Sessionization

I will demonstrate the use of the complex or nested structures in the example of sessionization. In sessionization, we want to find the behavior of an entity, identified by some ID over a period of time. While the original records may come in any order, we want to summarize the behavior over time to derive trends.

We already analyzed web server logs in Chapter 1, *Exploratory Data Analysis*. We found out how often different web pages are accessed over a period of time. We could dice and slice this information, but without analyzing the sequence of pages visited, it would be hard to understand each individual user interaction with the website. In this chapter, I would like to give this analysis more individual flavor by tracking the user navigation throughout the website. Sessionization is a common tool for website personalization and advertising, IoT tracking, telemetry, and enterprise security, in fact anything to do with entity behavior.

Let's assume the data comes as tuples of three elements (fields `1`, `5`, `11` in the original dataset in Chapter 1, *Exploratory Data Analysis*):

```
(id, timestamp, path)
```

Here, `id` is a unique entity ID, timestamp is an event `timestamp` (in any sortable format: Unix timestamp or an ISO8601 date format), and `path` is some indication of the location on the web server page hierarchy.

For people familiar with SQL, sessionization, or at least a subset of it, is better known as a windowing analytics function:

```
SELECT id, timestamp, path
  ANALYTIC_FUNCTION(path) OVER (PARTITION BY id ORDER BY timestamp) AS
agg
FROM log_table;
```

Here `ANALYTIC_FUNCTION` is some transformation on the sequence of paths for a given `id`. While this approach works for a relatively simple function, such as first, last, lag, average, expressing a complex function over a sequence of paths is usually very convoluted (for example, nPath from Aster Data (https://www.nersc.gov/assets/Uploads/AnalyticsFoundation5.0previewfor4.6.x-Guide.pdf)). Besides, without additional preprocessing and partitioning, these approaches usually result in big data transfers across multiple nodes in a distributed setting.

While in a pure functional approach, one would just have to design a function—or a sequence of function applications—to produce the desired answers from the original set of tuples, I will create two helper objects that will help us to simplify working with the

concept of a user session. As an additional benefit, the new nested structures can be persisted on a disk to speed up getting answers on additional questions.

Let's see how it's done in Spark/Scala using case classes:

```
akozlov@Alexanders-MacBook-Pro$ bin/spark-shell
Welcome to

      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.6.1-SNAPSHOT
      /_/

Using Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_40)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.
SQL context available as sqlContext.

scala> :paste
// Entering paste mode (ctrl-D to finish)

import java.io._

// a basic page view structure
@SerialVersionUID(123L)
case class PageView(ts: String, path: String) extends Serializable with
Ordered[PageView] {
  override def toString: String = {
    s"($ts :$path)"
  }
  def compare(other: PageView) = ts compare other.ts
}

// represent a session
@SerialVersionUID(456L)
case class Session[A  <: PageView](id: String, visits: Seq[A]) extends
Serializable {
  override def toString: String = {
```

```
    val vsts = visits.mkString("[", ",", "]")
    s"($id -> $vsts)"
  }
}^D
// Exiting paste mode, now interpreting.


import java.io._
defined class PageView
defined class Session
```

The first class will represent a single page view with a timestamp, which, in this case, is an ISO8601 `String`, while the second a sequence of page views. Could we do it by encoding both members as a `String` with a object separator? Absolutely, but representing the fields as members of a class gives us nice access semantics, together with offloading some of the work that we need to perform on the compiler, which is always nice.

Let's read the previously described log files and construct the objects:

```
scala> val rdd = sc.textFile("log.csv").map(x => { val z =
x.split(",",3); (z(1), new PageView(z(0), z(2))) } ).groupByKey.map( x =>
{ new Session(x._1, x._2.toSeq.sorted) } ).persist
rdd: org.apache.spark.rdd.RDD[Session] = MapPartitionsRDD[14] at map at
<console>:31


scala> rdd.take(3).foreach(println)
(189.248.74.238 -> [(2015-08-23 23:09:16 :mycompanycom>homepage),(2015-
08-23 23:11:00 :mycompanycom>homepage),(2015-08-23 23:11:02
:mycompanycom>running:slp),(2015-08-23 23:12:01
:mycompanycom>running:slp),(2015-08-23 23:12:03
:mycompanycom>running>stories>2013>04>themycompanyfreestore:cdp),(2015-
08-23 23:12:08
:mycompanycom>running>stories>2013>04>themycompanyfreestore:cdp),(2015-
08-23 23:12:08
:mycompanycom>running>stories>2013>04>themycompanyfreestore:cdp),(2015-
08-23 23:12:42 :mycompanycom>running:slp),(2015-08-23 23:13:25
:mycompanycom>homepage),(2015-08-23 23:14:00
:mycompanycom>homepage),(2015-08-23 23:14:06
:mycompanycom:mobile>mycompany photoid>landing),(2015-08-23 23:14:56
:mycompanycom>men>shoes:segmentedgrid),(2015-08-23 23:15:10
:mycompanycom>homepage)])
(82.166.130.148 -> [(2015-08-23 23:14:27 :mycompanycom>homepage)])
(88.234.248.111 -> [(2015-08-23 22:36:10 :mycompanycom>plus>home),(2015-
08-23 22:36:20 :mycompanycom>plus>home),(2015-08-23 22:36:28
:mycompanycom>plus>home),(2015-08-23 22:36:30
:mycompanycom>plus>onepluspdp>sport band),(2015-08-23 22:36:52
:mycompanycom>onsite search>results found),(2015-08-23 22:37:19
:mycompanycom>plus>onepluspdp>sport band),(2015-08-23 22:37:21
:mycompanycom>plus>home),(2015-08-23 22:37:39
```

```
:mycompanycom>plus>home),(2015-08-23 22:37:43
:mycompanycom>plus>home),(2015-08-23 22:37:46
:mycompanycom>plus>onepluspdp>sport watch),(2015-08-23 22:37:50
:mycompanycom>gear>mycompany+ sportwatch:standardgrid),(2015-08-23
22:38:14 :mycompanycom>homepage),(2015-08-23 22:38:35
:mycompanycom>homepage),(2015-08-23 22:38:37 :mycompanycom>plus>products
landing),(2015-08-23 22:39:01 :mycompanycom>homepage),(2015-08-23
22:39:24 :mycompanycom>homepage),(2015-08-23 22:39:26
:mycompanycom>plus>whatismycompanyfuel)])
```

Bingo! We have an RDD of Sessions, one per each unique IP address. The IP `189.248.74.238` has a session that lasted from `23:09:16` to `23:15:10`, and seemingly ended after browsing for men's shoes. The session for IP `82.166.130.148` contains only one hit. The last session concentrated on sports watch and lasted for over three minutes from `2015-08-23 22:36:10` to `2015-08-23 22:39:26`. Now, we can easily ask questions involving specific navigation path patterns. For example, we want analyze all the sessions that resulted in checkout (the path contains `checkout`) and see the number of hits and the distribution of times after the last hit on homepage:

```
scala> import java.time.ZoneOffset

import java.time.ZoneOffset


scala> import java.time.LocalDateTime

import java.time.LocalDateTime


scala> import java.time.format.DateTimeFormatter

import java.time.format.DateTimeFormatter


scala>

scala> def toEpochSeconds(str: String) : Long = {
LocalDateTime.parse(str, DateTimeFormatter.ofPattern("yyyy-MM-dd
HH:mm:ss")).toEpochSecond(ZoneOffset.UTC) }

toEpochSeconds: (str: String)Long


scala> val checkoutPattern = ".*>checkout.*".r.pattern

checkoutPattern: java.util.regex.Pattern = .*>checkout.*


scala> val lengths = rdd.map(x => { val pths = x.visits.map(y => y.path);
val pchs = pths.indexWhere(checkoutPattern.matcher(_).matches); (x.id,
x.visits.map(y => y.ts).min, x.visits.map(y => y.ts).max,
x.visits.lastIndexWhere(_ match { case PageView(ts,
"mycompanycom>homepage") => true; case _ => false }, pchs), pchs,
x.visits) } ).filter(_._4>0).filter(t => t._5>t._4).map(t => (t._5 -
t._4, toEpochSeconds(t._6(t._5).ts) - toEpochSeconds(t._6(t._4).ts)))
```

```
scala> lengths.toDF("cnt",
"sec").agg(avg($"cnt"),min($"cnt"),max($"cnt"),avg($"sec"),min($"sec"),ma
x($"sec")).show
+-----------------+--------+--------+-----------------+--------+--------
+


|         avg(cnt)|min(cnt)|max(cnt)|
avg(sec)|min(sec)|max(sec)|
+-----------------+--------+--------+-----------------+--------+--------
+
|19.77570093457944|       1|     121|366.06542056074767|      15|
2635|
+-----------------+--------+--------+-----------------+--------+--------
+



scala> lengths.map(x => (x._1,1)).reduceByKey(_+_).sortByKey().collect
res18: Array[(Int, Int)] = Array((1,1), (2,8), (3,2), (5,6), (6,7),
(7,9), (8,10), (9,4), (10,6), (11,4), (12,4), (13,2), (14,3), (15,2),
(17,4), (18,6), (19,1), (20,1), (21,1), (22,2), (26,1), (27,1), (30,2),
(31,2), (35,1), (38,1), (39,2), (41,1), (43,2), (47,1), (48,1), (49,1),
(65,1), (66,1), (73,1), (87,1), (91,1), (103,1), (109,1), (121,1))
```

The sessions last from 1 to 121 hits with a mode at 8 hits and from 15 to 2653 seconds (or about 45 minutes). Why would you be interested in this information? Long sessions might indicate that there was a problem somewhere in the middle of the session: a long delay or non-responsive call. It does not have to be: the person might just have taken a long lunch break or a call to discuss his potential purchase, but there might be something of interest here. At least one should agree that this is an outlier and needs to be carefully analyzed.

Let's talk about persisting this data to the disk. As you've seen, our transformation is written as a long pipeline, so there is nothing in the result that one could not compute from the raw data. This is a functional approach, the data is immutable. Moreover, if there is an error in our processing, let's say I want to change the homepage to some other anchor page, I can always modify the function as opposed to data. You may be content or not with this fact, but there is absolutely no additional piece of information in the result—transformations only increase the disorder and entropy. They might make it more palatable for humans, but this is only because humans are a very inefficient data-processing apparatus.

TIP

Why rearranging the data makes the analysis faster?

Sessionization seems just a simple rearranging of data—we just put the pages that were accessed in sequence together. Yet, in many cases, it makes practical data analysis run 10 to

100 times faster. The reason is data locality. The analysis, like filtering or path matching, most often tends to happen on the pages in one session at a time. Deriving user features requires all page views or interactions of the user to be in one place on disk and memory. This often beats other inefficiencies such as the overhead of encoding/decoding the nested structures as this can happen in local L1/L2 cache as opposed to data transfers from RAM or disk, which are much more expensive in modern multithreaded CPUs. This very much depends on the complexity of the analysis, of course.

There is a reason to persist the new data to the disk, and we can do it with either CSV, Avro, or Parquet format. The reason is that we do not want to reprocess the data if we want to look at them again. The new representation might be more compact and more efficient to retrieve and show to my manager. Really, humans like side effects and, fortunately, Scala/Spark allows you to do this as was described in the previous section.

Well, well, well...will say the people familiar with sessionization. This is only a part of the story. We want to split the path sequence into multiple sessions, run path analysis, compute conditional probabilities for page transitions, and so on. This is exactly where the functional paradigm shines. Write the following function:

```
def splitSession(session: Session[PageView]) : Seq[Session[PageView]] = {
… }
```

Then run the following code:

```
val newRdd = rdd.flatMap(splitSession)
```

Bingo! The result is the session's split. I intentionally left the implementation out; it's the implementation that is user-dependent, not the data, and every analyst might have it's own way to split the sequence of page visits into sessions.

Another use case to apply the function is feature generation for applying machine learning…well, this is already hinting at the side effect: we want to modify the state of the world to make it more personalized and user-friendly. I guess one cannot avoid it after all.