Chapter 8. Geospatial and Temporal Data Analysis on New York City Taxi Trip Data

Josh Wills

Nothing puzzles me more than time and space; and yet nothing troubles me less, as I never think about them.

Charles Lamb

New York City is widely known for its yellow taxis, and hailing one is just as much a part of the experience of visiting the city as eating a hot dog from a street vendor or riding the elevator to the top of the Empire State Building.

Residents of New York City have all kinds of tips based on their anecdotal experiences about the best times and places to catch a cab, especially during rush hour and when it's raining. But there is one time of day when everyone will recommend that you simply take the subway instead: during the shift change that happens between 4 and 5PM every day. During this time, yellow taxis have to return to their dispatch centers (often in Queens) so that one driver can quit for the day and the next one can start, and drivers who are late to return have to pay fines.

In March of 2014, the New York City Taxi and Limousine Commission shared an infographic on its Twitter account, @nyctaxi, that showed the number of taxis on the road and the fraction of those taxis that was occupied at any given time. Sure enough, there was a noticeable dip of taxis on the road from 4 to 6PM, and two-thirds of the taxis that were driving were occupied.

This tweet caught the eye of self-described urbanist, mapmaker, and data junkie Chris Whong, who sent a tweet to the @nyctaxi account to find out if the data it used in its infographic was publicly available. The taxi commission replied that he could have the data if he filed a Freedom of Information Law (FOIL) request and provided the commission with hard drives that they could copy the data on to. After filling out one PDF form, buying two new 500 GB hard drives, and waiting two business days, Chris had access to all of the data on taxi rides from January 1 through December 31, 2013. Even better, he posted all of the fare data online, where it has been used as the basis for a number of beautiful visualizations of transportation in New York City.

One statistic that is important to understanding the economics of taxis is *utilization*: the fraction of time that a cab is on the road and is occupied by one or more passengers. One factor that impacts utilization is the passenger's destination: a cab that drops off passengers near Union Square at midday is much more likely to find its next fare in just a minute or two, whereas a cab that drops someone off at 2AM on Staten Island may have to drive all the way back to Manhattan before it finds its next fare. We'd like to quantify these effects and find out the average time it takes for a cab to find its next fare as a function of the borough in which it dropped its passengers off—Manhattan, Brooklyn, Queens, the Bronx, Staten Island, or none of the above (e.g., if it dropped the passenger off somewhere outside of the city, like Newark International Airport).

To carry out this analysis, we need to deal with two types of data that come up all the time: *temporal data*, such as dates and times, and *geospatial information*, like points of longitude and latitude and spatial boundaries. Since the first edition of this book was released, there have been a number of improvements in the ways that we can work with temporal data in Spark, such as the new `java.time`package that was released in Java 8 and the incorporation of UDFs from the Apache Hive project in SparkSQL; these include many functions that deal with time, like `date_add` and `from_timestamp`, which make it much easier to work with time in Spark 2 than it was in Spark 1. Geospatial data, on the other hand, is still a fairly specialized kind of analysis, and we will need to work with third-party libraries and write our own custom UDFs in order to be able to effectively work with this data inside Spark.

Getting the Data

For this analysis, we're only going to consider the fare data from January 2013, which will be about 2.5 GB of data after we uncompress it. You can access the data for each month of 2013, and if you have a sufficiently large Spark cluster at your disposal, you can re-create the following analysis against all of the data for the year. For now, let's create a working directory on our client machine and take a look at the structure of the fare data:

```
$ mkdir taxidata
$ cd taxidata
$ curl -O https://nyctaxitrips.blob.core.windows.net/data/trip_data_1.csv.zip
$ unzip trip_data_1.csv.zip
$ head -n 10 trip_data_1.csv
```

Each row of the file after the header represents a single taxi ride in CSV format. For each ride, we have some attributes of the cab (a hashed version of the medallion number) as well as the driver (a hashed version of the *hack license*, which is what licenses to drive taxis are called), some temporal information about when the trip started and ended, and the longitude/latitude coordinates for where the passenger(s) were picked up and dropped off.

Working with Third-Party Libraries in Spark

One of the great features of the Java platform is the sheer volume of code that has been developed for it over the years: for any kind of data type or algorithm you might need to use, it's likely that someone else has written a Java library that you can use to solve your problem, and there's also a good chance that an open source version of that library exists that you can download and use without having to purchase a license.

Of course, just because a library exists and is freely available doesn't mean you necessarily want to rely on it to solve your problem. Open source projects have a lot of variation in terms of their quality, state of development in terms of bug fixes and new features, and ease-of-use in terms of API design and the presence of useful documentation and tutorials.

Our decision-making process is a bit different than that of a developer choosing a library for an application; we want something that will be pleasant to use for interactive data analysis and that is easy to use in a distributed application. In particular, we want to be sure that the main data types we will be working with in our RDDs implement the `Serializable` interface and/or can be easily serialized using libraries like Kryo.

Additionally, we would like the libraries we use for interactive data analysis to have as few external dependencies as possible. Tools like Maven and SBT can help application developers deal with complex dependencies when building applications, but for interactive data analysis, we would much rather simply grab a JAR file with all of the code we need, load it into the Spark shell, and start our analysis. Additionally, bringing in libraries with lots of dependencies can cause version conflicts with other libraries that Spark depends on, which can cause difficult-to-diagnose error conditions that developers refer to as *JAR hell*.

Finally, we would like our libraries to have relatively simple and rich APIs that do not make extensive use of Java-oriented design patterns like abstract factories and visitors. Although these patterns can be very useful for application developers, they tend to add a lot of complexity to our code that is unrelated to our analysis. Even better, many Java libraries have Scala wrappers that take advantage of Scala's power to reduce the amount of boilerplate code required to use them.

Geospatial Data with the Esri Geometry API and Spray

Working with temporal data on the JVM has become significantly easier in Java 8 thanks to the `java.time` package, whose design is based on the highly successful JodaTime library. For geospatial data, the answer isn't nearly so simple; there are many different libraries and tools that have different functions, states of development, and maturity levels, so there is not a dominant Java library for all geospatial use cases.

The first thing you must consider when choosing a library is determine what kind of geospatial data you will need to work with. There are two major kinds—vector and raster—and there are different tools for working with each type. In our case, we have latitude and longitude for our taxi trip records, and vector data stored in the GeoJSON format that represents the boundaries of the different boroughs of New York. So we need a library that can parse GeoJSON data and can handle spatial relationships, like detecting whether a given longitude/latitude pair is contained inside a polygon that represents the boundaries of a particular borough.

Unfortunately, there isn't an open source library that fits our needs exactly. There is a GeoJSON parser library that can convert GeoJSON records into Java objects, but there isn't an associated geospatial library that can analyze spatial relationships on the generated objects. There is the GeoTools project, but it has a long list of components and dependencies—exactly the kind of thing we try to avoid when choosing a library to work with from the Spark shell. Finally, there is the Esri Geometry API for Java, which has few

dependencies and can analyze spatial relationships but can only parse a subset of the GeoJSON standard, so it won't be able to parse the GeoJSON data we downloaded without us doing some preliminary data munging.

For a data analyst, this lack of tooling might be an insurmountable problem. But we are data scientists: if our tools don't allow us to solve a problem, we build new tools. In this case, we will add Scala functionality for parsing *all* of the GeoJSON data, including the bits that aren't handled by the Esri Geometry API, by leveraging one of the many Scala projects that support parsing JSON data. The code that we will be discussing in the next few sections is available in the book's Git repo, but has also been made available as a standalone library on GitHub, where it can be used for any kind of geospatial analysis project in Scala.

Exploring the Esri Geometry API

The core data type of the Esri library is the `Geometry` object. A `Geometry` describes a shape, accompanied by a geolocation where that shape resides. The library contains a set of spatial operations that allows analyzing geometries and their relationships. These operations can do things like tell us the area of a geometry and whether two geometries overlap, or compute the geometry formed by the union of two geometries.

In our case, we'll have `Geometry` objects representing dropoff points for cab rides (longitude and latitude), and `Geometry` objects that represent the boundaries of a borough in NYC. The spatial relationship we're interested in is containment: is a given point in space located inside one of the polygons associated with a borough of Manhattan?

The Esri API provides a convenience class called `GeometryEngine` that contains static methods for performing all of the spatial relationship operations, including a `contains` operation. The `contains` method takes three arguments: two `Geometry` objects, and one instance of the `SpatialReference` class, which represents the coordinate system used to perform the geospatial calculations. For maximum precision, we need to analyze spatial relationships relative to a coordinate plane that maps each point on the misshapen spheroid that is planet Earth into a 2D coordinate system. Geospatial engineers have a standard set of well-known identifiers (referred to as WKIDs) that can be used to reference the most commonly used coordinate systems. For our purposes, we will be using WKID 4326, which is the standard coordinate system used by GPS.

As Scala developers, we're always on the lookout for ways to reduce the amount of typing we need to do as part of our interactive data analysis in the Spark shell, where we don't have access to development environments like Eclipse and IntelliJ that can automatically complete long method names for us and provide some syntactic sugar to make it easier to read certain kinds of operations. Following the naming convention we saw in the NScalaTime library, which defined wrapper classes like `RichDateTime` and `RichDuration`, we'll define our own `RichGeometry` class that extends the Esri `Geometry` object with some useful helper methods:

```
import com.esri.core.geometry.Geometry
```

```
import com.esri.core.geometry.GeometryEngine
import com.esri.core.geometry.SpatialReference

class RichGeometry(val geometry: Geometry,
    val spatialReference: SpatialReference =
      SpatialReference.create(4326)) {
  def area2D() = geometry.calculateArea2D()

  def contains(other: Geometry): Boolean = {
    GeometryEngine.contains(geometry, other, spatialReference)
  }

  def distance(other: Geometry): Double = {
    GeometryEngine.distance(geometry, other, spatialReference)
  }
}
```

We'll also declare a companion object for `RichGeometry` that provides support for implicitly converting instances of the `Geometry` class into `RichGeometry` instances:

```
object RichGeometry {
  implicit def wrapRichGeo(g: Geometry) = {
    new RichGeometry(g)
  }
}
```

Remember, to be able to take advantage of this conversion, we need to import the implicit function definition into the Scala environment, like this:

```
import RichGeometry._
```

Intro to GeoJSON

The data we'll use for the boundaries of boroughs in New York City comes written in a format called *GeoJSON*. The core object in GeoJSON is called a *feature*, which is made up of a *geometry* instance and a set of key-value pairs called *properties*. A geometry is a shape like a point, line, or polygon. A set of features is called a `FeatureCollection`. Let's pull down the GeoJSON data for the NYC borough maps and take a look at its structure.

In the *taxidata* directory on your client machine, download the data and rename the file to something a bit shorter:

```
$ curl -O https://nycdatastables.s3.amazonaws.com/2013-08-19T18:15:35.172Z/
  nyc-borough-boundaries-polygon.geojson
$ mv nyc-borough-boundaries-polygon.geojson nyc-boroughs.geojson
```

Open the file and look at a feature record. Note the properties and the geometry objects—in this case, a polygon representing the boundaries of the borough, and the properties containing the name of the borough and other related information.

The Esri Geometry API will help us parse the `Geometry` JSON inside each feature but won't help us parse the `id` or the `properties` fields, which can be arbitrary JSON objects. To parse these objects, we need to use a Scala JSON library, of which there are many to choose from.

Spray, an open source toolkit for building web services with Scala, provides a JSON library that is up to the task. *spray-json* allows us to convert any Scala object to a corresponding `JsValue` by calling an implicit `toJson` method. It also allows us to convert any `String` that contains JSON to a parsed intermediate form by calling `parseJson`, and then converting it to a Scala type `T` by calling `convertTo[T]` on the intermediate type. Spray comes with built-in conversion implementations for the common Scala primitive types as well as tuples and collection types, and it also has a formatting library that allows us to declare the rules for converting custom types like our `RichGeometry` class to and from JSON.

First, we'll need to create a case class for representing GeoJSON features. According to the specification, a feature is a JSON object that is required to have one field named "geometry" that corresponds to a GeoJSON `Geometry` type, and one field named "properties" that is a JSON object with any number of key-value pairs of any type. A feature may also have an optional "id" field that may be any JSON identifier. Our `Feature` case class will define corresponding Scala fields for each of the JSON fields, and will add some convenience methods for looking up values from the map of properties:

```
import spray.json.JsValue

case class Feature(
    val id: Option[JsValue],
    val properties: Map[String, JsValue],
    val geometry: RichGeometry) {
  def apply(property: String) = properties(property)
  def get(property: String) = properties.get(property)
}
```

We're representing the `Geometry` field in `Feature` using an instance of our `RichGeometry`class, which we'll create with the help of the GeoJSON geometry parsing functions from the Esri Geometry API.

We'll also need a case class that corresponds to the GeoJson `FeatureCollection`. To make the `FeatureCollection` class a bit easier to use, we will have it extend the `IndexedSeq[Feature]` trait by implementing the appropriate `apply` and `length` methods so that we can call the standard Scala Collections API methods like `map`, `filter`, and `sortBy` directly on the `FeatureCollection` instance itself, without having to access the underlying `Array[Feature]` value that it wraps:

```
case class FeatureCollection(features: Array[Feature])
    extends IndexedSeq[Feature] {
  def apply(index: Int) = features(index)
  def length = features.length
}
```

After we have defined the case classes for representing the GeoJSON data, we need to define the formats that tell Spray how to convert between our domain objects (`RichGeometry`, `Feature`, and `FeatureCollection`) and a corresponding `JsValue` instance. To do this, we need to create Scala singleton objects that extend the `RootJsonFormat[T]` trait, which defines abstract `read(jsv: JsValue): T` and `write(t: T): JsValue` methods. For the `RichGeometry`class, we can delegate most of the parsing and formatting logic to the Esri Geometry API, particularly the `geometryToGeoJson` and `geometryFromGeoJson` methods on the `GeometryEngine`class. But for our case classes, we need to write the formatting code ourselves. Here's the formatting code for the `Feature` case class, including some special logic to handle the optional `id` field:

```
implicit object FeatureJsonFormat extends
    RootJsonFormat[Feature] {
  def write(f: Feature) = {
    val buf = scala.collection.mutable.ArrayBuffer(
      "type" -> JsString("Feature"),
      "properties" -> JsObject(f.properties),
      "geometry" -> f.geometry.toJson)
    f.id.foreach(v => { buf += "id" -> v})
    JsObject(buf.toMap)
  }

  def read(value: JsValue) = {
    val jso = value.asJsObject
    val id = jso.fields.get("id")
    val properties = jso.fields("properties").asJsObject.fields
    val geometry = jso.fields("geometry").convertTo[RichGeometry]
    Feature(id, properties, geometry)
  }
}
```

The `FeatureJsonFormat` object uses the `implicit` keyword so that the Spray library can look it up when the `convertTo[Feature]` method is called on an instance of `JsValue`. You can see the rest of the `RootJsonFormat` implementations in the source code for the GeoJSON library on GitHub.

Preparing the New York City Taxi Trip Data

With the GeoJSON and JodaTime libraries in hand, it's time to begin analyzing the NYC taxi trip data interactively using Spark. Let's create a *taxidata* directory in HDFS and copy the trip data into the cluster:

```
$ hadoop fs -mkdir taxidata
$ hadoop fs -put trip_data_1.csv taxidata/
```

Now start the Spark shell, using the `--jars` argument to make the libraries we need available in the REPL:

```
$ mvn package
$ spark-shell --jars ch08-geotime-2.0.0-jar-with-dependencies.jar
```

Once the Spark shell has loaded, we can create a data set from the taxi data and examine the first few lines, just as we have in other chapters:

```
val taxiRaw = spark.read.option("header", "true").csv("taxidata")
taxiRaw.show()
```

The taxi data appears to be a well-formatted CSV file with clearly defined data types. In Chapter 2, we used the built-in type inference library included in `spark-csv` to automatically convert our CSV data from strings to column-specific types. The cost of this automatic conversion is two-fold. First, we need to do an extra pass over the data so that the converter can infer the type of each column. Second, if we only want to use a subset of the columns in the data set for our analysis, we will need to spend extra resources to do type inference on columns that we will end up dropping immediately when we begin our analysis. For smaller data sets, like the record linkage data we analyzed in Chapter 2, these costs are relatively inconsequential. But for very large data sets that we're planning on analyzing again and again, it may be a better use of our time to create code for performing custom type conversions on just the subset of columns we know we're going to need. In this chapter, we're going to opt to do the conversion ourselves via custom code.

Let's begin by defining a case class that contains just the information about each taxi trip that we want to use in our analysis. Since we're going to use this case class as the basis for a data set, we need to be mindful of the fact that a data set can only be optimized for a relatively small set of data types, including `Strings`, primitives (like `Int`, `Double`, etc.), and certain special Scala types like `Option`. If we want to take advantage of the performance enhancements and analysis utilities that the `Dataset`class provides, our case class can only contain fields that belong to this small set of supported types. (Note that the code listings below are only illustrative extracts from the complete code that you will need to execute to follow along with this chapter. Please refer to the accompanying Chapter 8 source code repository, in particular *GeoJson.scala*.)

```
case class Trip(
  license: String,
  pickupTime: Long,
  dropoffTime: Long,
  pickupX: Double,
  pickupY: Double,
  dropoffX: Double,
  dropoffY: Double)
```

We are representing the `pickupTime` and `dropoffTime` fields as `Longs` that are the number of milliseconds since the Unix epoch, and storing the individual `xy` coordinates of the pickup and dropoff locations in their own fields, even though we will typically work with them by converting these values to instances of the `Point` class in the Esri API.

To parse the `Rows` from the `taxiRaw` data set into instances of our case class, we will need to create some helper objects and functions. First, we need to be mindful of the fact that it's likely that some of the fields in a row may be missing from the data, so it's possible that when we go to retrieve them from the `Row`, we'll first need to check to see if they are `null` before we retrieve them or else we'll get an error. We can write a small helper class to handle this problem for any kind of `Row` we need to parse:

```
class RichRow(row: org.apache.spark.sql.Row) {
  def getAs[T](field: String): Option[T] = {
    if (row.isNullAt(row.fieldIndex(field))) {
      None
    } else {
      Some(row.getAs[T](field))
    }
  }
}
```

In the `RichRow` class, the `getAs[T]` method always returns an `Option[T]` instead of the raw value directly so that we can explicitly handle a situation in which a field is missing when we parse a `Row`. In this case, all of the fields in our data set are `Strings`, so we'll be working with values of type `Option[String]`.

Next, we'll need to process the pickup and dropoff times using an instance of Java's `SimpleDateFormat` class with an appropriate formatting string to get the time in milliseconds:

```
def parseTaxiTime(rr: RichRow, timeField: String): Long = {
 val formatter = new SimpleDateFormat(
     "yyyy-MM-dd HH:mm:ss", Locale.ENGLISH)
 val optDt = rr.getAs[String](timeField)
 optDt.map(dt => formatter.parse(dt).getTime).getOrElse(0L)
}
```

Then we will parse the longitude and latitude of the pickup and dropoff locations from `Strings` to `Doubles` using Scala's implicit `toDouble` method, defaulting to a value of `0.0` if the coordinate is missing:

```
def parseTaxiLoc(rr: RichRow, locField: String): Double = {
  rr.getAs[String](locField).map(_.toDouble).getOrElse(0.0)
}
```

Putting these functions together, our resulting `parse` method looks like this:

```
def parse(row: org.apache.spark.sql.Row): Trip = {
  val rr = new RichRow(row)
  Trip(
    license = rr.getAs[String]("hack_license").orNull,
    pickupTime = parseTaxiTime(rr, "pickup_datetime"),
    dropoffTime = parseTaxiTime(rr, "dropoff_datetime"),
    pickupX = parseTaxiLoc(rr, "pickup_longitude"),
```

```
      pickupY = parseTaxiLoc(rr, "pickup_latitude"),
      dropoffX = parseTaxiLoc(rr, "dropoff_longitude"),
      dropoffY = parseTaxiLoc(rr, "dropoff_latitude")
    )
  }
```

We can test the `parse` function on several records from the head of the `taxiRaw` data to verify that it can correctly handle a sample of the data.

Handling Invalid Records at Scale

Anyone who has been working with large-scale, real-world data sets knows that they invariably contain at least a few records that do not conform to the expectations of the person who wrote the code to handle them. Many MapReduce jobs and Spark pipelines have failed because of invalid records that caused the parsing logic to throw an exception.

Typically, we handle these exceptions one at a time by checking the logs for the individual tasks, figuring out which line of code threw the exception, and then figuring out how to tweak the code to ignore or correct the invalid records. This is a tedious process, and it often feels like we're playing whack-a-mole: just as we get one exception fixed, we discover another one on a record that came later within the partition.

One strategy that experienced data scientists deploy when working with a new data set is to add a `try-catch` block to their parsing code so that any invalid records can be written out to the logs without causing the entire job to fail. If there are only a handful of invalid records in the entire data set, we might be okay with ignoring them and continuing with our analysis. With Spark, we can do even better: we can adapt our parsing code so that we can interactively analyze the invalid records in our data just as easily as we would perform any other kind of analysis.

For any individual record in an RDD or data set, there are two possible outcomes for our parsing code: it will either parse the record successfully and return meaningful output, or it will fail and throw an exception, in which case we want to capture both the value of the invalid record and the exception that was thrown. Whenever an operation has two mutually exclusive outcomes, we can use Scala's `Either[L, R]` type to represent the return type of the operation. For us, the "left" outcome is the successfully parsed record and the "right" outcome is a tuple of the exception we hit and the input record that caused it.

The `safe` function takes an argument named `f` of type `S => T` and returns a new `S => Either[T, (S, Exception)]` that will return either the result of calling `f` or, if an exception is thrown, a tuple containing the invalid input value and the exception itself:

```
def safe[S, T](f: S => T): S => Either[T, (S, Exception)] = {
  new Function[S, Either[T, (S, Exception)]] with Serializable {
    def apply(s: S): Either[T, (S, Exception)] = {
      try {
        Left(f(s))
      } catch {
```

```
        case e: Exception => Right((s, e))
      }
    }
  }
}
```

We can now create a safe wrapper function called `safeParse` by passing our `parse` function (of type `String => Trip`) to the `safe` function, and then applying `safeParse` to the backing RDD of the `taxiRaw` data set:

```
val safeParse = safe(parse)
val taxiParsed = taxiRaw.rdd.map(safeParse)
```

(Note that we cannot apply `safeParse` to the `taxiRaw` data set directly because the `Either[L, R]` type isn't supported by the Dataset API.)

If we want to determine how many of the input lines were parsed successfully, we can use the `isLeft` method on `Either[L, R]` in combination with the `countByValue` action:

```
taxiParsed.map(_.isLeft).
countByValue().
foreach(println)
...
(true,14776615)
```

What luck—none of the records threw exceptions during parsing! We can now convert the `taxiParsed` RDD into a `Dataset[Trip]` instance by getting the left element of the `Either`value:

```
val taxiGood = taxiParsed.map(_.left.get).toDS
taxiGood.cache()
```

Even though the records in the `taxiGood` data set parsed correctly, they may still have data quality problems that we want to uncover and handle. To find the remaining data quality problems, we can start to think of conditions that we expect to be true for any correctly recorded trip.

Given the temporal nature of our trip data, one reasonable invariant that we can expect is that the dropoff time for any trip will be sometime after the pickup time. We might also expect that trips will not take more than a few hours to complete, although it's certainly possible that long trips, trips that take place during rush hour, or trips that are delayed by accidents could go on for several hours. We're not exactly sure what the cutoff should be for a trip that takes a "reasonable" amount of time.

Let's define a helper function named `hours` that uses Java's `TimeUnit` helper method to convert the difference of the pickup and dropoff times in milliseconds to hours:

```
val hours = (pickup: Long, dropoff: Long) => {
```

```
  TimeUnit.HOURS.convert(dropoff - pickup, TimeUnit.MILLISECONDS)
}
```

We would like to be able to use our `hours` function to compute a histogram of the number of trips that lasted at least a given number of hours. This sort of calculation is exactly what the Dataset API and Spark SQL are designed to do, but by default, we can only use these methods on the columns of a `Dataset` instance, whereas the `hour` UDF is computed from two of these columns—the `pickupTime` and the `dropoffTime`. We need a mechanism that allows us to apply the `hours`functions to the columns of a data set and then perform the normal filtering and grouping operations that we are familiar with to the results.

This is exactly the use case that Spark SQL UDFs are designed to address. By wrapping a Scala function in an instance of Spark's `UserDefinedFunction` class, we can apply the function to the columns of a data set and analyze the results. Let's start by wrapping `hours` in a UDF and computing our histogram:

```
import org.apache.spark.sql.functions.udf
val hoursUDF = udf(hours)
taxiGood.
  groupBy(hoursUDF($"pickupTime", $"dropoffTime").as("h")).
  count().
  sort("h").
  show()
...
+---+--------+
|  h|   count|
+---+--------+
| -8|       1|
|  0|22355710|
|  1|   22934|
|  2|     843|
|  3|     197|
|  4|      86|
|  5|      55|
...
```

Everything looks fine here, except for one trip that took a -8 hours to complete! Perhaps the DeLorean from *Back to the Future* is moonlighting as an NYC taxi? Let's examine this record:

```
taxiGood.
  where(hoursUDF($"pickupTime", $"dropoffTime") < 0).
  collect().
  foreach(println)
```

This reveals the one odd record—a trip that began around 6PM on January 25 and finished just before 10AM the same day. It isn't obvious what went wrong with the recording of this trip but because it only seemed to happen for a single record, it should be okay to exclude it from our analysis for now.

Looking at the remainder of the trips that went on for a nonnegative number of hours, it appears that the vast majority of taxi rides last for no longer than three hours. We'll apply a filter to the `taxiGood`RDD so that we can focus on the distribution of these "typical" rides and ignore the outliers for now. Because it's a bit easier to express this filtering condition in Spark SQL, let's register our `hours` function with Spark SQL under the name "hours", so that we can use it inside SQL expressions:

```
spark.udf.register("hours", hours)
val taxiClean = taxiGood.where(
  "hours(pickupTime, dropoffTime) BETWEEN 0 AND 3"
)
```

TO UDF OR NOT TO UDF?

Spark SQL makes it very easy to inline business logic into functions that can be used from standard SQL, as we did here with the `hours` function. Given this, you might think that it would be a good idea to move all of your business logic into UDFs in order to make it easy to reuse, test, and maintain. However, there are a few caveats for using UDFs that you should be mindful of before you start sprinkling them throughout your code.

First, UDFs are opaque to Spark's SQL query planner and execution engine in a way that standard SQL query syntax is not, so moving logic into a UDF instead of using a literal SQL expression could hurt query performance.

Second, handling null values in Spark SQL can get complicated quickly, especially for UDFs that take multiple arguments. To properly handle nulls, you need to use Scala's `Option[T]` type or write your UDFs using the Java wrapper types, like `java.lang.Integer` and `java.lang.Double`, instead of the primitive types `Int` and `Double` in Scala.

Geospatial Analysis

Let's start examining the geospatial aspects of the taxi data. For each trip, we have longitude/latitude pairs representing where the passenger was picked up and dropped off. We would like to be able to determine which borough each of these longitude/latitude pairs belongs to, and identify any trips that did not start or end in any of the five boroughs. For example, if a taxi took passengers from Manhattan to Newark International Airport, that would be a valid ride that would be interesting to analyze, even though it would not end within one of the five boroughs. However, if it looks as if a taxi took a passenger to the South Pole, we can be reasonably confident that the record is invalid and should be excluded from our analysis.

To perform our borough analysis, we need to load the GeoJSON data we downloaded earlier and stored in the *nyc-boroughs.geojson* file. The `Source` class in the `scala.io` package makes it easy to read the contents of a text file or URL into the client as a single `String`:

```
val geojson = scala.io.Source.
  fromFile("nyc-boroughs.geojson").
  mkString
```

Now we need to import the GeoJSON parsing tools we reviewed earlier in the chapter using Spray and Esri into the Spark shell so that we can parse the `geojson` string into an instance of our `FeatureCollection` case class:

```
import com.cloudera.datascience.geotime._
import GeoJsonProtocol._
import spray.json._

val features = geojson.parseJson.convertTo[FeatureCollection]
```

We can create a sample point to test the functionality of the Esri Geometry API and verify that it can correctly identify which borough a particular `xy` coordinate belongs to:

```
import com.esri.core.geometry.Point
val p = new Point(-73.994499, 40.75066)
val borough = features.find(f => f.geometry.contains(p))
```

Before we use the `features` on the taxi trip data, we should take a moment to think about how to organize this geospatial data for maximum efficiency. One option would be to research data structures that are optimized for geospatial lookups, such as quad trees, and then find or write our own implementation. But let's see if we can come up with a quick heuristic that will allow us to bypass that bit of work.

The `find` method will iterate through the `FeatureCollection` until it finds a feature whose geometry contains the given `Point` of longitude/latitude. Most taxi rides in NYC begin and end in Manhattan, so if the geospatial features that represent Manhattan are earlier in the sequence, most of the `find` calls will return relatively quickly. We can use the fact that the `boroughCode` property of each feature can be used as a sorting key, with the code for Manhattan equal to 1 and the code for Staten Island equal to 5. Within the features for each borough, we want the features associated with the largest polygons to come before the smaller polygons, because most trips will be to and from the "major" region of each borough. Sorting the features by the combination of the borough code and the `area2D()` of each feature's geometry should do the trick:

```
val areaSortedFeatures = features.sortBy(f => {
  val borough = f("boroughCode").convertTo[Int]
  (borough, -f.geometry.area2D())
})
```

Note that we're sorting based on the negation of the `area2D()` value because we want the largest polygons to come first, and Scala sorts in ascending order by default.

Now we can broadcast the sorted features in the `areaSortedFeatures` sequence to the cluster and write a function that uses these features to find out in which of the five boroughs (if any) a particular trip ended:

```
val bFeatures = sc.broadcast(areaSortedFeatures)
```

```
val bLookup = (x: Double, y: Double) => {
  val feature: Option[Feature] = bFeatures.value.find(f => {
    f.geometry.contains(new Point(x, y))
  })
  feature.map(f => {
    f("borough").convertTo[String]
  }).getOrElse("NA")
}
val boroughUDF = udf(bLookup)
```

We can apply `boroughUDF` to the trips in the `taxiClean` RDD to create a histogram of trips by borough:

```
taxiClean.
  groupBy(boroughUDF($"dropoffX", $"dropoffY")).
  count().
  show()
...
+----------------------+--------+
|UDF(dropoffX, dropoffY)|   count|
+----------------------+--------+
|                Queens|  672192|
|                    NA| 7942421|
|              Brooklyn|  715252|
|         Staten Island|    3338|
|             Manhattan|12979047|
|                 Bronx|   67434|
+----------------------+--------+
```

As we expected, the vast majority of trips end in the borough of Manhattan, while relatively few trips end in Staten Island. One surprising observation is the number of trips that end outside of any borough; the number of `NA` records is substantially larger than the number of taxi rides that end in the Bronx. Let's grab some examples of this kind of trip from the data:

```
taxiClean.
  where(boroughUDF($"dropoffX", $"dropoffY") === "NA").
  show()
```

When we print out these records, we see that a substantial fraction of them start and end at the point `(0.0, 0.0)`, indicating that the trip location is missing for these records. We should filter these events out of our data set because they won't help us with our analysis:

```
val taxiDone = taxiClean.where(
  "dropoffX != 0 and dropoffY != 0 and pickupX != 0 and pickupY != 0"
).cache()
```

When we rerun our borough analysis on the `taxiDone` RDD, we see this:

```
taxiDone.
  groupBy(boroughUDF($"dropoffX", $"dropoffY")).
  count().
  show()
```

```
...
+----------------------+--------+
|UDF(dropoffX, dropoffY)|   count|
+----------------------+--------+
|                Queens|  670912|
|                    NA|   62778|
|              Brooklyn|  714659|
|       Staten Island|    3333|
|           Manhattan|12971314|
|                 Bronx|   67333|
+----------------------+--------+
```

Our zero-point filter removed a small number of observations from the output boroughs, but it removed a large fraction of the `NA` entries, leaving a much more reasonable number of observations that had dropoffs outside the city.

Sessionization in Spark

Our goal, from many pages ago, was to investigate the relationship between the borough in which a driver drops his passenger off and the amount of time it takes to acquire another fare. At this point, the `taxiDone` data set contains all of the individual trips for each taxi driver in individual records distributed across different partitions of the data. To compute the length of time between the end of one ride and the start of the next one, we need to aggregate all of the trips from a shift by a single driver into a single record, and then sort the trips within that shift by time. The sort step allows us to compare the dropoff time of one trip to the pickup time of the next trip. This kind of analysis, in which we want to analyze a single entity as it executes a series of events over time, is called *sessionization*, and is commonly performed over web logs to analyze the behavior of the users of a website.

Sessionization can be a very powerful technique for uncovering insights in data and building new data products that can be used to help people make better decisions. For example, Google's spell-correction engine is built on top of the sessions of user activity that Google builds each day from the logged records of every event (searches, clicks, maps visits, etc.) occurring on its web properties. To identify likely spell-correction candidates, Google processes those sessions looking for situations where a user typed a query, didn't click anything, typed a slightly different query a few seconds later, and then clicked a result and didn't come back to Google. Then it counts how often this pattern occurs for any pair of queries. If it occurs frequently enough (e.g., if every time we see the query "untied stats," it's followed a few seconds later by the query "united states"), then we assume that the second query is a spell correction of the first.

This analysis takes advantage of the patterns of human behavior that are represented in the event logs to build a spell-correction engine from data that is more powerful than any engine that could be created from a dictionary. The engine can be used to perform spell correction in any language, and can correct words that might not be included in any

dictionary (e.g., the name of a new startup), and can even correct queries like "untied stats" where none of the words are misspelled! Google uses similar techniques to show recommended and related searches, as well as to decide which queries should return a OneBox result that gives the answer to a query on the search page itself, without requiring that the user click through to a different page. There are OneBoxes for weather, scores from sports games, addresses, and lots of other kinds of queries.

So far, information about the set of events that occurs to each entity is spread out across the RDD's partitions, so, for analysis, we need to place these relevant events next to each other and in chronological order. In the next section, we'll show how to efficiently construct and analyze sessions using advanced functionality that was introduced in Spark 2.0.

Building Sessions: Secondary Sorts in Spark

The naive way to create sessions in Spark is to perform a `groupBy` on the identifier we want to create sessions for and then sort the events postshuffle by a timestamp identifier. If we only have a small number of events for each entity, this approach will work reasonably well. However, because this approach requires all the events for any particular entity to be in memory at the same time, it will not scale as the number of events for each entity gets larger and larger. We need a way of building sessions that does not require all of the events for a particular entity to be held in memory at the same time for sorting.

In MapReduce, we can build sessions by performing a *secondary sort*, where we create a composite key made up of an identifier and a timestamp value, sort all of the records on the composite key, and then use a custom partitioner and grouping function to ensure that all of the records for the same identifier appear in the same output partition. Fortunately, Spark can also support this same secondary sort pattern by combining its `repartition` and a `sortWithinPartitions` transformation; in Spark 2.0, sessionizing a data set can be done in three lines of code:

```
val sessions = taxiDone.
  repartition($"license").
  sortWithinPartitions($"license", $"pickupTime")
```

First, we use the `repartition` method to ensure that all of the `Trip` records that have the same value for the `license` column end up in the same partition. Then, within each of these partitions, we sort the records by their `license` value (so all trips by the same driver appear together) and then by their `pickupTime`, so that the sequence of trips appear in sorted order within the partition. Now when we process the trip records using a method like `mapPartitions`, we can be sure that the trips are ordered in a way that is optimal for sessions analysis. Because this operation triggers a shuffle and a fair bit of computation, and we'll need to use the results more than once, we cache them:

```
sessions.cache()
```

Executing a sessionization pipeline is an expensive operation, and the sessionized data is often useful for many different analysis tasks that we might want to perform. In settings where one might want to pick up on the analysis later or collaborate with other data scientists, it's a good idea to amortize the cost of sessionizing a large data set by only performing the sessionization once, and then writing the sessionized data to HDFS so that it can be used to answer lots of different questions. Performing sessionization once is also a good way to enforce standard rules for session definitions across the entire data science team, which has the same benefits for ensuring apples-to-apples comparisons of results.

At this point, we are ready to analyze our sessions data to see how long it takes for a driver to find his next fare after a dropoff in a particular borough. We will create a `boroughDuration` method that takes two instances of the `Trip` class and computes both the borough of the first trip and the duration in seconds between the dropoff time of the first trip and the pickup time of the second:

```
def boroughDuration(t1: Trip, t2: Trip): (String, Long) = {
  val b = bLookup(t1.dropoffX, t1.dropoffY)
  val d = (t2.pickupTime - t1.dropoffTime) / 1000
  (b, d)
}
```

We want to apply our new function to all sequential pairs of trips inside our `sessions` data set. Although we could write a `for` loop to do this, we can also use the `sliding` method of the Scala Collections API to get the sequential pairs in a more functional way:

```
val boroughDurations: DataFrame =
  sessions.mapPartitions(trips => {
    val iter: Iterator[Seq[Trip]] = trips.sliding(2)
    val viter = iter.
      filter(_.size == 2).
      filter(p => p(0).license == p(1).license)
    viter.map(p => boroughDuration(p(0), p(1)))
  }).toDF("borough", "seconds")
```

The `filter` call on the result of the `sliding` method ensures that we ignore any sessions that contain only a single trip, or any trip pairs that have different values of the `license` field. The result of our `mapPartitions` over the sessions is a data frame of borough/duration pairs that we can now examine. First, we should do a validation check to ensure that most of the durations are nonnegative:

```
boroughDurations.
  selectExpr("floor(seconds / 3600) as hours").
  groupBy("hours").
  count().
  sort("hours").
  show()
...
+-----+--------+
|hours|   count|
+-----+--------+
```

```
|   -3|       2|
|   -2|      16|
|   -1|    4253|
|    0|13359033|
|    1|  347634|
|    2|   76286|
|    3|   24812|
|    4|   10026|
|    5|    4789|
```

Only a few of the records have a negative duration, and when we examine them more closely, there don't seem to be any common patterns to them that we could use to understand the source of the erroneous data. If we exclude these negative duration records from our input data set and look at the average and standard deviation of the pickup times by borough, we see this:

```
boroughDurations.
  where("seconds > 0 AND seconds < 60*60*4").
  groupBy("borough").
  agg(avg("seconds"), stddev("seconds")).
  show()
...
+-------------+------------------+--------------------+
|      borough|      avg(seconds)|stddev_samp(seconds)|
+-------------+------------------+--------------------+
|       Queens|2380.6603554494727|  2206.6572799118035|
|           NA|  2006.53571169866|  1997.0891370324784|
|     Brooklyn| 1365.394576250576|  1612.9921698951398|
|Staten Island|         2723.5625|  2395.7745475546385|
|    Manhattan| 631.8473780726746|   1042.919915477234|
|        Bronx|1975.9209786770646|   1704.006452085683|
+-------------+------------------+--------------------+
```

As we would expect, the data shows that dropoffs in Manhattan have the shortest amount of downtime for drivers, at around 10 minutes. Taxi rides that end in Brooklyn have a downtime of more than twice that, and the relatively few rides that end in Staten Island take a driver an average of almost 45 minutes to get to his next fare.

As the data demonstrates, taxi drivers have a major financial incentive to discriminate among passengers based on their final destination; dropoffs in Staten Island, in particular, involve an extensive amount of downtime for a driver. The NYC Taxi and Limousine Commission has made a major effort over the years to identify this discrimination and has fined drivers who have been caught rejecting passengers because of where they wanted to go. It would be interesting to attempt to examine the data for unusually short taxi rides that could be indicative of a dispute between the driver and the passenger about where the passenger wanted to be dropped off.

Where to Go from Here

Imagine using this same technique on the taxi data to build an application that could recommend the best place for a cab to go after a dropoff based on current traffic patterns and the historical record of next-best locations contained within this data. You could also look at the information from the perspective of someone trying to catch a cab: given the current time, place, and weather data, what is the probability that I will be able to hail a cab from the street within the next five minutes? This sort of information could be incorporated into applications like Google Maps to help travelers decide when to leave and which travel option they should take.

The Esri API is one of a few different tools that can help us interact with geospatial data from JVM-based languages. Another is GeoTrellis, a geospatial library written in Scala, that seeks to be easily accessible from Spark. A third is GeoTools, a Java-based GIS toolkit.