50

Developing Elastic Software for the Cloud

Shigeru Imai, Pratik Patel, and Carlos A. Varela

Rensselaer Polytechnic Institute, USA

50.1 Introduction

Developing standalone applications running on a single computer is very different from developing scalable applications running on the cloud, such as data analytics applications that process terabytes of data, Web applications that receive thousands of requests per second, or distributed computing applications where components run simultaneously across many computers. Cloud computing service providers help facilitate the development of these complex applications through their *cloud programming frameworks*. A cloud programming framework is a software platform to develop applications in the cloud that takes care of *nonfunctional concerns,* such as scalability, elasticity, fault tolerance, and load balancing. Using cloud programming frameworks, application developers can focus on the functional aspects of their applications and benefit from the power of cloud computing.

In this chapter, we will show how to use some of the existing cloud programming frameworks in three application domains: data analytics, Web applications, and distributed computing. More specifically, we will explain how to use *MapReduce* (Dean and Ghemawat, 2008) for data analytics, *Google App Engine* (Google, 2014) for Web applications, and *SALSA* (Varela and Agha, 2001) for distributed computing. The rest of the chapter is structured as follows. In section 50.2, we describe nonfunctional concerns supported at different levels of cloud services and go through existing cloud programming frameworks. In section 50.3, we explain MapReduce, Google App Engine, and Simple Actor Language System and Architecture (SALSA). In section 50.4, we illustrate how to use these three programming frameworks by showing example applications. Finally, we conclude the chapter in section 50.5.

Encyclopedia of Cloud Computing, First Edition. Edited by San Murugesan and Irena Bojanova.

50.2 Programming for the Cloud

50.2.1 Nonfunctional Concerns

Figure 50.1 illustrates how cloud programming frameworks hide nonfunctional concerns from application developers by providing programming languages or application programming interfaces (APIs) to manage the application's execution on different cloud service models, such as IaaS and PaaS.

Nonfunctional concerns are not directly related to the main functionality of a system but guarantee important properties such as security or reliability. In the context of cloud computing services, important nonfunctional concerns include the following:

- *Scalability*: the ability to scale up and out computing resources to process more workload or to process it faster as demanded by cloud users.
- *Elasticity*: the ability of an application to adapt in order to scale up *and down* as service demand grows or shrinks.
- *Fault tolerance*: the ability to keep the system working properly even in the event of a computing resource failure.
- *Load balancing*: the ability to balance the workload between heterogeneous networked computing resources.

If not using cloud computing, application developers would have to acquire physical infrastructure (machines, networks, etc.) to support their needs. Using IaaS, developers can create virtual machines (VMs) without up-front costs for hardware; however, they have to install and configure the VM management

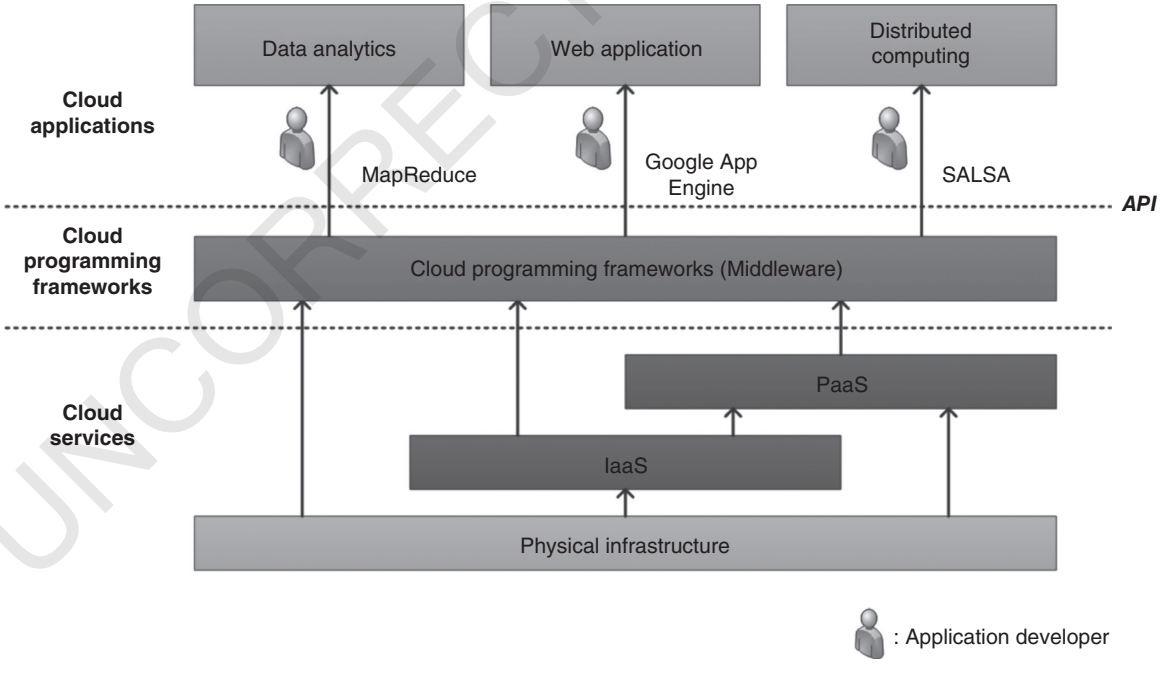

Figure 50.1 *Cloud programming frameworks take advantage of layered services by exposing an Application Programming Interface (API) to developers*

software, networking, operating system, and any additional libraries that their application needs. Finally, different PaaS providers offer from transparent scalability and elasticity to transparent fault tolerance and load balancing services. Typically, these nonfunctional concerns are offered, constraining developers to specific programming models and patterns, such as MapReduce, Web applications, and distributed actor computations.

50.2.2 Overview of Cloud Programming Frameworks

Data analytics. MapReduce is a popular data-processing framework created by Google following a data-parallel programming model based on the *map* and *reduce* higher-order abstractions from functional programming. Hadoop (see http://www.apache.org/, accessed January 5, 2016) is a popular open-source implementation of MapReduce and has a large user and contributor base. There are a number of projects derived from Hadoop. One such project is *Pig* (http://www.apache.org/), which was first developed by Yahoo! Pig offers a high-level language called *Pig Latin* to express data analytics programs. Programs authored in Pig Latin are translated into Java-based Hadoop code and thus Pig users benefit from Hadoop's scalability and fault-tolerance properties as well as Pig Latin's simplicity. Another Hadoop related project is *Hive* (http://www.apache.org/), initially developed by Facebook. Hive is a data warehouse platform built on top of Hadoop. *Mahout* (http://www.apache.org/) is a set of scalable machine learning libraries that also works over Hadoop. *Spark* (http://www.apache.org/) is a cluster-computing framework that focuses on efficient use of data objects to speed up iterative algorithms such as machine learning or graph computation. Whereas Hadoop reads (writes) data from (to) storage repeatedly, Spark caches created data objects in memory so that it can perform up to 100 times faster than Hadoop for a particular class of applications. *Shark* (Xin *et al.,* 2013) is a Structured Query Language (SQL) query engine that has compatibility with Hive and runs on Spark. These frameworks do not support elastic behavior of applications, but have good scalability, load balancing, and fault tolerance.

Web applications. As Web applications have unpredictable and varying demands, they constitute a very good match for cloud computing. There are several PaaS providers that offer a framework for developing and hosting Web applications, such as Google App Engine, Microsoft Azure (Microsoft, 2014), and Heroku (2014). Google App Engine supports Python, Java, PHP, and Go. The runtime environment for Google App Engine is restrictive (e.g., no socket use and no file system access), but you get good scalability with a few lines of code. In contrast, Microsoft Azure offers a more flexible runtime environment; however, it requires you to write more code. Microsoft Azure supports .NET, Java, PHP, Node.js, and Python. Heroku originally supported only Ruby, but now supports Java, Node.js, Scala, Clojure, Python, and PHP. In terms of coding flexibility, Heroku is also as restrictive as Google App Engine; however, it has libraries to support data management, mobile users, analytics, and others.

Distributed computing. Distributed computing systems typically have components that communicate with each other via message passing to solve large or complex problems cooperatively. Erlang (Armstrong *et al.,* 1993) and SALSA are concurrent and distributed programming languages based on the *actor* model (Agha, 1986), in which each actor runs concurrently and exchanges messages asynchronously while not sharing any state with any other actor. Actor systems can therefore be reconfigured dynamically, while transparently preserving message passing semantics, which is very helpful for scalability, elasticity, and load balancing. Erlang is a functional language that supports fault tolerance and hot swapping. SALSA's compiler generates Java code allowing programmers to use the entire Java library collection. It also supports transparent actor migration across the Internet based on a universal naming system. Several research efforts have been made to support nonfunctional concerns for SALSA programs. The *Internet Operating System* (IOS) (El Maghraoui *et al.,* 2006) is a distributed middleware framework that provides support for dynamic reconfiguration of large-scale distributed applications through opportunistic load balancing.

Table 50.1 *Summary of cloud programming frameworks*

Category	Name	Nonfunctional concerns	Description
Data analytics	MapReduce, Hadoop	Scalability, fault tolerance, load balancing	MapReduce exposes simple abstractions: *map* and *reduce*. Hadoop is an open-source implementation of MapReduce.
	Pig, Hive		Pig and Hive are high-level languages for Hadoop.
	Mahout, Spark, Shark		Mahout is a machine-learning library running on top of Hadoop. Spark is a data analytics framework especially for iterative and graph applications. Shark is a Hive-compatible SQL engine running on top of Spark.
Web applications	Google App Engine	Scalability, elasticity, fault tolerance, load balancing	A PaaS from Google. The runtime environment is restrictive, but provides a good scalability. Python, Java, PHP, and Go are supported.
	Microsoft Azure		A PaaS from Microsoft with automatic scaling, automatic patching, and security services.
	Heroku		A PaaS from Heroku with libraries for data management, mobile users, analytics, and others.
Distributed computing	SALSA, Erlang	Scalability, fault tolerance (for Erlang)	General-purpose programming languages based on the actor model.
	IOS, COS	Scalability, elasticity (COS), load balancing	Middleware framework for managing distributed SALSA programs.

In addition to the distributed load-balancing capability, the *Cloud Operating System* (COS) (Imai *et al.*, 2013) further supports elasticity, enabling adaptive virtual machine allocation and de-allocation on hybrid cloud environments, where private clouds connects to public clouds.

Table 50.1 summarizes the cloud programming frameworks mentioned above.

50.3 Cloud Programming Frameworks

In this section, we present existing cloud programming frameworks for data analytics, web applications, and distributed computing, namely, MapReduce, Google App Engine, and the SALSA programming language.

50.3.1 Data Parallelism with MapReduce

MapReduce created by Google is a data-parallel programming model based on the *map* and *reduce* higher order abstractions from functional programming. Its implementation is also called MapReduce; this is designed to schedule automatically the parallel processing of large data sets distributed across many computers. MapReduce was designed to have good scalability to achieve high throughput, and fault tolerance to deal with unavoidable hardware failures. These nonfunctional concerns are transparent to application developers; MapReduce provides a simple application programming interface requiring to define only the `map` and `reduce` functions that have the format shown in Listing 50.1.

```
map(k1, v1) → list (k2, v2)
reduce(k2, list(v2)) → list (v3)
```

Listing 50.1 *MapReduce abstract application programming interface*

```
map(string key, string value)
   //key: the position of value in the input file
  //value: partial DNA sequence in the input file
  foreach character c in value do
      emit(c, 1);
  end

reduce(string key, string value)
   //key: DNA character (C, G, T, or A)
  //value: a list of counts
 sum = 0;
     foreach count in value do
        sum = sum + count;
     end
emit(sum);
```

Listing 50.2 *MapReduce pseudo code for DNA sequence analysis*

The computation in MapReduce basically consists of two phases: the "map" phase producing intermediate results, followed by the "reduce" phase processing the intermediate results. Both phases take a (`key, value`) pair as input and output a list. Keys and values can be arbitrary numbers, strings, or user-defined types. First, the `map` function gets called by MapReduce and transforms a `(k1, v1)` pair into an intermediate list of `(k2, v2)`. For instance, in text-mining applications, `v1` may be a partial text of an input file. Next, these intermediate lists are aggregated into pairs where each pair has all `v2` values associated with the same `k2` key. Finally, the `reduce` function is invoked with a `(k2, list(v2))` pair and it outputs a list of `v3` values.

Suppose you have a big input file consisting of DNA sequences, which are combinations of {C, G, T, A} characters, one way to write map and reduce functions to count each character occurrence in the input file can be given as shown in Listing 50.2.

The map function emits count 1 as an occurrence of a DNA character. The reduce function takes a list of counts and emits the sum of these counts for a particular DNA character. By calling these simple two functions repeatedly, MapReduce outputs occurrences of each DNA character. An example execution for the DNA sequence counting example is shown in Figure 50.2.

50.3.2 Service-Oriented Programming with Google App Engine

Google App Engine (App Engine hereafter) is a programming framework to develop scalable Web applications running on Google's infrastructure. Developed Web applications are hosted by Google and accessible via the user's own domain name. Google automatically allocates more computing resources when service demand grows and also balances the workload among the computing resources. App Engine is regarded as a PaaS

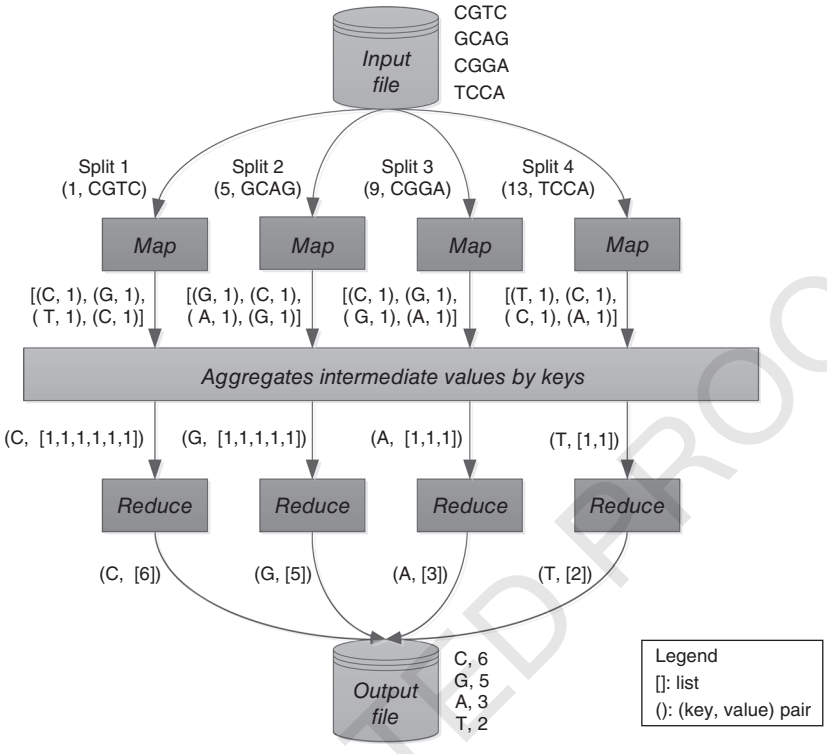

Figure 50.2 *Execution sequence of MapReduce*

implementation and charges costs based on the use of storage, CPU, and bandwidth. App Engine currently supports development in Python, Java, PHP, and Go. App Engine's core features and its programming framework are described in the following sections. We use only Python for brevity.

50.3.2.1 Core Features

The core features of App Engine (as of July 2014) are as follows:

- *Sandbox.* To prevent harmful operations to the underlying operating system, applications run in an isolated secure environment called *sandbox*. In the sandbox, an application is allowed to only access other computers on the Internet using Uniform Resource Locator (URL) Fetch and Mail services, and application code is able to only run in a response to a Web request. The application is not allowed to write to the file system directly, but instead uses `Datastore` or `Memcache` services.
- *Storing data.* App Engine `Datastore` is a data storing service based on Google's *BigTable* (Chang *et al.,* 2008), which is a distributed storage system that scales up to petabytes of data.
- *Account management.* Applications can be easily integrated with *Google Accounts* for user authentication. With Google Accounts, an application can detect if the current user has signed in. If not, it can redirect the user to a sign-in page.
- *App Engine services.* App Engine provides a variety of services via APIs to applications including fetching Web resources, sending e-mails, using cache and memory instead of secondary storage, and manipulating images.

50.3.2.2 Programming Framework for Python

We describe the App Engine's programming framework for Python as visualized in Figure 50.3. When the Web server receives a Hyper-Text Transfer Protocol (HTTP) request from the browser, the Web server passes the request to a framework (a library that helps Web application development) via Web Server Gateway Interface (WSGI). The framework then invokes a handler in a user script (in this example, `main.py`) and the invoked handler processes the request and creates an HTTP response dynamically.

The Python runtime in App Engine uses WSGI as an interface to connect the Web server to the Web applications. WSGI is simple, but reduces programming effort and enables more efficient application development. Here, we give an example of a request handler using the `webapp2` framework that interacts with WSGI as shown in Listing 50.3.

The `application` object – an instance of `WSGIApplication` class of `webapp2` framework – handles the requests. When creating the `application` object, a request handler called `MainPage` is associated with the root URL path (/). The `webapp2` framework invokes the `get` function in the `MainPage` class when it receives an HTTP GET request to the URL /. In the `get` function, it creates an HTTP response with a `Content-Type` header and a body containing a "*Hello, World!*" message.

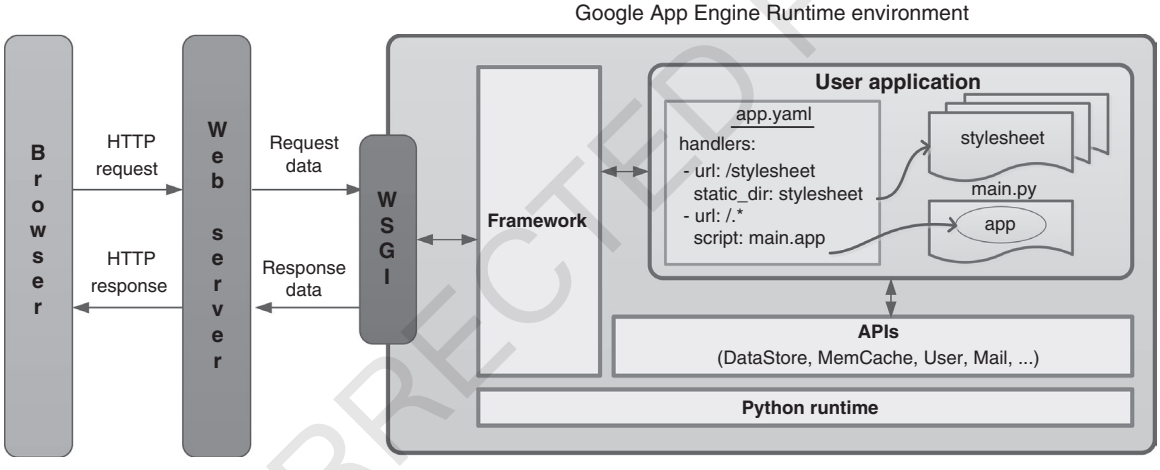

Figure 50.3 *Google App Engine programming framework for Python*

```
import webapp2

class MainPage( webapp2.RequestHandler ):
    def get( self ):
        self.response.headers[ 'Content-Type' ] = 'text/plain'
        self.response.write( 'Hello, World!' )

application = webapp2.WSGIApplication(
                                [('/', MainPage)],
                                        debug=True )
```

Listing 50.3 *Application Python script for the Helloworld example*

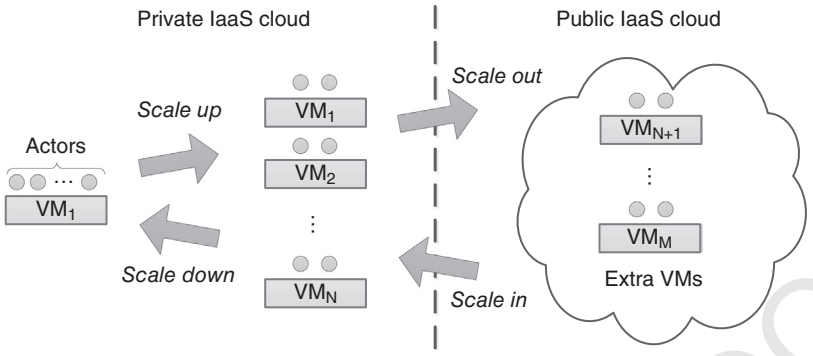

Figure 50.4 *Workload scalability over a hybrid cloud using actor migration. Source: Imai et al. (2013)*

50.3.3 Distributed Actor Systems with SALSA

Simple Actor Language System and Architecture is a concurrent programming language based on the actor model (Agha, 1986). Each actor runs concurrently and exchanges messages asynchronously while encapsulating its state. Since an actor does not share any memory with other actors, it can migrate to another computing host easily. As shown in Figure 50.4, under a hybrid IaaS cloud environment, actors can migrate between the private and public clouds seamlessly with runtime software installed on virtual machines (VMs) on both ends. In this scenario, application developers need either to use COS middleware or support appropriate nonfunctional concerns by themselves as IaaS clouds only provide scalability by means of VM addition / removal. Nevertheless, hybrid clouds can be attractive for those who can access their private computing resources at no additional cost and who need high computing power only occasionally.

In the following subsections, we introduce the actor-oriented programming model followed by a distributed application written in SALSA.

50.3.3.1 Actor-Oriented Programming

Actors provide a flexible model of concurrency for open distributed systems. Each actor encapsulates a state and a thread of control that manipulates this state. In response to a message, an actor may perform one of the following actions (see Figure 50.5):

- alter its current state, possibly changing its future behavior;
- send messages to known actors asynchronously;
- create new actors with a specified behavior;
- migrate to another computing host.

Analogous to a class in Java, SALSA programmers can write a *behavior* which includes encapsulated state and message handlers for actor instances:

- New actors are created in SALSA by instantiating particular behaviors with the `new` keyword. Creating an actor returns its reference. For instance:
  ```
  ExampleActor exActor = new ExampleActor();
  ```
- The message sending operation (`<-`) is used to send messages to actors; messages contain a name that refers to the message handler for the message and a possibly empty list of arguments. For instance:
  ```
  exActor<-m(1, 2);
  ```
- Actors, once created, process incoming messages, one at a time.

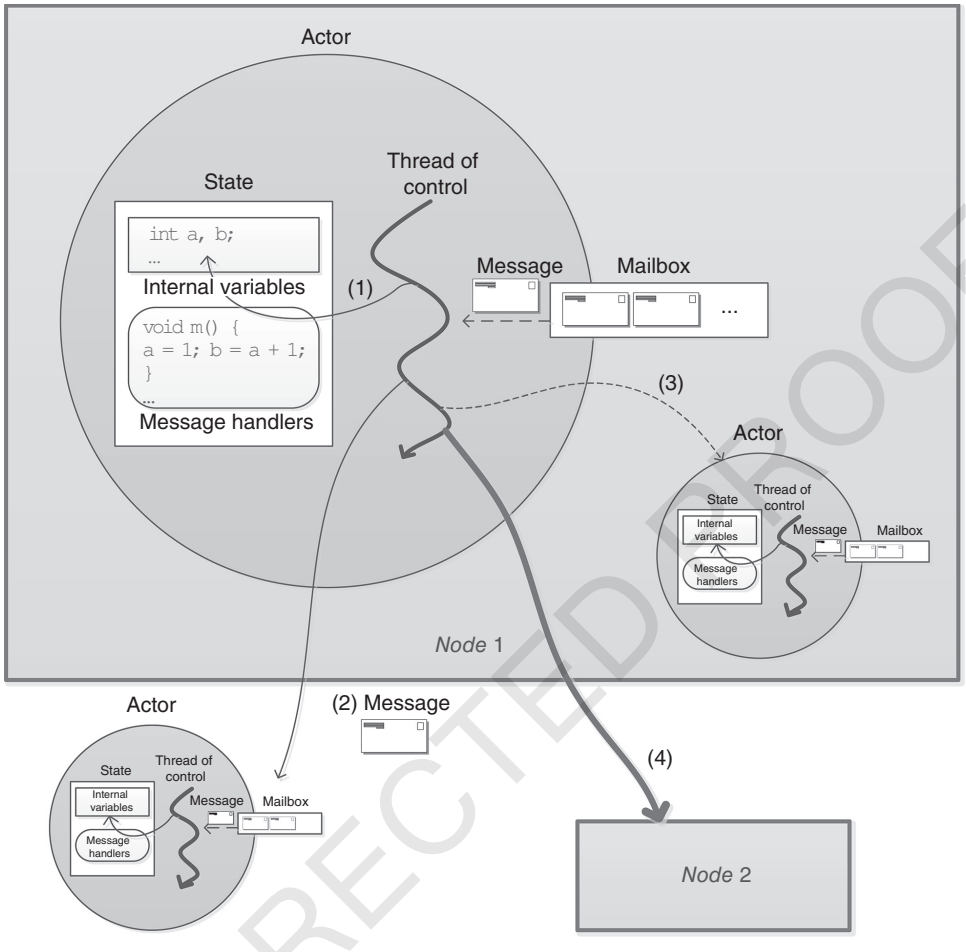

Figure 50.5 *Actors programming model*

50.3.3.2 Programming Distributed Applications

A simple distributed SALSA application that migrates an actor with name as specified by an universal actor name (UAN) from a location specified by a universal actor location (UAL) is shown in Listing 50.4. A UAN is an identifier that represents an actor during its lifetime in a location-dependent manner. An actor's UAN is mapped by a naming service into a UAL, which provides access to an actor in a specific location.

Listing 50.4 defines a behavior of the `Migrate` actor. First, the program starts from the `act` message handler and a `Migrate` actor named `migrateActor` is created with a UAN `"uan://wcl.cs.rpi.edu:3030/myName"` at a UAL `"rmsp://host1.cs.rpi.edu:4040/myLocator"`. Next, the actor receives a `print` message and prints out a string `"Migrate actor is here!"` in the standard output of `host1`. Right after printing the string, the actor migrates to a new location specified by a UAL `"rmsp://host2.cs.rpi.edu:4040/myLocator"`. Finally, after the migration, the actor prints out the same string at `host2`. Note that SALSA's transparent migration support enables execution of the same print message in two different hosts.

```
behavior Migrate {
     void print() {
         standardoutput<-println("Migrate actor is here!");
     }

     void act(string[] args) {
         UAN uan = new UAN("uan://wcl.cs.rpi.edu:3030/myName");
         UAL ual = new UAL(
             "rmsp://host1.cs.rpi.edu:4040/myLocator");

         Migrate migrateActor = new Migrate() at (uan, ual);
         migrateActor<-print()@
         migrateActor<-migrate(
             "rmsp://host2.cs.rpi.edu:4040/myLocator")@
         migrateActor<-print();
     }
}
```

Listing 50.4 *SALSA migration code example*

50.4 Sample Applications

In this section, we present sample applications for the three programming frameworks. First, we describe a data analytics application that computes the average temperatures of historical weather data using Hadoop. Next, we show a simple Web-based bulletin board application using Google App Engine. Thirdly, we describe a distributed face recognition application using SALSA.

50.4.1 Data Analytics

The application presented in this subsection processes a large amount of data by Hadoop to compute average temperatures for every month using temperature data collected from all over the world.

50.4.1.1 Global Surface Summary of Day Data (GSOD)

Global surface summary of day (GSOD) data is a collection of daily weather data produced by the National Climatic Data Center. The weather data has been collected from 1929 to the present by stations all over the world. The elements contained in the daily data include mean temperature, mean sea level pressure, mean visibility, and mean wind speed.

The datasets are provided in ASCII characters. Each row contains weather data for a station on a particular day (see, for example, Listing 50.5). Fields are explained in Table 50.2.

Since we compute average temperatures for every month, the fields we are interested in are `YEAR`, `MODA` and `TEMP`.

```
STN---   YEARMODA    TEMP     DEWP      SLP         STP     …
030050  19291001  45.3 4  40.0  4 1001.6 4  9999.9  0 …
```

Listing 50.5 *Example of GSOD data*

Table 50.2 *Field definitions of GSOD data*

Field	Position	Type	Description
STN	1–6	Integer	Station number for the location
YEAR	9–12	Integer	The year
MODA	13–16	Integer	The month and day
TEMP	19–24	Real	Mean temperature for the day in degrees
…	…	…	…

```
map(String key, String value)
    //key: tag, value: number
    pair = (value, 1);
    emit(key, pair);

reduce(String key, String value)
    //key: tag, value: a list of (number, count)pairs
    sum = 0; count = 0;
    foreach pair (n, c) in value do
        sum = sum + n;
        count = count + c;
    end
    average = sum / count;
    emit(average);
```

Listing 50.6 *MapReduce pseudo code for average number calculation*

50.4.1.2 MapReduce Functions for Averaging Numbers

Suppose we are given an input file that consists of (*tag*, *number*) pairs, where tag is an arbitrary string of a month, year, station, and so on, we can compute an average number for each tag by the `map` and `reduce` functions shown in Listing 50.6.

The `map` function emits an intermediate (*tag,* (*number, 1*)) pair for every input and then the `reduce` function computes the average number $\frac{1}{N}\sum_{i=1}^{N} number_i$ for each tag from an aggregated pair (*tag,* [(*number*$_1$, *1*), …, (*number*$_N$, *1*)]).

50.4.1.3 GSOD Data Analytics Application

Based on the functions' pseudo code presented in the previous section, we show a Hadoop application that computes average temperatures for every month using GSOD temperature data. The `map` function is defined as a `GsodMapper` class implementation that extends the `Mapper` class as shown in Listing 50.7.

```
public class GsodMapper extends
    Mapper<Object, Text, Text, DoubleIntPair> {
    private Text word = new Text();
    private DoubleIntPair pair = new DoubleIntPair();

    public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
        if (value.toString().startsWith( "STN" )) {
            return;
        }
         String[] data = value.split( "+" );
         String yearmon = data[2].substring(0, 6);
         word.set( yearmon );
         pair.set( Double.parseDouble( data[3] ), 1 );
         context.write( word, pair )
    }
}
```

Listing 50.7 *GsodMapper class definition*

The `Mapper` class is a generic class that takes four type variables – the input key type, the input value type, the output key type, and the output value type. In this example, the input key type is `Object`, the input key value is `Text`, the output key type is `Text`, and the output value type is `DoubleIntPair`. `DoubleIntPair` is used to store a pair of `double` and `int` values.

The `map` function repeatedly receives one line of GSOD weather input data in the `value` variable. If the `value` starts with an "`STN`", that means that it is a line for weather field names, therefore we skip it. Otherwise, split the `value` into a `data` array in which each element is separated by spaces. Since we want to compute the average temperature for each month, we are interested in the `YEARMODA` and `TEMP` fields, which are the third and fourth fields of the weather data. From these two fields, we can extract a year and month by `data[2].substring(0,6)` in the `YYYYMM` format and temperature by `data[3]`. By calling `context.write()` function, the `map` function outputs a key value pair in the format (*key*=YYYYMM, *value*=(temperature, 1)).

The `GsodReducer` class that implements the `reduce` function is defined in Listing 50.8. The specified input types are `Text` and `DoubleIntPair` and output types are `Text` and `DoubleWritable`. The `reduce` function iterates the `values`, which is a list of a temperature and count, and sums up the temperatures and counts to compute the average temperature.

50.4.2 Bulletin Board Web Application

In this subsection we present a simple bulletin board Web application developed on the Google App Engine framework (see Figure 50.6 for user interface). This application lets users post a message with their usernames to a public bulletin board. The posted messages are stored in the server and shared among the users accessing the application.

```
public class GsodReducer extends
    Reducer<Text, DoubleIntPair, Text, DoubleWritable> {
    private DoubleWritable average = new DoubleWritable();

    public void reduce(Text key,
                       Iterable<DoubleIntPair> values,
                       Context context)
        throws IOException, InterruptedException {
        double sum = 0;
        int count = 0;
        for (DoubleIntPair value: values)
            sum += value.getDouble();
            count += value.getInt();
        }
         average.set(sum / count);
         context.write(key, average)
    }
}
```

Listing 50.8 *GsodReducer class definition*

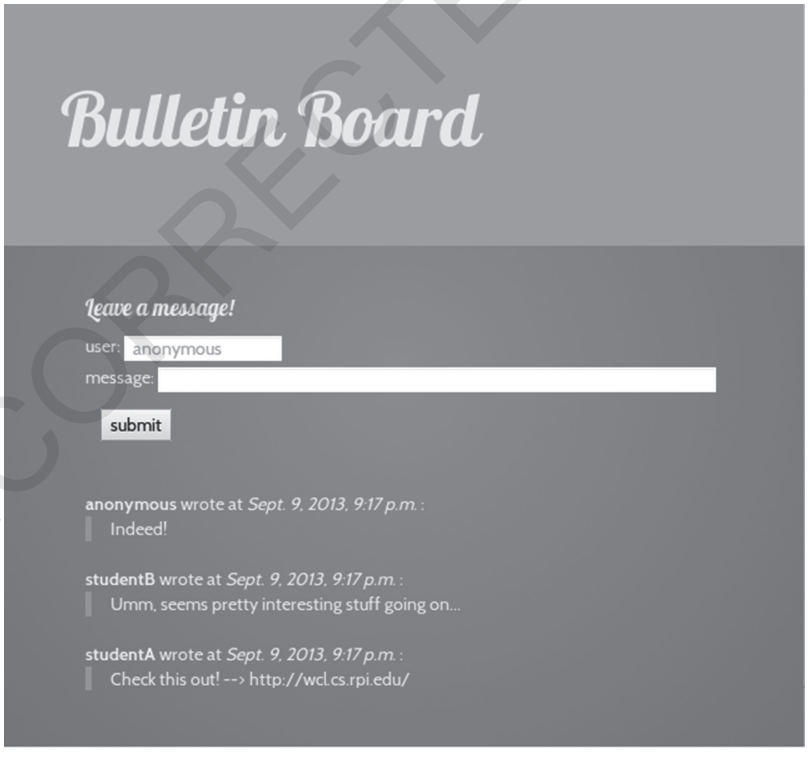

Figure 50.6 *User interface of the bulletin board application*

```
<html>
  <head>
    <link type="text/css" rel="stylesheet"
          href="/css/main.css" />
  </head>
  <body>
    <div class="headline">
      <h1>Bulletin Board</h1>
    </div>

    <div class="content">
      <h4>Leave a message!</h4>
      <form action="/postmsg" method="post" class="form-inline">
        <label>user: </label>
        <input type="text" name="user" placeholder="anonymous">
        <label>message: </label>
        <input type="text" name="content">
        <button type="submit">submit</button>
      </form>

      {% for msg in msgs %}
        <b>{{ msg.user }}</b> wrote at <i>{{ msg.date }}</i>:
        <blockquote>{{ msg.content|escape }}</blockquote>
      {% endfor %}
    </div>
  </body>
</html>
```

Listing 50.9 *Jinja2 HTML template for bulletin board application*

50.4.2.1 HTML Template

Unlike the `helloworld` example, which just returns an HTTP response containing a "*Hello, World!*" string to the browser, we can create a more elaborate and dynamically generated response using HTML templates such as *Jinja2* (Ronacher, 2011). Jinja2 is a template engine for Python supported by Google App Engine. For the bulletin board application, we can use a Jinja2 template as shown in Listing 50.9.

In the second `div` section, this HTML file first creates a form to post a username and a message to the Web server. Once these forms are filled by the user and the `submit` button is pressed, an HTTP POST request is sent to the URL specified by /`postmsg`. Then, the message-showing part is expressed using the Jinja2 template. In the Jinja2 template, "`{{ variable }}`" refers to the value of a variable. Similarly, "`{% control logic %}`" refers to control logic such as `for`, `while`, or `if`. In this example, the stored messages are referred to as `msgs` and the username, creation date, and content of each message are displayed by iterating `msgs`. Using HTML templates, we can express dynamic and static HTML contents side by side easily.

```
import os, jinja2, webapp2
 from google.appengine.ext import ndb

 JINJA_ENVIRONMENT = jinja2.Environment(
     loader=jinja2.FileSystemLoader(os.path.dirname(__file__)),
     extensions=['jinja2.ext.autoescape'])

 class Messages(ndb.Model):
     user = ndb.StringProperty()
     content =  ndb.StringProperty()
     date = ndb.DateTimeProperty(auto_now_add=True)

 class PostMessage(webapp2.RequestHandler):
     def post(self):
         msgs = Messages()
         user = self.request.get('user')
         if user == "":
           msgs.user = "anonymous"
         else:
           msgs.user = user
         msgs.content = self.request.get('content')
         msgs.put()
         self.redirect('/')

 class MainPage(webapp2.RequestHandler):
     def get(self):
       msgs = ndb.gql("SELECT * FROM Messages
                       ORDER BY date DESC LIMIT 10")
       template_values = {'msgs': msgs}
       template = JINJA_ENVIRONMENT.get_template('index.html')
       self.response.write(template.render(template_values))

 application = webapp2.WSGIApplication(
     [('/', MainPage), ('/postmsg', PostMessage)], debug=True)
```

Listing 50.10 *Application Python script for bulletin board example*

50.4.2.2 Application Script

An example script for the bulletin board application is shown in Listing 50.10.

Once the script starts, a global variable `JINJA_ENVIRONMENT` is instantiated and configured to look for Jinja2 HTML templates from the current directory (specified by "`__file__`"). This program consists of three classes: `Messages` class for data model, `PostMessage` class for handling submitted messages, and `MainPage` class for handling requests for the main page. Meanwhile, an `application` object is created by `WSGIApplication` so that HTTP requests for "/" and "/`postmsg`" are handled by the `MainPage` and `PostMessage` class respectively.

The `Message` class defines a data model that consists of three fields: `user` for the username, `content` for the message content, and `date` for the current time when the model instance is added to the datastore (specified by "`auto_now_add=True`"). The `PostMessage` class defines the behavior when the application receives an HTTP POST message. It extracts the `user` and `content` fields from the request and puts them into a `Messages` data model instance. The `MainPage` class defines the behavior when the application receives an HTTP GET message to the main Web page. Using a SQL-like query called GQL, it retrieves stored messages from the `Messages` datastore up to ten in descending order of date (latest date comes first), and then passes the retrieved `msgs` messages to the template `index.html`. Finally, it creates a complete HTTP response from the template and returns the response to the browser.

50.4.3 Distributed Face Recognition

Face recognition is an application that can be vastly improved by leveraging cloud computing resources. Rather than using a single device, in this case a mobile phone, we can offload parts of the image processing to the cloud (Abolfazli *et al.,* 2014). By using cloud computing, we can save the battery in the mobile device, and also we can consider larger data sets. Using the SALSA programming language and the `FaceRecognizer` API from OpenCV (http://opencv.org/, accessed January 5, 2016), we can design a mobile phone application to recognize a face in a given image using a database of faces (see Figure 50.7).

The face recognition application consists of two stages: the training stage, and the prediction stage. The training stage trains a database of faces using the `FaceRecognizer` method defined in OpenCV. The prediction stage predicts a given face using the desired method with a certain confidence. When the database of faces is small, there is no need to offload computation because the phone can process the faces locally just as fast as it would take to offload to the cloud. As the database grows we run into the limitations mentioned above and offloading to the cloud can be beneficial.

The distributed face recognition model consists of a "farmer actor" that creates *N1* "worker actors" in the cloud. While the farmer and worker actors reside in the cloud, a client on the mobile phone requests the farmer actor to recognize an unknown face. The farmer actor assigns each worker actor a range of faces (*N2/N1* each)

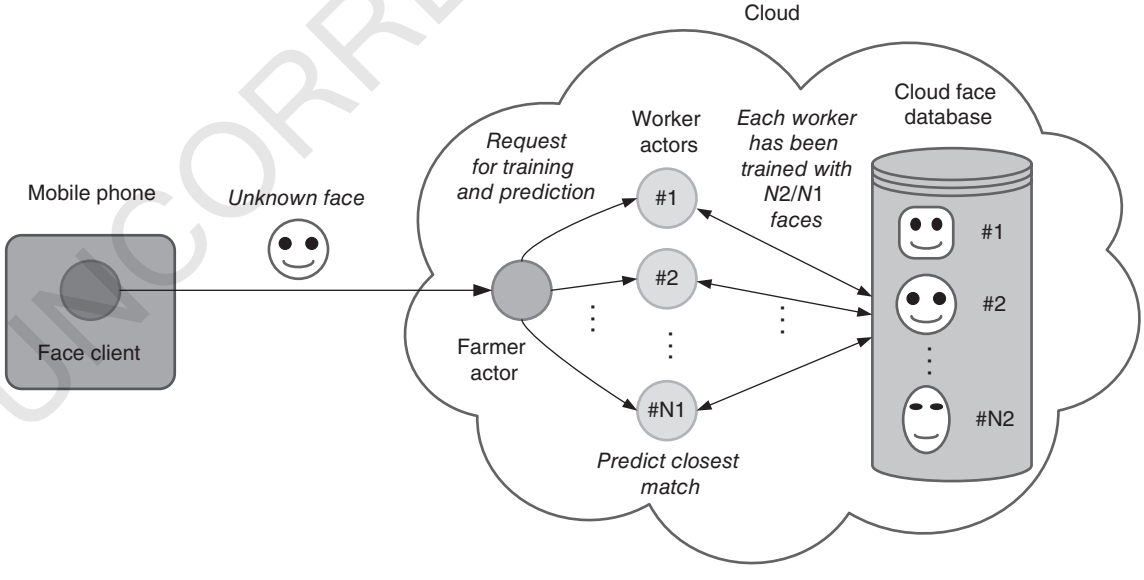

Figure 50.7 *Distributed face recognition using SALSA actors*

to train from the cloud face database containing *N2* faces. The worker actors then predict the closest match to the unknown face based on their assigned database. The farmer actor collects the closest match from each worker actor and calculates the best candidate for the unknown face. Pseudo programs for the client, worker and farmer SALSA programs are shown in Listings 50.11, 50.12, and 50.13 respectively.

In Listing 50.13, note that a `join` block in the `predictAll` message handler is used to synchronize all the worker actors executing predict. After all the workers finish processing `predict`, the join block returns an object array `ImageMetaData[]`, which contains prediction confidence from the workers. Finally, `getBestMatch` takes the array and finds the best matching image with the highest confidence.

50.5 Conclusions

Cloud computing has the potential to bring the benefits of large-scale data analytics and high-performance computing to everyone's fingertips enabling unprecedented societal applications. We have described three programming frameworks for cloud computing: Hadoop's MapReduce, Google App Engine, and SALSA. To illustrate the use of these frameworks, we have shown a weather data analytics application on Hadoop, a

```
behavior FaceClient {
    void act(String[] args) {
        FaceFarmer farmer = (FaceFarmer)
            FaceFarmer.getReferenceByName(
                "uan://nameserver/facefarmer");
        Image unknownImage = new Image(args[0]);
        farmer<-predictAll(unknownImage)@
            displayImage(token);
    }
}
```

Listing 50.11 *SALSA pseudo code for face recognition client actor*

```
behavior FaceWorker {
    FaceRcognizer faceRecognizer;
    void train(Image[] assignedDatabase) {
        faceRecognizer.train(assignedDatabase);
    }

    ImageMetaData preditct (Image testImage) {
        // return the closest match
        return faceRecognizer.predict(testImage);
    }
}
```

Listing 50.12 *SALSA pseudo code for face recognition worker actor*

```
behavior FaceFarmer implements ActorService {
   //CloudFaceDatabase contains N2 faces.
   void act(String[] args) {
      faceWorker[] workers = new faceWorker[N1];
      for (i = 0; i < N1; i++) {
          workers[i] = new faceWorker(new UAN(…));
          workers[i]<-migrate(new UAL(…));
          workers[i]<-train(
                N2/N1 faces from CloudFaceDatabase);

   }
}
 Image predictAll(Image testImage)
    join {
       for (i = 0; i < N1; i++)
           workers[i]<-predict(testImage);
    }@getBestMatch(token)@currentContinuation;
}
 Image getBestMatch(ImageMetaData[] closestMatches) {
     // find best match with the highest confidence.
 }
}
```

Listing 50.13 *SALSA pseudo code for face recognition farmer actor*

simple bulletin board Web application on Google App Engine, and a distributed face recognition application on SALSA offloading computation from a mobile device to the cloud. As we have seen, the target applications for these frameworks are very different from each other, and the support level of nonfunctional concerns is also different between IaaS and PaaS. These frameworks are ideal for the illustrated target application domains, but not necessarily for other application scenarios. Therefore, to get the best from these cloud services, application developers still need to carefully consider the characteristics of their applications and find best matching cloud services in terms of QoS, cost, programmability, vendor lock-in possibility, and others.

Acknowledgment

This work is partially supported by an Amazon AWS in Education Research Grant and a Google Cloud Credits Award.

References

Abolfazli, S., Sanaei, Z., Ahmed, E., Gani, A., and Buyya, R. (2014) Cloud-based augmentation for mobile devices: Motivation, taxonomies, and open challenges. *IEEE Communications Surveys and Tutorials* **16**(1), 337–368.

Agha, G. (1986) *Actors: a Model of Concurrent Computation in Distributed Systems,* MIT Press, Cambridge, MAs.

Armstrong, J., Virding, R., Wikström, C., and Williams, M. (1993) *Concurrent Programming in ERLANG,* Prentice Hall, Englewood Cliffs, NJ.

Chang, F., Dean, J., Ghemawat, S., *et al.* (2008) Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* **26**(2), 4.

Dean, J. and Ghemawat, S. (2008) MapReduce: Simplified data processing on large clusters. *Communications of the ACM* **51**(1), 107–113.

El Maghraoui, K., Desell, T. J., Szymanski, B. K., and Varela, C. A. (2006) The internet operating system: Middleware for adaptive distributed computing. *International Journal of High Performance Computing Applications* **20**(4), 467–480.

Google (2014) *Google App Engine*, https://cloud.google.com/appengine/docs (accessed January 5, 2016).

Imai, S., Chestna, T., and Varela, C. A. (2013) *Accurate Resource Prediction for Hybrid IaaS Clouds Using Workload-Tailored Elastic Compute Units.* Proceedings of the 2013 IEEE Sixth International Conference on Utility and Cloud Computing (UCC). IEEE.

Heroku. (2014) *Heroku*, http://www.heroku.com/ (accessed January 5, 2016).

Microsoft (2014) *Windows Azure*, http://www.windowsazure.com/ (accessed January 5, 2016).

Ronacher, A. (2011) *Jinja*, http://jinja.pocoo.org/ (accessed January 5, 2016).

Varela, C. A. & Agha, G. (2001) Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Notices* **36**(12), 20–34.

Varela, C. A. (2013) *Programming Distributed Computing Systems,* MIT, Cambridge, MA.

Xin, R. S., Rosen, J., Zaharia, M., *et al.* (2013) *Shark: SQL and Rich Analytics at Scale.* Proceedings of the 2013 International Conference on Management of Data. ACM, pp. 13–24.