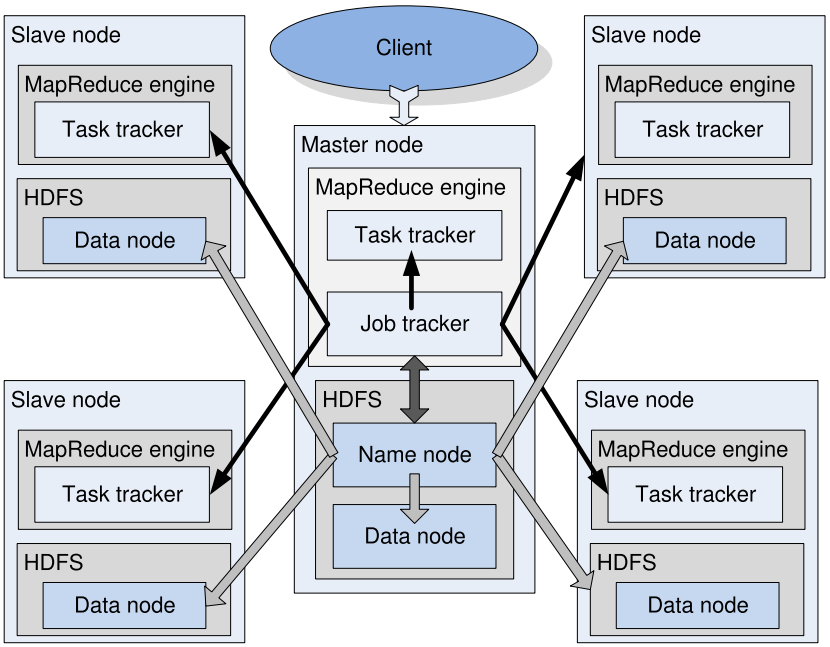

FIGURE 11.8

A Hadoop cluster using HDFS; the cluster includes a master and four slave nodes. Each node runs a MapReduce engine and a database engine, often HDFS. The *job tracker* of the master's engine communicates with the *task trackers* on all the nodes and with the *name node* of HDFS. The *name node* of the HDFS shares information about the data placement with the *job tracker* to minimize communication between the nodes where data is located and the ones where it is needed.

locality allows Hadoop and Spark to compete with traditional High Performance Computing (HPC) running on supercomputers with high-bandwidth storage and faster interconnection networks.

The Apache Hadoop framework has the following modules:

1. Common—contains libraries and utilities needed by all Hadoop modules.
2. Distributed File System (HDFS)—a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster.
3. Yarn—a resource-management platform responsible for managing computing resources in clusters and using them for scheduling of users' applications.
4. MapReduce—an implementation of the MapReduce programming model.

Additional software packages such as Apache Pig, Apache Hive, Apache HBase, Apache Phoenix, Apache Spark, Apache ZooKeeper, Cloudera Impala, Apache Flume, Apache Sqoop, Apache Oozie, and Apache Storm are also available.

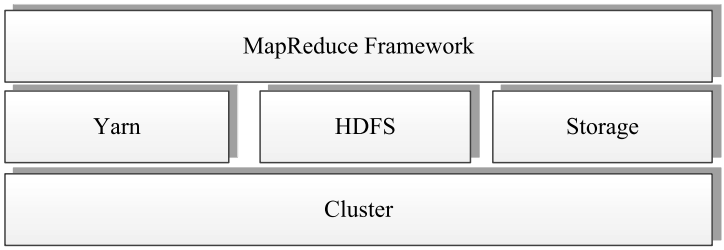

FIGURE 11.9

Resources needed by the MapReduce framework are provided by the cluster and are managed by Yarn; HDFS provides permanent, reliable, and distributed storage; multiple organizations of the storage system are supported, e.g., AWS implementation of Hadoop offers S3.

Several Hadoop frameworks are used to manage and run deep analytics. SQL processing is particularly important to gain insights from large collections of data and the number of SQL-on-Hadoop systems has increased. There are many SQL engines running on Hadoop, including BigSQL from IBM, Impala from Cloudera, and HAWQ from Pivotal. All these engines implement a standard language specification and compete on performance and extended services. Users are insulated from difficulties because the applications that talk to those engines are portable.

Systems such as Pig, Hive, and Impala, discussed in Section 11.8, are native Hadoop-based systems, or database-Hadoop hybrids. HadoopOthers, such as *Hadapt* [5], exploits Hadoop scheduling and fault-tolerance, but uses a relational database, PostgreSQL, to execute query fragments.

Yarn. Yarn is a resource management system supplying CPU cycles, memory, and other resources needed by a single job or to a DAG of MapReduce applications. Resource allocation and job scheduling in Hadoop versions prior to 2.0 were done by the MapReduce framework. In the newer versions of Hadoop, Yarn carries out these functions. This new system organization allows frameworks such a Spark to share cluster resources. The organization of Hadoop including Yarn in Fig. 11.9 shows the three system elements: (1) the MapReduce framework; (2) Yarn, HDFS, and the storage substrate; and (3) the cluster where the application is running.

Fig. 11.10 presents the organization of Yarn and shows the Resource Manager and Node Managers running in each node. A node manager is responsible for containers, monitors their resource usage (CPU, memory, disk, network), and reports to the resource manager tasked to arbitrate resources sharing among all applications. Each application has an Application Manager that negotiates with the resource manager the access to resources needed by the application. Once resources are allocated, the application manager interacts with the node managers of each node allocated to the application to start the tasks and then monitors their execution.

The Scheduler component of the resource manager uses the resource container abstraction that incorporates memory, CPU, disk, network, etc. and bases the resource allocation decisions an applications. The scheduler performs no monitoring or tracking application status and offers no guarantees about restarting failed tasks. A pluggable policy is responsible for sharing cluster resources among applications. The Capacity Scheduler and the Fair Scheduler are examples of scheduler plug-ins.

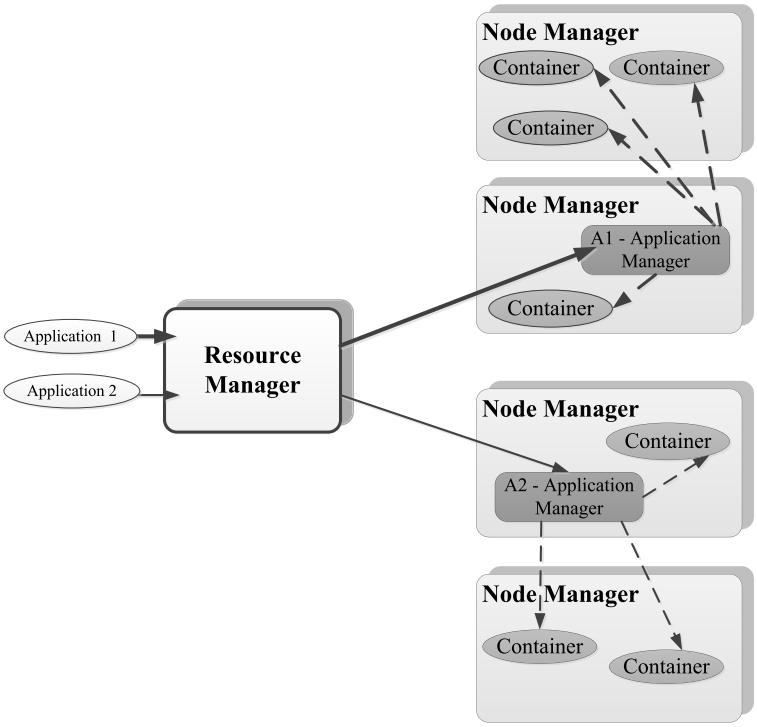

FIGURE 11.10

Yarn organization and application processing. Applications are submitted to the *Resource Manager*. The Resource Manager communicates with Node Managers running on every node of the cluster. The tasks of every application are managed by an Application Manager. Each task is packaged in a container.

MapReduce in Hadoop-2.x maintains API compatibility with previous stable release, thus MapReduce jobs should still run unchanged on top of Yarn after a recompile.

The process of starting an application involves several steps, illustrated in Fig. 11.11:

1. The user submits an application to the Resource Manager.
2. The Resource Manager invokes the Scheduler and allocates a container for the Application Manager.
3. The Resource Manager contacts the Node Manager where the container will be launched.
4. The Node Manager launches the container.
5. The container executes the Application Manager.
6. The Resource Manager contacts the Node Manager(s) where the tasks of the application will run.
7. Containers for the tasks of the application are created.
8. The Application Manager monitors the execution of the tasks until termination.

Tez. Tez is an extensible framework for building high-performance batch and interactive processing for Yarn-based applications in Hadoop. A job is decomposed into individual tasks, and each task of the job

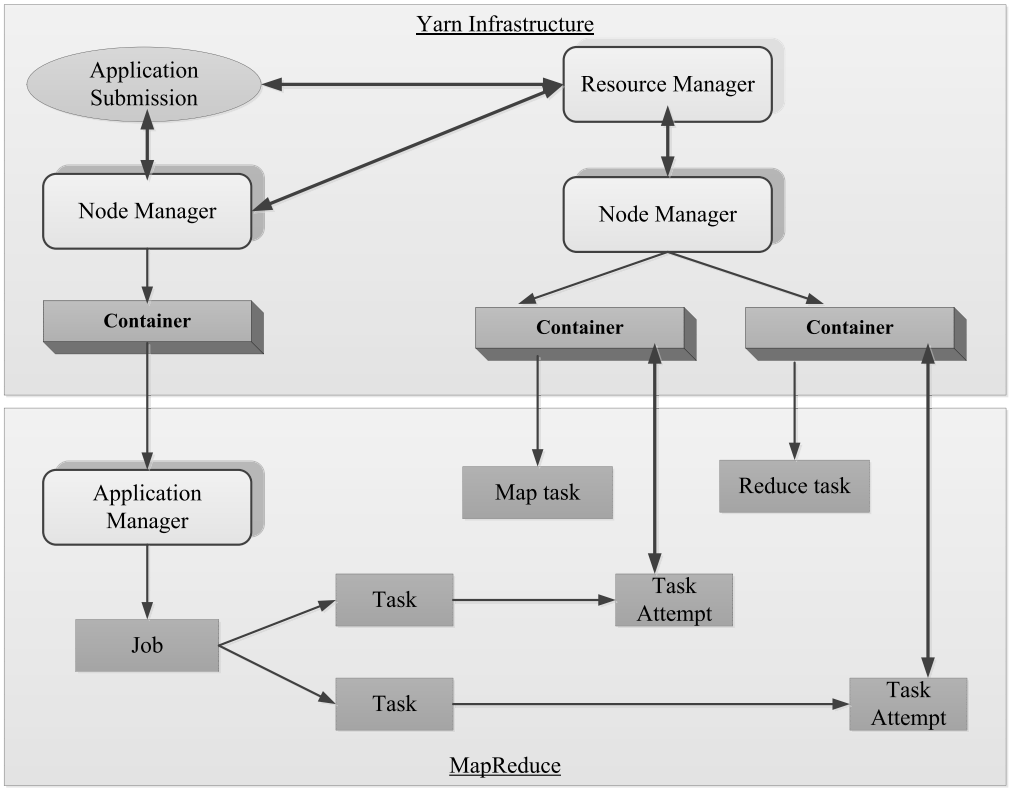

FIGURE 11.11

The interaction among the components of Yarn and MapReduce. Once an application is submitted, Yarn's Resource Manager contacts a Node Manager to create a container for the Application Manager. Then, the application manager is activated. The resource manager with the assistance of the scheduler selects the node manager(s) where the containers for the application tasks are created. Then, the containers start the execution of these tasks.

runs as an Yarn process. Tez models data processing as a DAG; the graph vertices represent application logic, and edges represent movement of data. A Java API is used to express the DAG representation of the workflow. The execution engine uses Yarn to acquire resources and reuses every component in the pipeline to avoid operation duplication. Apache Hive and Apache Pig use Apache Tez to improve the speed of MapReduce applications; see http://hortonworks.com/apache/tez/.

11.8 SQL on Hadoop: Pig, Hive, and Impala

The landscape of parallel SQL database vendors prior to 2009 included IBM, Oracle, Microsoft, ParAccel, Greenplum, Teradata, Netezza, and Vertica, but none of them were supporting SQL queries along

with MapReduce jobs. From their early years in business, Yahoo and Facebook used the MapReduce platform extensively to store, process, and analyze huge amounts of data. Both companies wanted a faster and easier-to-work-with platform supporting SQL queries.

MapReduce is a heavyweight, high-latency execution framework and does not support workflows, join operations for combined processing of several datasets, filtering, aggregation, top-k thresholding, and high-level operations. To address these challenges, Yahoo created a dataflow system called Pig in 2009. Facebook built Hive on MapReduce because it was the shortest path to SQL on Hadoop. Pig and Hive have their own language, Pig Latin and HiveQL, respectively. In both systems, a user types a query, then a parser reads the query, figures out what the user wants, and runs a series of MapReduce jobs. That was a sensible decision given the requirements of the time.

In 2012, Cloudera developed Impala and published it under an Apache license. Later, Facebook developed Presto, its next-generation query processing engine for real-time access to data via SQL. It was built, like Hive, from the ground up, as a distributed query processing engine.

Pig. The system presented in [190] supports workflows, joint operations for combined processing of several data sets, filtering, aggregation, and high-level operations. Pig compiles programs written in a language called Pig Latin into a set of Hadoop jobs and coordinates their execution. It also supports several user interaction modes: (1) Interactive—using a shell for Pig commands; (2) Batch—a user submits a script containing a series of Pig commands; and (3) Embedded—commands are submitted via method invocation from a Java program. The job processing stages are:

1. Parsing—the parser performs syntactic verification, type checking, and schema inference and produces a DAG called a Logical Plan. In this plan, an operator is annotated with the schema of its output data, with braces indicating a bag of tuples.
2. Logical optimization of the DAG and creation of a Physical Plan describing data distribution.
3. Compilation of the optimized Physical Plan into a set of MapReduce jobs, followed by optimization phase, e.g., partial aggregation, resulting in an optimized DAG. Distributive and algebraic aggregation functions, e.g., AVERAGE, are broken into series of three steps: initial (e.g., generate [sum, count] pairs), intermediate (e.g., combine n [sum, count] pairs into a single pair), final (e.g., combine n [sum, count] pairs and take the quotient). These steps are assigned to the Map, Combine, and Reduce stages, respectively.
4. The DAG is topologically sorted and jobs are submitted to Hadoop for execution. The flow control is implemented using a pull model with a single threaded implementation and a simple API for user-defined functions for moving tuples through the execution pipeline. An operator can respond in one of three ways when asked to produce a tuple: return the tuple, declare that it has finished, or return a pause signal indicating that either it is not finished or unable to produce an output tuple.

Compilation and execution is triggered by the STORE command. If a Pig Latin program contains more than one STORE command, the generated physical plan contains a SPLIT physical operator. Fig. 11.12 illustrates the transformation of a Pig Latin program into a Logical Plan, followed by the transformation of the Logical Plan into a Physical Plan, and finally the transformation of the Physical Plan into a MapReduce Plan.

Memory management is challenging because Pig is implemented in Java, and program controlled memory allocation and deallocation is not feasible. A handler can be registered using the *MemoryPoolMXBean* Java class. The handler is notified whenever a configurable memory threshold is reached; then the system releases registered bags until enough memory is freed.

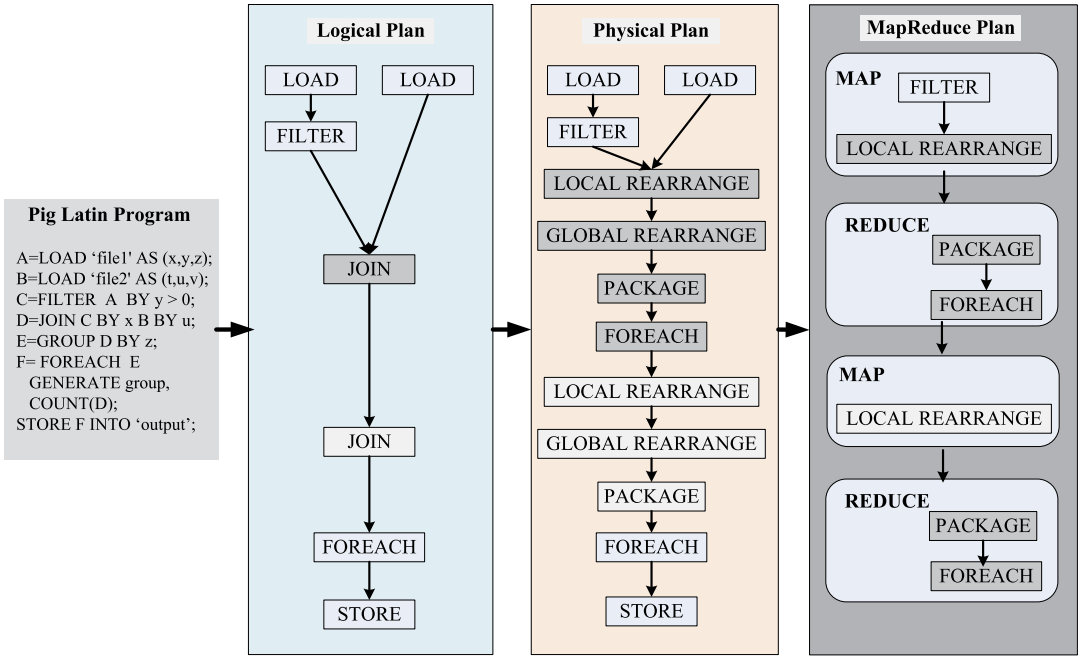

FIGURE 11.12

The transformation of a Pig Latin program into a Logical Plan, followed by the transformation of the Logical Plan into a Physical Plan, and finally the transformation of the Physical Plan into a MapReduce Plan. Each one of the two JOINs in the Logical Plan generate LOCAL REARRANGE, GLOBAL REARRANGE, PACKAGE, and FOREACH statements. A pair of MAP, REDUCE) is then generated in the MapReduce Plan.

Pig Mix is a benchmark used at Yahoo and elsewhere to evaluate system's performance and to exercise a wide range of the system functionality. About 60% of the ad hoc Hadoop jobs at Yahoo use Pig. The use of the system outside Yahoo has been increasing.

Hive. Hive an open-source system for data-warehousing supporting queries expressed in an SQL-like declarative language called HiveQL [475]. These queries are then compiled into MapReduce jobs executed on Hadoop. The system includes the Metastore, a catalog for schemas, and the statistics used for query optimization.

Hive supports data organized as tables, partitions, and buckets. *Tables* are inherited from relational databases and can be stored internally or externally in HDFS, NFS, or a local directory. A table is serialized and stored in the files of an HDFS directory. The format of each serialized table is stored in the system catalog, and it is accessed during query compilation and execution.

Partitions are components of tables described by the subdirectories of table directory. In turn, *buckets* consists of partition data stored as a file in the partition's directory and selected based on the hash of one of the columns of the table. The query language supports data-definition statements to create ta-

bles with specific serialization formats and partitioning and bucketing columns, as well as user-defined column transformation and aggregation functions implemented in Java.

HiveQL accepts as input DDL, DML and allows user-defined MapReduce scripts written in any language using a simple row-based streaming interface. Data Description Language (DDL) has a syntax for defining data structures similar to a computer programming language, and it is widely used for database schemas. Data Manipulation Language (DML) is used to retrieve, store, modify, delete, insert, and update data in a database; SELECT, UPDATE, INSERT statements or query statements are examples of DML.

The system has several components:

- *External Interface*—includes a command line (CLI), a web user-interface (UI), and language APIs such as JDBC and ODBC.[5]
- *Thrift Server*—a client API for execution of HiveQL statements.[6] Java and C++ clients can be used to build JDBC or ODBC common drivers, respectively.
- *Megastore*—the system catalog. It contains several objects: (1) Database—a namespace for tables; (2) Table—contains the list of columns and their types, owner, storage, the location of the table data, data formats, and bucketing information; (3) Partition—each partition can have its own columns and storage information.
- *Driver*—manages the compilation, optimization, and execution of HiveQL statements.
- *Database*—the namespace for tables.
- *Table*—the metadata for tables containing the list of columns and their types, owner, storage, and a wealth of other data including the location of the table data, data formats, and bucketing information. It also includes SerDe metadata regarding the implementation class of serializer and deserializer methods and information required by their implementation.
- *Partition*—information about the columns in a partition including SerDe and storage information.

The HiveQL compiler has several components: the *Parser* transforms an input string into a parse tree; then, the *Semantic Analyzer* transforms this tree into an internal representation, the *Logical Plan Generator* converts this internal representation into a *Logical Plan*, and finally the *Optimizer* rewrites the logical plan.

Facebook's Hive warehouse contains over 700 TB of data and supports more than 5 000 daily queries. Hive is an Apache project, with an active user and developer community.

Impala is a query engine exploiting a shared-nothing parallel database architecture written in C++ and Java and designed to use standard Hadoop components (HDFS, HBase, Metastore, Yarn, Sentry) [282]. The system, developed by Cloudera and published under an Apache license in 2012, is designed from the ground up for SQL query execution on Hadoop, rather than a general-purpose distributed processing system. It delivers better performance than Hive because it does not translate an SQL query into another processing framework as Hive does. It supports most of the SQL-92 SELECT statement syntax and SQL-2003 analytic functions. It does not support UPDATE or DELETE, but supports bulk insertions $INSERT\,INTO\,O\ldots, SELECT\ldots$.

[5] Java Database Connectivity (JDBC) is an API for Java, defining how a client may access a database. Open Database Connectivity (ODBC) is an open standard application API for accessing a database.

[6] Thrift is a framework for cross-language services; see Apache Thrift, http://incubator.apache.org/thrift.

Every node in a Cloudera cluster has Impala code installed and waiting for SQL queries to execute. The Impala code lives on every node alongside MapReduce, Apache HBase, and third-party engines such as SAS and Apache Spark that a customer may choose to deploy. All these engines have access to the same data, and users can choose one of them depending on the application. Impala runs queries using long-running daemons on every HDFS DataNode, and pipelines intermediate results between computation stages.

The I/O layer of Impala spawns one I/O thread per disk on each node to read data stored in HDFS and achieves high utilization of both CPU and disks by decoupling asynchronous read requests from the synchronous actual reading of data. The system exploits Intel's SSE instructions discussed in Section 3.4 to parse and process textual data efficiently. Impala requires the working set of a query to fit in the aggregate physical memory of the cluster.

The system operates basic services offered by three daemons:

1. *Impalad*—accepts, plans, and coordinates query execution. An Impalad daemon is deployed on every server and operates a query planner, a query coordinator, and a query executor. The front end compiles SQL text into query plans executable by the back ends. In this stage, a parse tree is translated into a single-node plan tree including: HDFS/HBase scan, hash join, cross join, union, hash aggregation, sort, top-n, and analytic evaluation nodes. A second phase transforms the single-node plan into a distributed execution plan, and in this process, the goal is to minimize data movement and maximize scan locality.
2. *Statestored*—provides a metadata publish-subscribe service and disseminates cluster-wide metadata to all processes. It maintains a set of topics (key, value, version), triplets defined by an application. Processes wishing to receive updates to any topic express their interest by registering at start-up and providing a list of topics.
3. *Catalogd*—is the catalog repository and metadata access gateway. It pulls information from third-party metadata stores and aggregates it. Only a skeleton entry for each table it discovers is loaded at startup; then, table metadata is loaded in the background from third-party stores.

A recent paper [176] compares the performance of Impala and Hive. Both systems input their data from columnar storage in a Parquet and the Optimized Row Columnar, the Apache columnar storage formats shared by the software in the Hadoop ecosystem, regardless of the choice of data processing framework, data model, or programming language. Columnar formats improve the performance of queries in the context of relational databases.

Parquet stores data grouped together in logical horizontal partitions called *row groups*. Every row group contains a *column chunk* for each column in the table. A column chunk consists of multiple *pages* and is guaranteed to be stored contiguously on disk. Compression and encoding schemes work at a page level. Metadata is stored at all the levels in the hierarchy, i.e., file, column chunk, and page. An ORC file stores multiple groups of row data as *stripes* and has a *footer* containing the list of the stripes in the file, the number of rows stored in each stripe, the data type of each column, and column-level aggregates, such as: count, sum, min, and max.

The experiments in [176] used Hive version 0.12 (Hive-MR) and Impala version 1.2.2 on top of Hadoop 2.0.0-cdh4.5.0, Hive version 0.13 (Hive-Tez) on top of Tez 0.3.0, and *Apache Hadoop 2.3.0.1*. Hadoop was configured to run 12 containers per node, 1 per core. The HDFS replication factor is three, and the maximum JVM heap size is set to 7.5 GB per task. Impala uses MySQL as the metastore. One Impalad process runs on each compute node and has access to 90 GB of memory.

One of the nodes of a 21-node cluster used for the measurements hosts the HDFS NameNode, and there are 20 compute nodes. Each node runs 64-bit Ubuntu Linux 12.04 and has: one Intel Xeon CPUs @ 2.20GHz with 6 cores, eleven SATA disks (2TB, 7k RPM), one 10-Gigabit Ethernet card, and 96 GB of RAM. One out of the eleven SATA disks in each node hosts the OS, and the rest are used for HDFS.

The experiments reported in [176] run the 22 TPC-H[7] queries for Hive and Impala and report the execution time for each query. The file cache is flushed before each run in all compute nodes. The results show that Impala outperforms both *Hive-MapReduce* and Hive-Tez for all file formats, with or without compression; the improvements are in the range 1.5–13.5 times. Several factors contribute to the drastic performance boost:

- Impala has a more efficient I/O subsystem than Hive-MR and Hive-Tez.
- Long-running daemon processes handle queries in each node thus, the overhead of job initialization and scheduling due to MapReduce in Hive-MR is eliminated.
- The query execution is pipelined, while in Hive-MR data is written out at the end of each step and read in by subsequent step(s).

Impala uses code generation to eliminate the overhead of virtual function calls, inefficient instruction branching due to large switch statements, and of other sources of inefficiency. Query execution time improves by about 1.3 times for all 21 TPC-H queries combined when this feature is enabled at runtime. TPC-H Query 1 shows the largest improvement, 5.5 times, while the remaining queries improve up to 1.75 times.

A second set of experiments involve TPC-DS benchmark.[8] Results show that Impala is on average 8.2 times faster than Hive-MR and 4.3 times faster than Hive-Tez on TPC-DS and 10 times faster than Hive-MR and 4.4 times faster than Hive-Tez on a second workload. The first workload consists of 20 queries that access a single fact table and six dimension tables, while the second uses the same workload but removes the explicit partitioning predicate and uses the correct predicate values.

11.9 Current cloud applications and new applications opportunities

Existing cloud applications can be divided in several broad categories: (i) processing pipelines; (ii) batch processing systems; and (iii) web applications [487].

Processing pipelines are data-intensive and sometimes compute-intensive applications that represent a fairly large segment of applications currently running on a cloud. Several types of data processing applications can be identified:

- Indexing; processing pipeline supports indexing of large datasets created by web-crawler engines.
- Data mining; processing pipeline supports searching large sets of records to locate items of interests.

[7] The TPC BenchmarkH (TPC-H) is a decision-support benchmark consisting of a suite of business-oriented ad hoc queries and concurrent data modifications chosen to have broad industry-wide relevance. This benchmark is relevant for applications examining a large volume of data and executing queries with a high degree of complexity.

[8] TPC-DS is a de-facto industry standard benchmark for assessing the performance of decision support systems.

- Image processing; a number of companies allow users to store their images on the cloud, e.g., Flickr (flickr.com) and Picasa (http://picasa.google.com/). Image-processing pipelines support image conversion, e.g., enlarge an image or create thumbnails, and compress or encrypt images.
- Video transcoding; processing pipeline translates one video format to another, e.g., AVI to MPEG.
- Document processing; processing pipeline converts very large collection of documents from one format to another, e.g., from Word to PDF or encrypts the documents; they could also use OCR (Optical Character Recognition) to produce digital images of documents.

Batch processing systems also cover a broad spectrum of data-intensive applications in enterprise computing. Such applications typically have deadlines, and the failure to meet these deadlines could have serious economic consequences; security is also a critical aspect for many applications of batch processing. A nonexhaustive list of batch processing applications includes:

- Generation of daily, weekly, monthly, and annual activity reports for organizations in retail, manufacturing, and other economic sectors.
- Processing, aggregation, and summaries of daily transactions for financial institutions, insurance companies, and healthcare organizations.
- Inventory management for large corporations.
- Processing billing and payroll records.
- Management of the software development, e.g., nightly updates of software repositories.
- Automatic testing and verification of software and hardware systems.

Lastly, cloud applications in the area of web access are of increasing importance. Several groups of web sites have a periodic or temporary presence, e.g., web sites for conferences or other events. There are also web sites active during a particular season (e.g., the holiday season) or supporting a particular type of activity, such as US income tax reporting with the April 15 deadline each year. Other limited-time web site are used for promotional activities, or web sites that "sleep" during the night and auto-scale during the day.

It makes economic sense to store the data in the cloud close to where the application runs; as we have seen in Section 2.2, the cost per GB is low, and the processing is much more efficient when the data is stored close to the computational servers. This leads us to believe that several new classes of cloud computing applications could emerge in the years to come, for example, batch processing for decision support systems and other aspects of business analytics.

Another class of new applications could be parallel batch processing based on programming abstractions such as MapReduce, discussed in Section 11.5. Mobile interactive applications that process large volumes of data from different types of sensors and services that combine more than one data source, e.g., mashups,[9] are obvious candidates for cloud computing.

Science and engineering could benefit from cloud computing as many applications are compute- and data-intensive. Similarly, a cloud dedicated to education would be extremely useful. Mathematical software, e.g., MATLAB® and Mathematica, could also run on the cloud.

[9] A mashup is an application that uses and combines data, presentations, or functionality from two or more sources to create a service. The fast integration, frequently using open APIs and multiple data sources, produces results not envisioned by the original services; combination, visualization, and aggregation are the main attributes of mashups.

11.10 Clouds for science and engineering

For more than two thousand years, science was empirical. Several hundred years ago, theoretical methods based on models and generalization were introduced, and this propelled substantial progress in human knowledge. In the last few decades, we have witnessed the explosion of computational science based on the simulation of complex phenomena.

In a talk delivered in 2007 and posted on his web site just before he went missing in January 2007, Jim Gray discussed *eScience* as a transformative scientific method [235]. Today, *eScience* unifies experiment, theory, and simulation; data captured from measuring instruments, or generated by simulations is processed by software systems; data and knowledge are stored by computer systems and analyzed with statistical packages.

The generic problems in virtually all areas of science are: (1) collection of experimental data; (2) management of a very large volumes of data; (3) building and execution of models; (4) integration of data and literature; (5) documentation of the experiments; (6) sharing the data with others; and (7) data preservation for long periods of time. All these activities require powerful computing systems.

A typical example of a problem faced by agencies and research groups is data discovery in large scientific data sets. Examples of such large collections are the biomedical and genomic data at NCBI,[10] the astrophysics data from NASA,[11] and the atmospheric data from NOAA[12] and NCAR.[13] The process of online data discovery can be viewed as an ensemble of several phases: (i) recognition of the information problem; (ii) generation of search queries using one or more search engines; (iii) evaluation of the search results; (iv) evaluation of the web documents; and (v) comparing information from different sources. The web-search technology enables the scientists to discover text documents related to such data, but the binary encoding of many of them poses serious challenges.

High Performance Computing on AWS. Reference [256] describes the set of applications used at NERSC (National Energy Research Scientific Computing Center) and presents the results of a comparative benchmark of EC2 and three supercomputers. NERSC is located at Lawrence Berkeley National Laboratory and serves a diverse community of scientists; it has some 3 000 researchers and involves 400 projects based on some 600 codes. Some of the codes used are:

> CAM (Community Atmosphere Mode), the atmospheric component of CCSM (Community Climate System Model), is used for weather and climate modeling.[14] The code developed at NCAR uses two two-dimensional domain decompositions; one for the dynamics and the other for re-mapping. The first is decomposed over latitude and vertical level and the second is decomposed over longitude–latitude. The program is communication-intensive; on-node/processor data movement and relatively long MPI[15] messages that stress the interconnect point-to-point bandwidth are used to move data between the two decompositions.

[10] NCBI is the National Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov/.
[11] NASA is the National Aeronautics and Space Administration, http://www.nasa.gov/.
[12] NOAA is the National Oceanic and Atmospheric Administration, www.noaa.gov).
[13] NCAR is the National Center for Atmospheric Research.
[14] See http://www.nersc.gov/research-and-development/benchmarking-and-workload-characterization.
[15] MPI, Message Passing Interface, is a communication library based on a standard for a portable message-passing system.

GAMESS (General Atomic and Molecular Electronic Structure System) is used for *ab initio* quantum chemistry calculations. The code developed by the Gordon research group at the Department of Energy's Ames Lab at Iowa State University has its own communication library, the Distributed Data Interface (DDI) that is based on the SPMD (Same Program Multiple Data) execution model. DDI presents the abstraction of a global shared memory with one-side data transfers, even on systems with physically distributed memory. On the cluster systems at NERSC, the program uses socket communication; on the Cray XT4, the DDI uses MPI and only one-half of the processors compute, while the other half are data movers. The program is memory- and communication-intensive.

GTC (Gyrokinetic[16]) is a code for fusion research.[17] It is a self-consistent, gyrokinetic tridimensional Particle-in-cell (PIC)[18] code with a non-spectral Poisson solver; it uses a grid that follows the field lines as they twist around a toroidal geometry representing a magnetically confined toroidal fusion plasma. The version of GTC used at NERSC uses a fixed, one-dimensional domain decomposition with 64 domains and 64 MPI tasks. Communication is dominated by the nearest neighbor exchanges that are bandwidth-bound. The most computationally intensive parts of GTC involve gather/deposition of charge on the grid and particle "push" steps. The code is memory intensive because the charge deposition uses indirect addressing.

IMPACT-T (Integrated Map and Particle Accelerator Tracking Time) is a code for the prediction and performance enhancement of accelerators; it models the arbitrary overlap of fields from beamline elements and uses a parallel, relativistic PIC method with a spectral integrated Green function solver. This object-oriented *Fortran90* code uses a two-dimensional domain decomposition in the $y - z$ directions and dynamic load balancing based on the domains. Hockney's FFT (Fast Fourier Transform) algorithm is used to solve Poisson's equation with open boundary conditions. The code is sensitive to the memory bandwidth and MPI collective performance.

MAESTRO is a low Mach-number hydrodynamics code for simulating astrophysical flows.[19] Its integration scheme is embedded in an adaptive mesh refinement algorithm based on a hierarchical system of rectangular nonoverlapping grid patches at multiple levels with different resolution; it uses a multigrid solver. Parallelization is via a tri-dimensional domain decomposition using a coarse-grained distribution strategy to balance the load and minimize communication costs. The communication topology tends to stress simple topology interconnects. The code has a very low computational intensity; it stresses memory latency, and the implicit solver stresses global communications; the message sizes range from short to relatively moderate.

MILC (MIMD Lattice Computation) is a QCD (Quantum Chromo Dynamics) code used to study "strong" interactions binding quarks in protons and neutrons and holding them together in the nucleus.[20] The algorithm discretizes the space and evaluates field variables on sites and the links of a

[16] The trajectory of charged particles in a magnetic field is a helix that winds around the field line; it can be decomposed into a relatively slow motion of the guiding center along the field line and a fast circular motion called cyclotronic motion. Gyrokinetics describes the evolution of the particles without taking into account the circular motion.

[17] See http://www.scidacreview.org/0601/html/news4.html.

[18] PIC is a technique to solve a certain class of partial differential equations; individual particles (or fluid elements) in a Lagrangian frame are tracked in continuous phase space, whereas moments of the distribution such as densities and currents are computed simultaneously on Eulerian (stationary) mesh points.

[19] See http://www.astro.sunysb.edu/mzingale/Maestro/.

[20] See http://physics.indiana.edu/~sg/milc.html.

regular hypercube lattice in four-dimensional space-time. The integration of an equation of motion for hundreds or thousands of time steps requires inverting a large, sparse matrix. The CG (Conjugate Gradient) method is used to solve a sparse, nearly-singular matrix problem. Many CG iterations steps are required for convergence; the inversion translates into tri-dimensional complex matrix–vector multiplications. Each multiplication requires a dot product of three pairs of tri-dimensional complex vectors; a dot product consists of five multiply–add operations and one multiply. The MIMD computational model is based on a four-dimensional domain decomposition; each task exchanges data with its eight nearest neighbors and is involved in the *all-reduce* calls with very small payloads as part of the CG algorithm; the algorithm requires *gather* operations from widely separated locations in memory. The code is highly memory- and computational-intensive and is heavily dependent on prefetching.

PARATEC (PARAllel Total Energy Code) is a quantum mechanics code; it performs *ab initio* total-energy calculations using pseudopotentials, a plane wave basis set and an all-band (unconstrained) conjugate gradient (CG) approach. Parallel three-dimensional FFTs transform the wave functions between real and Fourier space. The FFT dominates the runtime; the code uses MPI and is communication-intensive. The code uses mostly point-to-point short messages. The code parallelizes over grid points, thereby achieving a fine-grain level of parallelism. The BLAS3 and one-dimensional FFT use optimized libraries, e.g., Intel's MKL or AMD's ACML, and this results in high cache reuse and a high percentage of per-processor peak performance.

The authors of [256] use the HPCC (High Performance Computing Challenge) benchmark to compare the performance of EC2 with the performance of three large systems at NERSC. HPCC[21] is a suite of seven synthetic benchmarks: three targeted synthetic benchmarks that quantify basic system parameters that characterize individually the computation and communication performance; four complex synthetic benchmarks that combine computation and communication and can be considered simple proxy applications. These benchmarks are:

- DGEMM[22]—the benchmark measures the floating point performance of a processor/core; the memory bandwidth does little to affect the results as the code is cache friendly. Thus, the results of the benchmark are close to the theoretical peak performance of the processor.
- STREAM[23]—the benchmark measures the memory bandwidth.
- The network latency benchmark.
- The network bandwidth benchmark.
- HPL[24]—a software package that solves a (random) dense linear system in double precision arithmetic on distributed-memory computers; it is a portable and freely available implementation of the High Performance Computing Linpack Benchmark.
- FFTE—measures the floating-point rate of execution of double precision complex one-dimensional DFT (Discrete Fourier Transform)

[21] For more information, see http://www.novellshareware.com/info/hpc-challenge.html.
[22] For more details, see https://computecanada.org/?pageId=138.
[23] For more details, see http://www.streambench.org/.
[24] For more details, see http://netlib.org/benchmark/hpl/.

Table 11.1 The results of the measurements reported in [256].

System	DGEMM Gflops	STREAM GB/s	Latency µs	Bndw GB/S	HPL Tflops	FFTE Gflops	PTRANS GB/s	RandAcc GUP/s
Carver	10.2	4.4	2.1	3.4	0.56	21.99	9.35	0.044
Frankl	8.4	2.3	7.8	1.6	0.47	14.24	2.63	0.061
Lawren	9.6	0.7	4.1	1.2	0.46	9.12	1.34	0.013
EC2	4.6	1.7	145	0.06	0.07	1.09	0.29	0.004

- PTRANS—parallel matrix transpose; it exercises the communications where pairs of processors communicate with each other simultaneously. It is a useful test of the total communications capacity of the network.
- RandomAccess—measures the rate of integer random updates of memory (GUPS).

The systems used for the comparison with cloud computing are:

Carver—a 400-node IBM iDataPlex cluster with quad-core Intel Nehalem processors running at 2.67 GHz and with 24 GB of RAM (3 GB/core). Each node has two sockets; a single Quad Data Rate (QDR) IB link connects each node to a network that is locally a fat-tree with a global two-dimensional mesh. The codes were compiled with the Portland Group suite version 10.0 of and Open MPI version 1.4.1.

Franklin—a 9 660-node Cray XT4; each node has a single quad-core 2.3 GHz AMD Opteron "Budapest" processor with 8 GB of RAM (2 GB/core). Each processor is connected through a 6.4 GB/s bidirectional HyperTransport interface to the interconnect via a Cray SeaStar-2 ASIC. The SeaStar routing chips are interconnected in a tri-dimensional torus topology, where each node has a direct link to its six nearest neighbors. Codes were compiled with the Pathscale or the Portland Group version 9.0.4.

Lawrencium—a 198-node (1 584 core) Linux cluster; a compute node is a Dell Poweredge 1950 server with two Intel Xeon quad-core 64 bit and 2.66 GHz Harpertown processors with 16 GB of RAM (2 GB/core). A compute node is connected to a Dual Data Rate InfiniBand network configured as a fat tree with a 3 : 1 blocking factor. Codes were compiled using Intel 10.0.018 and Open MPI 1.3.3.

The virtual cluster at Amazon had four EC2 CUs (Compute Units), two virtual cores with two CUs each, and 7.5 GB of memory (an `m1.large` instance in Amazon parlance); a Compute Unit is approximately equivalent to a 1.0–1.2 GHz 2007 Opteron or 2007 Xeon processor. The nodes are connected via a gigabit Ethernet. The binaries were compiled on Lawrencium. The results reported in [256] are summarized in Table 11.1.

The results in Table 11.1 give us some idea about the characteristics of scientific applications likely to run efficiently on the cloud. Communication intensive applications will be affected by the increased latency (more than 70 times larger then *Carver*) and lower bandwidth (more than 70 times smaller than *Carver*).

11.11 Cloud computing and biology research

Biology is one of the scientific fields that needs vast amounts of computing power and was one of the first to take advantage of cloud computing. Molecular dynamics computations are CPU-intensive, while protein alignment is data-intensive.

An experiment carried out by a group from Microsoft Research illustrates the importance of cloud computing for biology research [316]. The authors carried out an "all-by-all" comparison to identify the interrelationships of the 10 million protein sequences (4.2 GB size) in NCBI's nonredundant protein database using AzureBLAST, a version of the BLAST[25] program running on the Azure platform [316].

Azure offers VM with four levels of computing power depending on the number of cores: small (1 core), medium (2 cores), large (8 cores), and extra large (> 8 cores). The experiment used 8 core CPUs with 14 GB RAM and a 2-TB local disk. It was estimated that the computation would take six to seven CPU-years; thus, the experiment was allocated 3 700 weighted instances or 475 extra-large VMs from three data centers; each data center hosted three AzureBLAST deployments, each with 62 extra large instances. The 10 million sequences were divided into multiple segments; each segment was submitted for execution by one AzureBLAST deployment. With this vast amount of resources allocated, it took 14 days to complete the computations which produced 260 GB of compressed data spread across over 400 000 output files.

A post-experiment analysis led to a few conclusions useful for many scientific applications running on Azure. When a task runs for more than two hours, a message automatically reappears in the queue requesting the task to be scheduled, leading to repeated computations. The simple solution to this problem is to check if the result of a task has been generated before launching its execution. Many applications, including BLAST, allow for the setting of some parameters, but the computational effort for finding optimal parameters is prohibitive. To meet budget limitations each user is expected to decide on an optimal balance between cost and the number of instances.

A number of inefficiencies were observed: Many VMs were idle for extended periods of time. When a task finished execution, all worker instances waited for the next task. When all jobs use the same set of instances, resources are either under- or over-utilized. Load imbalance is another source of inefficiency; some of the tasks required by a job take considerably longer than others and delay the completion time of the job.

The analysis of the logs shows unrecoverable instance failures; some 50% of active instances lost connection to the storage service but were automatically recovered by the fabric controller. System updates caused several ensembles of instances to fail.

Another observation is that a computational science experiment requires the execution of several binaries, thus the creation of workflows, a challenging task for many domain scientists. To address this challenge, the authors of [303] developed a general platform for executing legacy Windows applications on the cloud. In the Cirrus system, a job has a description consisting of a prologue, a set of commands, and a set of parameters. The prologue sets up the running environment; the commands are sequences

[25] The Basic Local Alignment Search Tool (BLAST) finds regions of local similarity between sequences; it compares nucleotide or protein sequences to sequence databases and calculates the statistical significance of matches that can be used to infer functional and evolutionary relationships between sequences and to help identify members of gene families. More information is available at http://blast.ncbi.nlm.nih.gov/Blast.cgi.

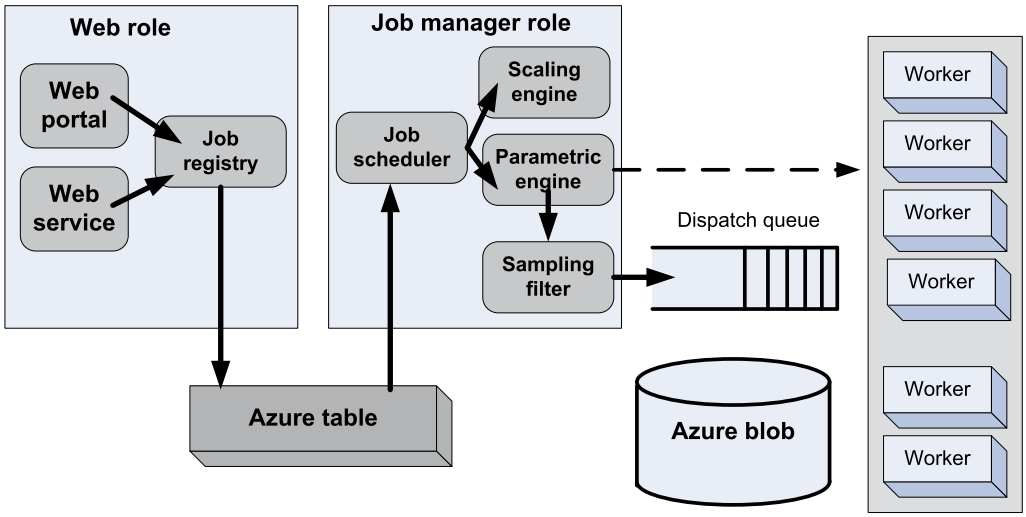

FIGURE 11.13

Cirrus—a general platform for executing legacy Windows applications on the cloud.

of shell scripts including Azure-storage-related commands to transfer data between Azure blob storage and the instance.

After the Windows Live ID service authenticates the user, it can submit and track a job through the portal provided by the web role; see Fig. 11.13; the job is added to a table called *job registry*. The execution of each job is controlled by a *job manager instance* that first scales the size of the worker based on the job configuration, and then, the parametric engine starts exploring the parameter space; if this is a test-run, the parameter sweeping result is sent to the sampling filter.

Each task is associated with a record in the task table, and this state record is updated periodically by the worker instance running the task; the progress of the task is monitored by the manager. The dispatch queue feeds into a set of worker instances. A worker periodically updates the task state in the task table and listens for any control signals from the manager.

A loosely coupled workload for an ensemble-based simulation on the Azure cloud is reported in [317]. A *role* in Azure is an encapsulation of an application; as noted earlier, there are two kinds of roles: (i) the web roles for web applications and front-end code; and (ii) the worker roles for background processing. Scientific applications, such as AzureBLAST, use worker roles for the compute tasks, and they implement their API that provides a run method and an entry point for the application and the state or configuration change notifications. The applications use the Blob Storage (ABS) for large raw data sets, the Table Storage (ATS) for semi-structured data, and the Queue Storage (AQS) for message queues; these services provide strong consistency guarantees, but the complexity is moved to the application space.

Fig. 11.14 illustrates the use of a software system called BigJob to decouple resource allocation from resource binding for the execution of loosely coupled workloads on an Azure platform [317]; this

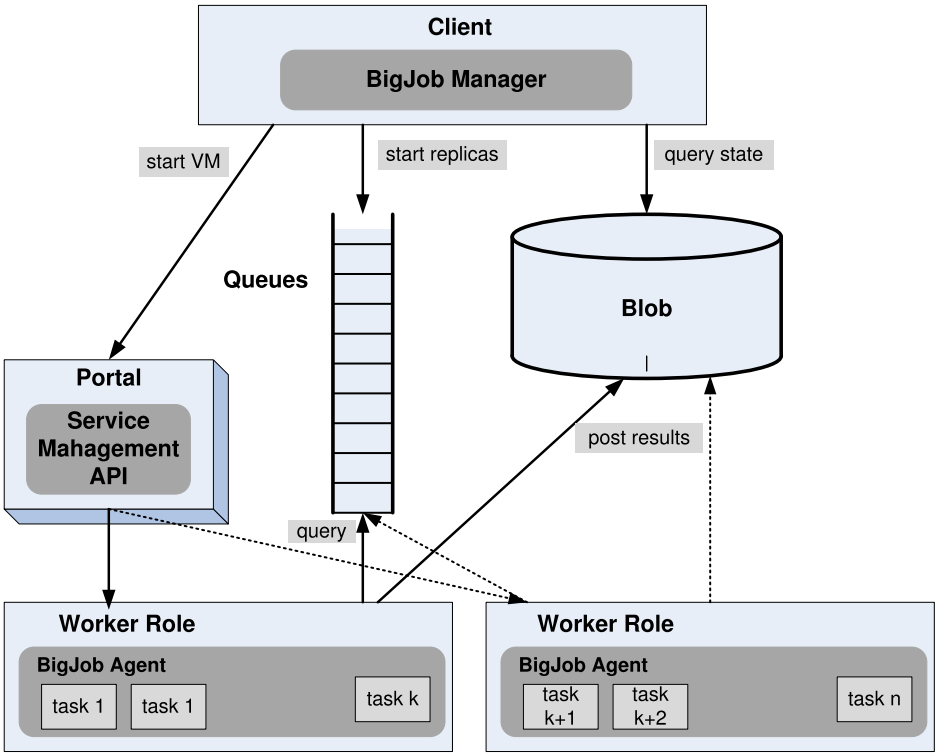

FIGURE 11.14

The execution of loosely coupled workloads using the Azure platform.

software eliminates the need for the application to manage individual VMs. The results of measurements show a noticeable overhead for starting VMs and for launching the execution of an application task on a remote resource; increasing the computing power of the VM decreases the completion time for long-running tasks.

11.12 Social computing, digital content, and cloud computing

Social networks play an increasingly important role in people's lives; they have expanded in terms of the size of the population involved and function performed. A promising solution for analyzing large-scale social networks data is to distribute the computation workload over a large number of cloud servers. Traditionally, evaluating the importance of a node or a relationship in a network is done using sampling and surveying, but in a very large network, structural properties cannot be inferred by scaling up the results from small networks. The evaluation of social closeness is computationally intensive.

Social intelligence is another area where social and cloud computing intersect. Knowledge discovery and techniques based on pattern recognition demand high-performance computing and resources that can only be provided by computing clouds. Case-based reasoning (CBR), the process of solving new problems based on the solutions of similar past problems, is used by context-aware recommendation systems; it requires similarity-based retrieval. As the case base accumulates, such applications must handle massive amounts of history data, and this can be done by developing new reasoning platforms running on the cloud. CBR is preferable to rule-based recommendation systems for large-scale social intelligence applications. Indeed, the rules can be difficult to generalize or apply to some domains; all triggering conditions must be strictly satisfied, scalability is a challenge as data accumulate, and the systems are hard to maintain because new rules have to be added as the amount of data increases.

The *BetterLife 2.0* [246] a CBR-based system, consists of a cloud layer, a case-based reasoning engine, and an API. The cloud layer uses Hadoop clusters to store application data represented by cases, as well as social network information, such as relationship topology and pairwise social-closeness information. CBR engine calculates similarity measures between cases to retrieve the most similar ones and also stores new cases back to the cloud layer. The API connects to a master node that is responsible for handling user queries, distributes the queries to server machines, and receives results.

A case consists of a problem description, solution, and optional annotations about the path to derive the solution. CBR uses MapReduce; all the cases are grouped by their *userId*, and then a *breath first search* (BFS) algorithm is applied to the graph where each node corresponds to one user. MapReduce is used to calculate the closeness according to pairwise relationship weight. A reasoning cycle has four steps: (a) retrieve the most relevant or similar cases from memory to solve the case; (b) reuse—map the solution from the prior case to the new problem; (c) revise—test the new solution in the real world or in a simulation and, if necessary, revise; and (d) retain—if the solution was adapted to the target problem, store the result as a new case.

In the past, social networks have been constructed for a specific application domain, e.g., MyExperiment and nanoHub for biology and nanoscience, respectively. These networks enable researchers to share data and provide a virtual environment supporting remote execution of workflows. Another form of social computing is the *volunteer computing* when a large population of users donate resources, such as CPU cycles and storage space for a specific project. The Mersenne Prime Search, initiated in 1996, followed in the late 1990s by the SETI@Home, the Folding@home, and the Storage@Home, a project to back up and share huge data sets from scientific research, are well-known examples of volunteer computing. Information about these projects is available online at: www.myExperiment.org, www.nanoHub.org, www.mersenne.org, setiathome.berkeley.edu, and at folding.stanford.edu.

Volunteer computing cannot be used for applications where users require some level of accountability. The PlanetLab project is a credit-based system in which users earn credits by contributing resources and then spend these credits when using other resources. Berkeley Open Infrastructure for Network Computing (BOINC) is the middleware for a distributed infrastructure suitable for multiple applications.

An architecture designed as a Facebook application for a social cloud is presented in [98]. Methods to get a range of data including friends, events, groups, application users, profile information, and photos are available through a Facebook API. The Facebook Markup Language is a subset of HTML with proprietary extensions, and Facebook JavaScript is a version of JavaScript. The prototype uses web services to create a distributed and decentralized infrastructure. There are numerous examples of cloud platforms for social networks. There are scalable cloud applications hosted by commercial clouds.

The new technologies supported by cloud computing favor the creation of digital content. *Data mashups* or *composite services* combine data extracted by different sources; *event-driven mashups*, also called Svc, interact through events rather than the request-response traditional method. A recent paper [454] argues that "the *mashup* and the cloud computing worlds are strictly related because very often the services combined to create new Mashups follow the SaaS model and more, in general, rely on cloud systems." The paper also argues that the Mashup platforms rely on cloud computing systems, for example, the IBM Mashup Center and the JackBe Enterprise Mashup server.

There are numerous examples of monitoring, notification, presence, location, and map services based on the Svc approach, including: Monitor Mail, Monitor RSSFeed, Send SMS, Make Phone Call, GTalk, Fireeagle, and Google Maps. As an example, consider a service to send a phone call when a specific email is received; the Mail Monitor Svc uses input parameters such as: User ID, Sender Address Filter, Email Subject Filter, to identify an email and generate an event that triggers the *Make TTS Call* action of a *Text To Speech Call* Svc linked to it.

The system in [454] supports creation, deployment, activation, execution, and management of Event Driven Mashups; it has a user interface, a graphics tool called Service Creation Environment that supports easily the creation of new Mashups and a platform called *Mashup Container* that manages Mashup deployment and execution. The system consists of two subsystems, the *service execution platform* for Mashups execution and the *deployer* module that manages the installation of Mashups and Svcs. A new Mashup is created using the graphical development tool, and it is saved as an XML file; it can then be deployed into a *Mashup Container* following the Platform as a Service (PaaS) approach. The *Mashup Container* supports a primitive SLA allowing the delivery of different levels of service.

The prototype uses the JAVA Message Service (JMS), which supports an asynchronous communication; each component sends/receives messages, and the sender does not block waiting for the recipient to respond. The system's fault tolerance was tested on a system based on the VMware vSphere. In this environment, the fault tolerance is provided transparently by the VMM, and neither the VMs nor the applications are aware of the fault tolerance mechanism; two VMs, a Primary and a Secondary one, run on distinct hosts and execute the same set of instructions such that, when the Primary fails, the Secondary continues the execution seamlessly.

11.13 Software fault isolation

Software fault isolation (SFI) offers a technical solution for sandboxing binary code of questionable provenance that can affect security in cloud computing. Insecure and tampered VM images comprise one of the security threats; binary codes of questionable provenance for native plugins to a web browser can pose a security threat because web browsers are used to access cloud services.

A recent paper [444] discusses the application of the sandboxing technology for two modern CPU architectures, ARM and x86-64. ARM is a load/store architecture with 32-bit instruction, 16 general-purpose registers. It tends to avoid multicycle instructions, and it shares many of the RISC architecture features, but: (a) it supports a "thumb" mode with 16-bit instruction extensions; (b) has complex addressing modes and a complex barrel shifter; and (c) condition codes can be used to predicate most instructions. In the x86-64 architecture, general-purpose registers are extended to 64-bits, with an *r* replacing the *e* to identify the 64 versus 32-bit registers, e.g., *rax* instead of *eax*; there are eight new

Table 11.2 The features of the SFI for the Native Client on the x86-32, x86-64 and ARM; ILP stands for Instruction Level Parallelism.

Feature/Architecture	x86-32	x86-64	ARM
Addressable memory	1 GB	4 GB	1 GB
Virtual base address	any	44GB	0
Data model	ILP32	ILP32	ILP 32
Reserved registers	0 of 8	1 of 16	0 of 16
Data address mask	None	Implicit in result width	Explicit instruction
Control address mask	Explicit instruction	Explicit instruction	Explicit instruction
Bundle size (bytes)	32	32	16
Data in text segment	forbidden	forbidden	allowed
Safe address registers	all	rsp, rbp	sp
Out-of-sandbox store	trap	wraps mod 4 GB	No effect
Out-of-sandbox jump	trap	wraps mod 4 GB	wraps mod 1 GB

general-purpose registers named $r8$–$r15$. To allow legacy instructions to use these additional registers, x86-64 defines a set of new prefix bytes to use for register selection.

This SFI implementation is based on the previous work of the same authors on Google Native Client (NC). This implementation assumes an execution model where a trusted runtime system shares a process with the untrusted multithreaded plugin. The rules for binary code generation of an untrusted plugin are:

1. The code section is read-only, and it is statically linked.
2. The code is divided into 32 byte *bundles,* and no instruction or pseudo-instruction crosses the bundle boundary.
3. The disassembly starting at the bundle boundary reaches all valid instructions.
4. All indirect flow control instructions are replaced by pseudo-instructions that ensure address alignment to bundle boundaries.

The features of the SFI for the Native Client on the *x86*-32, *x86*-64, and *ARM* are summarized in Table 11.2 [444]. The control flow and store sandboxing for the ARM SFI incur less then 5% average overhead, and the ones for x86-64 SFI incur less than 7% average overhead.

11.14 Further readings

MapReduce is discussed in [129], and [487] presents the GrepTheWeb application. Cloud applications in biology are analyzed in [316] and [317], and social applications of cloud computing are presented in [98], [246], and [454]. Benchmarking of cloud services is analyzed in [106], [256]. The use of structured data is covered in [321]. An extensive list of publications related to Divisible Load Theory is at http://www.ece.sunysb.edu/~tom/dlt.html.

There are several cluster programming models. Data flow models of MapReduce and Dryad [255] support a large collection of operations and share data through stable data. High-level programming languages, such as DryadLINQ [530] and FlumeJava [90], enable users to manipulate parallel collec-

tions of datasets using operators such as *map* and *join*. There are several systems providing high-level interfaces for specific applications such as *HaLoop* [77]. There are also caching systems including Spark [533] and *Tachyon* [302], discussed in Chapter 4, and *Nectar* [214].

Hive [475] was the first SQL over Hadoop to use another framework such as MapReduce or Tez to process SQL-like queries. Shark uses another framework, Spark [521] as its runtime. Impala [176] from Cloudera, LinkedIn Tajo (http://tajo.incubator.apache.org/), MapR *Drill* (http://www.mapr.com/resources/community-resources/apache-drill) and Facebook *Presto* (http://prestodb.io/) resemble parallel databases and use long-running custom-built processes to execute SQL queries in a distributed fashion. Hadapt [5] uses a relational database (PostgreSQL) to execute query fragments. Microsoft PolyBase [144] and Pivotal [97] use database query optimization and planning to schedule query fragments and read HDFS data into database workers for processing.

A discussion of cost-effective cloud-based high-performance computing and a comparison with supercomputers [478] is reported in [87]. [517] discusses scientific computing on clouds. Service level checking is analyzed in [99].

11.15 Exercises and problems

Problem 1. Download and install the *Zookeeper* from the site http://zookeeper.apache.org/. Use the API to create the basic workflow patterns shown in Fig. 11.3.

Problem 2. Use *AWS Simple Workflow Service* to create the basic workflow patterns in Fig. 11.3.

Problem 3. Use *AWS CloudFormation* service to create the basic workflow patterns in Fig. 11.3.

Problem 4. Define a set of keywords ordered based on their relevance to the topic of cloud security; then search the web using these keywords to locate 10–20 papers and store the papers in an *S3* bucket. Create a MapReduce application modeled after the one discussed in Section 11.6 to rank the papers based on the incidence of the relevant keywords. Compare your ranking with the rankings of the search engine you used to identify the papers.

Problem 5. Use the *AWS MapReduce* service to rank the papers in Problem 4.

Problem 6. The paper [84] describes the *elasticLM*, a commercial product that provides license and billing Web-based services. Analyze the merits and the shortcomings of the system.

Problem 7. Search the web for reports of cloud system failures and discuss the causes of each incident.

Problem 8. Identify a set of requirements you would like to be included in a SLA.

Problem 9. Consider the workflow for your favorite cloud application. Use XML to describe this workflow, including the instances and the storage required for each task. Translate this description into an file that can be used for the *Elastic Beanstalk* AWS.

Problem 10. In Section 11.10, we analyze cloud-computing benchmarks and compare them with the results of the same benchmarks performed on a supercomputer. This is not unexpected; discuss the reasons why we should expect the poor performance of fine-grained parallel computations on a cloud.

Problem 11. An IT company decides to provide free access to a public cloud dedicated to higher education. Which one of the three cloud computing delivery models, SaaS, PaaS, or IaaS, should it embrace and why? What applications would be most beneficial for the students? Will this solution have an impact on distance learning? Why?