Generadores ScalaCheck para property-based testing de programas Spark y Spark Streaming

Trabajo fin de grado del Grado en Ingeniería Informática
Universidad Complutense de Madrid

Año 2015 - 2016

AUTOR
Max Arnulfo Tello Ortiz

DIRECTORES
Adrián Riesco Rodriguez
Juan Rodríguez Hortalá

DEDICATION

To my parents, for the incessant support and their invaluable teachings.

Autorización de difusión y utilización

Yo, Max Arnulfo Tello Ortiz, autorizo a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor, tanto la presente memoria, el código y/o los contenidos extras desarrollados durante la realización de este proyecto.

Fdo.

Max Arnulfo
Tello Ortiz

Resumen

En los últimos años hemos sido testigos de la expansión del paradigma *big data* a una velocidad vertiginosa. Los cambios en este campo, nos permiten ampliar las áreas a tratar; lo que a su vez implica una mayor complejidad de los sistemas software asociados a estas tareas, como sucede en sistemas de monitorización o en el Internet de las Cosas (Internet of Things). Asimismo, la necesidad de implementar programas cada vez robustos y eficientes, es decir, que permitan el cómputo de datos a mayor velocidad y de los se obtengan información relevante, ahorrando costes y tiempo, ha propiciado la necesidad cada vez mayor de herramientas que permitan evaluar estos programas.

En este contexto, el presente proyecto se centra en extender la herramienta sscheck. Sscheck permite la generación de casos de prueba basados en propiedades de programas escritos en Spark y Spark Streaming. Estos lenguajes forman parte de un mismo marco de código abierto para la computación distribuida en clúster. Dado que las pruebas basadas en propiedades generan datos aleatorios, es difícil reproducir los problemas encontrados en una cierta sesion; por ello, la extensión se centrará en cargar y guardar casos de test en disco mediante el muestreo de datos desde colecciones mayores.

Palabras clave

Spark, testeo basado en propiedades, Scala, generadores, ScalaCheck, big data.

Abstract

In recent years, we have been witness to the speedy expansion of big data. The changes in this field have led us to expand the treatable areas; which also implies more complex software systems associated to these tasks, as it happens in monitoring systems or in the Internet of Things. Likewise, the necessity of implementing reliable and efficient programs, that is, ones that allows us to compute data in a faster pace and to extract valuable information, saving money and time, has led to the increasing need of tools that permit us evaluate these programs.

In this context, the present project centres its mission in extend the sscheck tool. Sscheck is able to generate test cases based in properties of programs wrote in Spark and Spark Streaming. These languages are part of the same open source cluster computing framework. Since the property-based tests generate random data, it's complicated to reproduce the problems found in a given session; therefore, the extension will center in loading and saving test cases from disk through the sampling of data from bigger collections.

Keywords

Spark, property-based testing, Scala, generators, ScalaCheck, big data.

Índice

1. Introducción

> *"El testeo muestra la presencia de errores, no la ausencia."- Edsger W. Dijkstra*

La información es la piedra angular por la que se rige cualquier modelo de negocio actual, la razón, quizá principal, por la cual hemos experimentado un salto exponencial en materia de computación y de almacenamiento [1]. Hemos sido testigos de cómo la forma en la que creamos, captamos y procesamos los datos ha evolucionado a una velocidad vertiginosa. En ese sentido el *big data*, que hace referencia a volúmenes masivos de datos (estructurados o no estructurados) que son difíciles de procesar usando técnicas tradicionales[1], ha supuesto una revolución en cuanto a la manera de enfocar lo que es o no información relevante.

El volumen de datos generado por una organización puede pasar de los megabytes o gigabytes a los terabytes o incluso petabytes en cuestión de meses si la situación así se propicia [2]. Hablando en términos de formas de procesamiento, los datos pueden tratarse en forma de lotes o en tiempo real. La variedad de datos se ha diversificado de simples tablas a ser capaces de extraerla de prácticamente cualquier componente electrónico[2]. Grandes volúmenes de datos esperando ser procesados en el menor tiempo posible empujan a tener herramientas cada vez más potentes y rápidas que permitan cumplir este cometido.

La necesidad cada vez mayor por parte de las empresas de ser capaces de procesar toda esta información ha propiciado que un paradigma totalmente nuevo surja para cubrir esta misión. Tecnologías como Hadoop [3] o Spark [4] se erigen como alternativas para realizar esta labor.

Las compañías exigen que el testeo de programas basados en Hadoop y/o Spark sea cada vez más rigurosos y exactos. Esto supone la necesidad de mejores herramientas que cumplan este cometido. La meticulosidad que se exige conlleva la necesidad de tests cada vez más estrictos y meticulosos, que dependen fundamentalmente de dos factores: la rigurosidad con la que se implementen dichos test y que buscan simular el comportamiento del programa y el tiempo que se esté dispuesto emplear en diseñarlos.

El testeo basado en propiedades [5] (de ahora en adelante TBP) surge como alternativa a los métodos de testeo basado en la ejemplificación, en los que la simulación de entradas y salidas pasa a un segundo plano y nos centramos en la especificación de propiedades expresadas en lógica de orden superior, que nos

[1] Big Data Analytics: Time For New Tools
[2] http://www.wired.com/insights/2014/11/the-internet-of-things-bigger/

permiten especificar una fuente de pruebas automatizada y unas propiedades como parámetros de otra propiedad, que se encarga de, para cada caso de test generado, evaluarlo según la propiedad y comprobar si la refuta o la satisface.

ScalaCheck [6] es una biblioteca para el testeo basado en propiedades para programas implementados en Scala, en la cual se basa sscheck, la herramienta que voy a extender. Dado que tanto ScalaCheck como Spark están implementados en Scala usaremos este lenguaje en el proyecto. Esta biblioteca propone un acercamiento desde la perspectiva descrita en el anterior párrafo. La idea se basa en el "menos es más", dónde podemos especificar propiedades y generadores de datos en escasas líneas, estas propiedades se valen de los generadores para crear datos automatizados con los que podemos evaluar una determinada porción de código. A diferencia de los test de unidad en el que tenemos que simular un comportamiento y evaluar una salida con un aserto que también debemos especificar, limitando el test a tantas salidas como queramos implementar.

El resto de esta memoria se estructura como sigue: la sección 2 presenta los preliminares al desarrollo del proyecto, en el cual se explican entre otros las razones por las que se extiende la herramienta sscheck, los objetivos y el plan de trabajo seguido, la sección 3 presenta las tecnologías usadas y el ecosistema necesario para poder implementar el proyecto, la sección 4 describe el trabajo realizado de principio a fin y un ejemplo que ilustra los logros obtenidos. Por último, la sección 5 recoge las conclusiones finales, una valoración personal del proyecto por parte del alumno y se describen posibles mejoras futuras que sigan en la línea de la extensión hecha a la herramienta sscheck y que le doten de mayor funcionalidad.

1. Introduction

> *"Testing shows the presence of bugs, not the absence."-*
> *Edsger W. Dijkstra*

Information rules every business model nowadays, perhaps the main reason why we have experienced an exponential progress in computation and storage capabilities [1]. We have witnessed how the way we create, collect, and process data has evolved in at vertiginous pace. In this regard, big data, which references to massive volumes of data (structured or not) that are hard to process using traditional techniques,[3] has meant a revolution in the way we see and classify information.

An organization can go from generating megabytes or gigabytes of data to terabytes or even petabytes in some cases [2]. Talking in terms of how we process information, data can be treated in periodic batches or in a continuous stream. The variety of data has diversified from simple datatables to practically being able of extracting information from any electronic device.[4] Huge volumes of data waiting to be processed in the shortest possible time forge the need of having even more reliable and faster tools that allow to reach this mission.

The necessity of more and more companies of being capable to process all this information has led to a new paradigm being born specifically to fulfil this mission. Technologies as Hadoop [3] or Spark [4] raise as the alternatives to fulfil this task.

Organizations demand that the testing for Hadoop-based or Spark-based applications to be more rigorous and exact. This means a necessity of more capable testing tools. This required meticulousness leads to a need of tests even more complex and strict that mainly relies on two factors: the precision in the way these tests are implemented, that looks to simulate the way the program should behave. The second one being the time the testing teams are willing to spend on designing the tests.

Property-based testing [5] (from now on PBT) emerges as an alternative to other testing techniques based on exemplification, where the simulation of inputs and outputs are left behind and property specification expressed in high-order logic takes the lead role. This allows us to specify an automatized source of test cases and a property as parameters of another property. The mission of the high-order property is to evaluate every given test case with property specified as parameter and check whether it holds or not.

[3] Big Data Analytics: Time For New Tools
[4] http://www.wired.com/insights/2014/11/the-internet-of-things-bigger/

ScalaCheck [6] is a property-based testing library written in Scala, in which sscheck is based, the tool I will extend in this project. Given the fact that ScalaCheck and Spark are implemented in Scala, I will be using this language in the project. This library proposes an approach from the perspective described in the previous paragraph. The main idea revolves around the “less is more” philosophy, where one can specify properties and data generators in a few lines of code. These properties use the generators to create automatized test cases with which we can evaluate a certain part of code, reducing considerably the amount of lines of code needed. This differs from unit testing, where we simulate a behavior and evaluate an output with an assert that we must also specify, restricting the test to as much outputs as we want to implement.

The rest of this paper is structured as it follows: Section 2 presents the preliminary notions required to extend our tool, in which it is explained the motivation for this project, the objectives, and the working plan. Followed by Section 3 which describes the technologies used and the ecosystem set to implement the project. Section 4 describes the work done from start to finish and a short example to show what has been accomplished with the extensions. Section 5 concludes this report by describing a personal assessment of the project and the possible future implementations.

2. Preliminares

En esta sección presentamos los conceptos básicos para comprender la presente memoria. Para ello en la sección 2.1. presentamos cómo surgió la herramienta con la que trabajamos y las razones que empujaron a su creación. En la sección 2.2. exponemos las los objetivos y las causas que dan sentido a este proyecto y por último en la sección 2.3. describimos la línea de trabajo que se siguió y cómo se organizó de cara a llevarlo a cabo.

2.1. Antecedentes y motivación

El TBP se fundamenta en la utilización de los tipos de los parámetros de entrada para crear mediante generadores casos de test automatizados. Estos casos de test son evaluados usando las propiedades especificadas por el usuario, de tal manera que comprobamos si el programa cumple esta propiedad; de no cumplirse se trata de encontrar el contraejemplo más simple a partir de la entrada utilizada.

Pero vamos por partes, una propiedad es una regla o ley que hace de caja negra, a la cual le importa los parámetros que recibe y el resultado pero no el comportamiento de lo que intenta evaluar. En ScalaCheck una propiedad sería descrita como se ve en la imagen 2.1. En este ejemplo especificamos que dado un entero *x* y una lista *xs* si concatenamos el valor *x* al principio de la lista, el resultado de aplicar la función *tail* a la concatenación tiene como resultado siempre *xs*. ScalaCheck se encarga de generar automáticamente valores con lo que probar la propiedad.

```
property("list tail") =
  forAll { (x: Int, xs: List[Int]) =>
    (x::xs).tail == xs
  }
```

Imagen 2.1. Propiedad ScalaCheck.

Un generador, por otra parte, en programación funcional se entendería como una estructura que representa cálculos definidos como una secuencia de pasos. En ScalaCheck concretamente, podríamos definirlos como funciones que toman uno o varios parámetros y producen un valor. En la imagen 2.2 podemos ver un generador sencillo que produce una pareja de enteros (n,m).

Para el primer valor elige un valor aleatorio entre 1 y 50, el segundo elemento depende directamente del primero: será un valor entre dicho elemento y su doble.

```
val miGen = for{
  n <- choose(1,50)
  m <- choose(n,n*2)
} yield (n,m)
```

Imagen 2.2. Generador ScalaCheck.

En el siguiente ejemplo se intenta explicar el resultado de combinar propiedades y generadores ScalaCheck de forma que permita hacerse una idea del potencial de esta herramienta. En este ejemplo definimos una propiedad que, dados dos generadores que devuelven un valor entero positivo cada uno, para todo valor *x* e *y,* la suma de ambos será siempre mayor que el primer y que el segundo individualmente (ver imagen 2.3).

```
val p = Prop.forAll(
  Gen.choose(0, Int.MaxValue) :| "x",
  Gen.choose(0, Int.MaxValue) :| "y"
  ) { case (x,y) => (x+y) >= x && (x+y) >=y
}

p.check
```

Imagen 2.3. Propiedad con dos generadores.

Especificada la propiedad pudiera parecer a primer vista que esta se cumple para cualquier caso, sin embargo al ejecutarla (ver imagen 2.4), Scalacheck encuentra un contraejemplo rápidamente, en este caso en el primer intento, aún más interesante, nos muestra el par de valores más sencillos para el cual la propiedad es refutada. Esto se debe a una implementación interna llamada *Shrinking* la cual una vez refutada la propiedad minimiza automáticamente el contraejemplo hasta encontrar una entrada más sencilla para el cual la propiedad sigue sin cumplirse.

```
! Falsified after 0 passed tests.
> x: 124385234
> x_ORIGINAL: 707079750
> y: 2023098414
> y_ORIGINAL: 2049771958
```

Imagen 2.4. Ejecución de la propiedad ScalaCheck.

Bajo esta premisa, los profesores Adrián Riesco y Juan Rodríguez al involucrarse con tecnologías *big data* se dieron cuenta que los programas que desarrollaban eran muchas veces bastante complicados de testear y el escribir test de unidades suponía un excesivo empleo de tiempo y recursos. Por dicha razón surge sscheck [7,8] una herramienta que extiende ScalaCheck y está escrita en Scala para el testeo de programas basado en Spark y Spark Streaming.

Esta biblioteca permite usar una variante de la lógica lineal temporal para especificar propiedades que pueden ser evaluadas por ScalaCheck. En dichas propiedades se busca poner a prueba sistemas de procesamiento de flujo de datos y encontrar posibles brechas que vulneren la integridad de estos programas.

Por ejemplo, para el caso descrito en la imagen 2.5 en el que se especifica una fórmula para Spark Streaming, para 20 lotes de datos, que recibe un flujo de tuplas [número, Booleano], donde los números son identificadores de usuario y el Booleano indica si se puede confiar en esa persona. Queremos que la función cuando encuentre a alguien en quien no se puede confiar, guarde a ese alguien en una lista, por si le vemos en el futuro no confiar en el. La fórmula indica que hasta que no se encuentra a nadie sospechoso nos fiamos de todos y, una vez desconfiamos de alguien, desconfiamos de él para siempre (para más información en fórmulas temporales ver referencia 9).

Sin embargo, el hecho de generar casos de test aleatoriamente es tanto una ventaja como un problema en el TBP. Si nos topamos con un contraejemplo que refuta una propiedad nos enfrentamos a una situación que plantea dos problemas: el primero es que hemos llegado a un contraejemplo al azar, por lo que si volviésemos a testear la función, tal vez no daríamos con el mismo error y nuestro test tendría un resultado favorable, encontrándonos con un falso positivo; la segunda es que está asociada a la primera: un contraejemplo es un caso valioso de test (podría incluso considerarse su inclusión como test de unidad) y en este marco lo estamos perdiendo.

```
def checkExtractBannedUsersList(testSubject : DStream[(UserId, Boolean)] => DStream[User
    val batchSize = 20
    val (headTimeout, tailTimeout, nestedTimeout) = (10, 10, 5)
    val (badId, ids) = (15L, Gen.choose(1L, 50L))
    val goodBatch = BatchGen.ofN(batchSize, ids.map((_, true)))
    val badBatch = goodBatch + BatchGen.ofN(1, (badId, false))
    val gen = BatchGen.until(goodBatch, badBatch, headTimeout) ++
              BatchGen.always(Gen.oneOf(goodBatch, badBatch), tailTimeout)

    type U = (RDD[(UserId, Boolean)], RDD[UserId])
    val (inBatch, outBatch) = ((_ : U)._1, (_ : U)._2)

    val formula : Formula[U] = {
      val badInput : Formula[U] = at(inBatch)(_ should existsRecord(_ == (badId, false))
      val allGoodInputs : Formula[U] = at(inBatch)(_ should foreachRecord(_._2 == true))
      val badIdBanned : Formula[U] = at(outBatch)(_ should existsRecord(_ == badId))

      ( allGoodInputs until badIdBanned on headTimeout ) and
      ( always { badInput ==> (always(badIdBanned) during nestedTimeout) } during tailTi
    }

    forAllDStream(gen)(testSubject)(formula)
  }
```

Imagen 2.5. Ejemplo de fórmula para flujos continuos en Spark Streaming.

La motivación es, pues, encontrar una solución al problema de no poder reproducir las situaciones que ocasionan fallos, desde un acercamiento que nos permita guardar los contraejemplos que se encuentren. Si podemos almacenar estos contraejemplos podremos reutilizarlos en el futuro. De forma que nos permitan hacer cambios en el programa y constatar si una vez arreglados, siguen fallando.

2.2. Objetivos y justificación

El objetivo de este proyecto es el de complementar la herramienta sscheck ampliando su funcionalidad, de tal forma que nos permita especificar generadores ScalaCheck a partir de *schemas* SparkSQL y *schemas* Apache Avro para un determinado flujo de datos, guardando y volviendo a usar los contraejemplos encontrados para posteriormente reutilizarlos y verificar si el programa se ha corregido adecuadamente.

Para ellos planteamos una serie de hitos que, primero, permitan familiarizarnos con las tecnologías aquí utilizadas, ya que en muchos casos algunas tienen pocos años de vida, la documentación existente es relativamente escasa y no

se tocan en ninguna asignatura de la carrera. Lo que supondrá como veremos más adelante, un reto añadido al desarrollo de la herramienta. Segundo, una vez asentados los conocimientos, plantear una situación en la que podamos desarrollar generadores incisivos, los cuales nos permiten especificar una propiedad, un generador Scalacheck y un colección de datos. Ejecutarlos en un test evaluando la colección y ser capaces de guardar los contraejemplos encontrados en un fichero.

La extensión de la herramienta supondrá que una vez finalizados los tests la información persista, podamos corregir posibles errores en los programas evaluados y nos permita simular nuevos tests con los contraejemplos encontrados. Si estos resultan satisfactorios supondrá que hemos podido corregir realmente dichos programas.

2.3. Plan de trabajo

El proyecto se llevó a cabo entre los meses de octubre de 2015 y junio de 2016, bajo la tutela de los profesores Adrián Riesco y Juan Rodríguez Hortalá. Se mantuvo el contacto durante el curso regularmente mediante correo electrónico y reuniones en su despacho, al principio con mucha menor regularidad siendo solo necesarias unas pequeñas directrices para empezar, hasta las reuniones semanales entrado el segundo cuatrimestre para ir apuntalando dudas y cambios.

Se discutió que además de los hitos de implementación necesarios para la realización del proyecto existía la necesidad de desarrollar una fase de de preparación previa que resultó tener una curva de aprendizaje bastante marcada, ya que el lenguaje de programación y el paradigma en el que se desenvuelve Scala (híbrido entre la programación orientada a objetos y la funcional) nunca habían sido dadas por el alumno. Esto sumado a la juventud (y por tanto escasamente documentadas) de las bibliotecas usadas sumaron un rato añadido al proyecto.

Posteriormente a esto y bajo previa reunión con los tutores se establecieron tres hitos de rigurosa implementación para la realización del proyecto. En la siguiente sección paso a describirlos brevemente y con la mera intención de dar una idea general del trabajo hecho. Posteriormente explicaremos detalladamente el trabajo realizado desde un acercamiento más riguroso.

2.3.1. Fases del proyecto

Preparación previa, comprendió los meses entre octubre y febrero, tiempo que el alumno usó para familiarizarse con el lenguaje Scala, valiéndose de libros de la facultad [7] o realizando cursos online, como el dado por Martin Odersky, creador del lenguaje, en Coursera[5]. Posteriormente a esto hubo que familiarizarse con el uso de las bibliotecas ScalaCheck[6] y specs2[7] necesarias para la especificación de los tests, labor bastante compleja debido a la poca documentación existente dado que las herramientas usadas son bastante recientes, lo que en muchos casos dificultó el aprendizaje y la adaptación por parte del alumno.

Por último la configuración del entorno necesario para el desarrollo (sbt, ScalaIDE, gitHub y Travis CI, herramientas que describiremos en la sección 3 de esta memoria) requirió especial dedicación, debido primero a la complejidad a la hora de coordinar las versiones de cada biblioteca para que el proyecto compilara y segundo porque el alumno no había trabajado nunca con un software de integración continua sumado a la gestión del uso de una herramienta para la gestión y construcción de proyectos como es sbt.

Hito 1, apuntalados los conocimientos necesarios se procedió a la implementación del primer hito en el cual implementamos un generador ScalaCheck a partir de la lectura de casos de test serializados en un fichero *avro* (se decidió por este tipo de ficheros como formato de serialización ya que ofrece la posibilidad de almacenar datos junto a su *schema,* es decir, la forma en la que están estructurados los datos). Dicho generador debía devolver los datos leídos desde el fichero hasta su final o en caso contrario, hasta leer la última entrada y seguir devolviendo este último dato hasta la finalización de la prueba. El principal reto de esta fase fue encontrar la forma en la que el generador tuviese estado, de manera que supiéramos cuándo todos los casos de test del fichero habían sido leídos y evitar que el test falle y termine abruptamente.

Hito 2, los resultados del primer hito son usados para obtener generadores scalaCheck desde *batches* Spark con muestreo. La idea principal es la de obtener, a partir de un RDD (unidad básica de abstracción de Spark, explicado con mayor detalle en la sección 3) con un tipo genérico, una muestra de datos con un tamaño especificado por el

[5] https://www.coursetalk.com/providers/coursera/courses/functional-programming-principles-in-scala
[6] https://www.scalacheck.org/documentation.html
[7] https://etorreborre.github.io/specs2/guide/SPECS2-3.8.3/org.specs2.guide.UserGuide.html

usuario y devolver para cada registro contenido un generador scalaCheck del mismo tipo. Indicando también si llegado al final de la muestra se devuelve el último registro leído o si por el contrario se devuelve un error (indicado por un generador fallido). Implementamos esta funcionalidad, primero leyendo desde un fichero *avro* y luego transformando lo leído a un RDD, para posteriormente muestrear en una lista que nos permita extraer registros y devolverlos dentro de un generador.

Hito 3, concluidos los hito 1 y 2, la idea principal es la de implementar generadores incisivos. De forma que al implementar un test, cuando un caso de test no satisfaga una propiedad especificada por el usuario, no perdamos este contraejemplo. Los casos de test los obtenemos desde un generador que devuelve registros extraídos desde un RDD. Siendo el fin el ser capaces de almacenar este contraejemplo en un fichero de manera que este hallazgo persista y posteriormente, una vez el usuario arregle el motivo por lo que la propiedad falló, pueda ser cargado y probado de nuevo. Para ello nos valemos de propiedades de orden superior que nos permiten especificar un generador determinado, una propiedad y una ruta desde la cual cargar los datos y probar para todos la casos si la propiedad se mantiene.

Analizando los hitos, se puede apreciar que el proyecto toca una parte del programa estudiado a lo largo de la carrera. Con el desarrollo de esta herramienta se ponen en práctica conocimientos de programación (FP, TP y EDA) y lógica matemática. Pero también puso a prueba la capacidad del alumno de aprendizaje y adaptabilidad, familiarizándose con la programación funcional, el tratamiento de ficheros serializados, la implementación de tests de unidad y el uso del *framework* Spark.
Dado que el trabajo se desarrolló de manera individual el alumno no valoró la necesidad de elaborar un plan de proyecto propiamente dicho junto a una especificación de requisitos, al no ser una aplicación como tal, se decidió descartar este acercamiento.

3. Tecnologías empleadas

Antes de explicar el desarrollo del proyecto, es relevante dar a conocer las herramientas (software) que se han utilizado para poder llevarlo a cabo. Este proyecto no ha necesitado de ningún soporte hardware especial. Se trabajó sobre un ordenador personal con las tecnologías descritas en este apartado. Valiéndose en muchos casos de bibliotecas experimentales y de corta existencia y que carecían de ningún tipo de soporte técnico al momento de realizarse este proyecto debido a la juventud de la tecnología en la que se basó esta herramienta (la primera versión estable de Spark data del 2014).

3.1. Lenguajes y bibliotecas

A continuación presento cada una de las tecnologías usadas en este proyecto junto con un breve resumen de las mismas y su aportación.

Scala (ver. 2.10.6) es un lenguaje de programación multi-paradigma diseñado para expresar patrones comunes de programación en forma concisa, elegante y con tipos seguros. Integra características de lenguajes funcionales y orientados a objetos. La implementación actual corre en la máquina virtual de Java y es compatible con las aplicaciones Java existentes. Como extra añadido Spark está escrito en Scala, por lo que era conveniente decantarnos por este lenguaje.

ScalaCheck (ver 1.12.2)[8] es una herramienta para testeo de programas escritos en Scala y Java desarrollada por Rickard Nilsson, basado en una biblioteca de Haskell del que también coge su nombre llamada QuickCheck. Permitió definir las propiedades que describen el comportamiento de los flujos de datos y de corroborar que estas se cumpliesen. Todos los datos de prueba se generan de forma automática y de forma aleatoria. Por norma general al ejecutar un test ScalaCheck genera 100 casos distintos aleatorios y si son satisfactorios el test se da por pasado.

Spark - Spark Streaming (ver. 1.6.1.) es un framework de código abierto para la computación en paralelo de clústeres. Originalmente desarrollado por la Universidad de California y luego donado a la fundación Apache. Provee una interfaz para la programación de clústeres con paralelismo implícito de datos y con tolerancia a fallos. Permitió cargar mediante RDDs (acrónimo para *resilient*

[8] https://github.com/rickynils

distributed dataset, su estructura de datos central que representa una colección particionada de elementos inmutables que pueden ser operados en paralelo), *batches* desde local para su posterior tratamiento, análisis y testeo. También fueron usados *DataFrames*, otra tipo de colección distribuida que ofrece Spark dentro de su biblioteca SparkSQL con columnas nombradas.

Specs2[9] (ver. 3.6.4) es una biblioteca que nos permite escribir especificaciones ejecutables basadas en Scala. Integra ScalaCheck, lo que nos permitió tener un pequeño ecosistema que integraba nuestras propiedades, generadores y un ámbito definido en el que se desarrollasen los tests, todo en una sola clase.

Sscheck (ver. 0.2), herramienta escrita en Scala que permite el testeo con scalaCheck para programas Spark y Spark Streaming. Para la implementación de los tests de unidad usa la biblioteca specs2 que a su vez integra perfectamente ScalaCheck.

Travis-CI[10] es un servicio alojado y distribuido de integración continua, que nos permite crear y testear proyectos creados en GitHub. Nos permitió previo acceso a su web llevar el control de los *commits* y *push* que se realizaban en el repositorio y si estas actualizaciones habían permitido que el proyecto se construyese de manera satisfactoria, ejecutando y mostrando los resultados de los tests creados (ver imagenes 3.1 y 3.2).

SBT[11] (ver 0.13), es una herramienta de código abierto para construir proyectos Scala y Java similares a Maven, del cual hablamos brevemente en la sección 4.1, pero dejando de lado su estructura XML. Permitió mediante un plugin para Eclipse crear un archivo especificando las bibliotecas usadas en el proyecto lo que facilitó la portabilidad de este, ya que compilando con sbt desde consola en cualquier ordenador permitía descargarse todas las bibliotecas necesarias para la correcta ejecución del proyecto.

[9] https://etorreborre.github.io/specs2/
[10] https://travis-ci.org/
[11] http://www.scala-sbt.org/

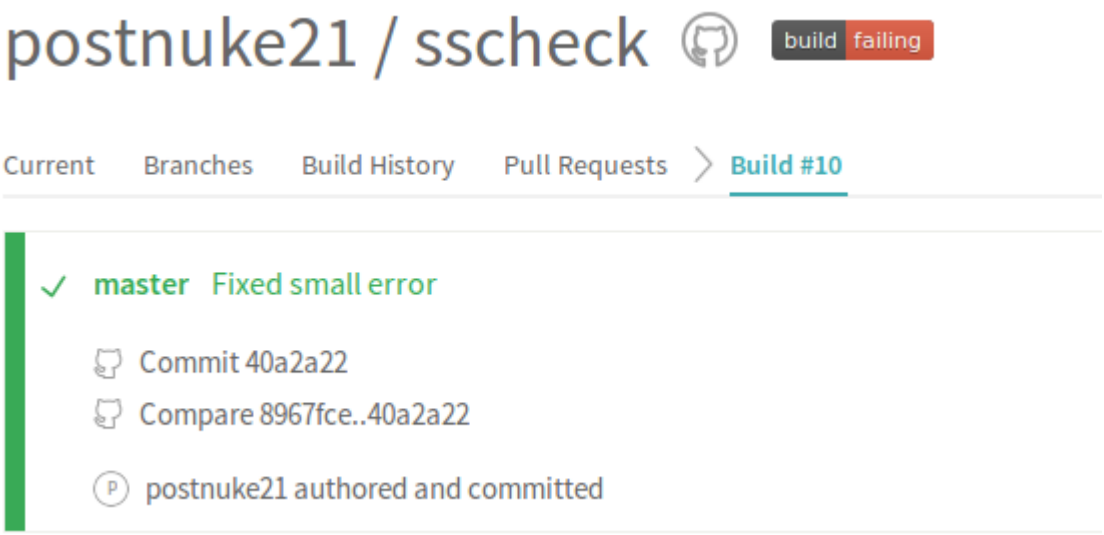


Imagen 3.1. Interfaz web de Travis-CI avisando de una construcción fallida.

```
1430 reg = [DarkTemplar,I strike from the shadows!,1366184681]
1431 reg = [DarkTemplar,I strike from the shadows!,1366184681]
1432 reg = [DarkTemplar,I strike from the shadows!,1366184681]
1433 16/05/21 17:47:32 WARN SharedSparkContext: stopping test Spark context
1434 [info] FromRDDGenTestFromRDDGenTest where
1435 [info]     + prop1
1436 [info] OK, passed 8 tests.
1437 [info]
1438 [info] Total for specification FromRDDGenTest
1439 [info] Finished in 1 second, 416 ms
1440 [info] 1 example, 10 expectations, 0 failure, 0 error
1441 [info]
1442
```

Imagen 3.2. Log con el resultado de los tests ejecutados mostrado por Travis-CI.

Avrohugger[12] **(ver 0.10.1)** es una herramienta que permite la creación de case-classes de Scala a partir de una fichero *avro* que especifica un *schema* determinado. Fue usado en el proyecto como acercamiento a la implementación del hito 1 para la creación de clases dado un fichero *avro* que contenía datos y su respectivo schema embebido.

Scalavro[13] **(ver. 0.6.2)** es una biblioteca que permite, en tiempo de ejecución y basándose en la reflexión, la des/serialización de datos desde Scala hacia un archivo *avro*. Fue usado en el primer hito (ver sección 4.2), como herramienta para la creación de un fichero *avro* con los datos obtenidos en los tests. Se descontinuó su uso ya que, aunque permitía la serialización de datos, no incluía en el fichero el *schema*, por lo que en el siguiente hito se hizo uso de la biblioteca Sparkavro.

Sparkavro[14] **(ver 2.0.1)** es una biblioteca para la lectura y escritura de datos *avro* desde SparkSQL. Usado en el hito 2 para la creación de *dataframes* que nos permitieron a partir de un fichero *avro* tratar los datos en Spark previa conversión a RDDs y así poder realizar muestreos.

Apache Avro[15] **(ver 1.7.7)** es un *framework* desarrollado dentro del proyecto Apache Hadoop. Usa JSON para definir tipos de datos y permite serializar información persistente en binario. Incluye dentro del fichero *avro* que crea la manera en la que se estructuran los datos.

3.2. Entorno de desarrollo

La herramienta sscheck fue desarrollada usando el IDE para Scala de Eclipse por lo que era natural seguir utilizando esta opción. El proyecto está configurado en un ecosistema en el que en el que Scala-IDE era usado para la codificación del proyecto y su ejecución. A su vez este estaba integrado a un repositorio en github.

En lo que se refiere a construir el proyecto esta labor quedó relegada al sbt por lo que el alumno tuvo que familiarizarse con esta herramienta y su uso desde consola. Hubo que aprender a especificar las bibliotecas usadas en el archivo

[12] https://github.com/julianpeeters/avrohugger
[13] https://github.com/GenslerAppsPod/scalavro
[14] https://github.com/databricks/spark-avro
[15] https://avro.apache.org/

build.sbt, este archivo que era incluido en el proyecto, se encargaba de, en el momento de realizar el build, revisar todas las dependencias del proyecto y descargar si procedía todas las nuevas bibliotecas añadidas. Esto suponía que una vez compilado desde Eclipse se podía acceder a todas las funcionalidades de las herramientas incluidas.

Como última pieza del proyecto entró en juego Travis-CI para llevar el control de la integración continua. Este construye el proyecto en su servidor ejecutando el código y los test creados mostrando su respectiva salida y avisando por correo electrónico si había sucedido algún fallo o si por el contrario había tenido éxito la operación.

3.3. Control de versiones

Lo último que queda por nombrar son los controladores de versiones. Para este proyecto dado que 'sscheck' ya estaba alojada en gitHub se procedió a realizar un 'fork' del repositorio original a uno ubicado en 'https://github.com/postnuke21/sscheck'. *Scala IDE for Eclipse* implementa GitHub dentro de su interfaz, lo que facilitó la gestión de los *commits* y la subida de código al repositorio.

4. Herramienta

Abordamos en esta sección el trabajo de extensión propiamente dicho de la herramienta 'sscheck'. En la sección 4.1 explicaremos la estructura del proyecto y su distribución y en las secciones 4.2 a 4.5 explicaremos detalladamente el proceso de desarrollo hito por hito y los resultados obtenidos.

4.1. Estructura del proyecto

Antes de presentar la implementación de la herramienta se debe profundizar en la estructura utilizada en el proyecto. Como se indicó en la sección 3.3 (Entorno de trabajo), la implementación de la aplicación se ha llevado a cabo mediante el uso del entorno de desarrollo Scala IDE, ya que ofrece un medio de trabajo muy completo para el desarrollador. Aunque el proyecto usa sbt para construirse, sigue el estándar de disposición de directorios especificado por Apache Maven[16], que establece una series de normas para la organización de un proyecto y la distribución de las diferentes carpetas lo que facilita a los desarrolladores trabajar con una jerarquía única (ver imagen 4.1).

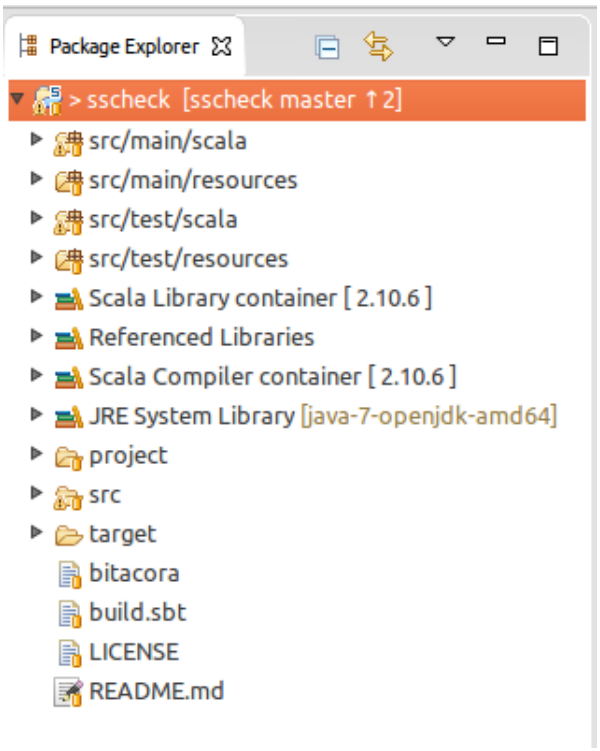


Imagen 4.1. Distribución del proyecto.

[16] https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

Dentro del proyecto como partes más relevantes podemos distinguir las siguientes secciones:

- src/main/scala, donde ubicamos todas las clases necesarias para la construcción del proyecto como tal.

- src/main/resources, donde se encuentran los recursos necesarios para la correcta construcción de la herramienta.

- src/test/scala, donde ubicamos todas las clases test que nos hagan falta, por normal general se llaman igual que sus análogas en el directorio main más el sufijo -Test.

- src/test/resources, donde ubicamos los recursos necesarios para ejecutar los test. Como pueden ser los ficheros con los datos a cargar por los generadores.

- project, carpeta donde se almacenan las rutas a las clases dentro del proyecto, es decir la jerarquía de clases. También es donde se ubica el fichero plugins.sbt (ver imagen 4.2). En nuestro caso fue necesario incluir el plugin de Eclipse para sbt[17], el cual automatiza la creación de especificaciones de proyectos.

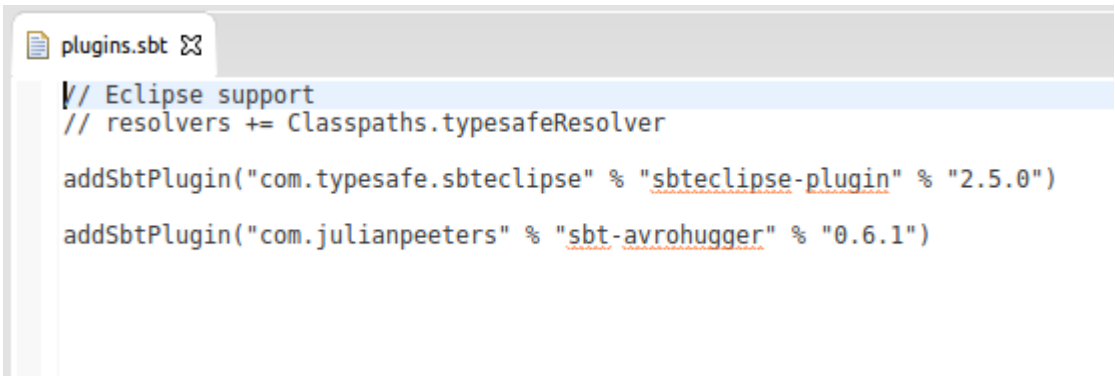


Imagen 4.2. Plugins usados en el proyecto.

- build.sbt, fichero donde se especifica las bibliotecas y versiones utilizadas dentro del proyecto (ver imagen 4.3). Esto me permitía, tras compilar mediante sbt, poder acceder a todas la funcionalidades de las bibliotecas

[17] https://github.com/typesafehub/sbteclipse/

especificadas desde Scala IDE. La inclusión de este fichero facilitaba el trabajo de desarrollo ya que mediante su uso se evita el tener que agregar manualmente cada una de las bibliotecas.

build.sbt

```
import com.typesafe.sbteclipse.plugin.EclipsePlugin.EclipseKeys._

name := "sscheck-max2"

version := "1.0"

scalaVersion := "2.10.5"

crossScalaVersions  := Seq("2.10.5")

lazy val sparkVersion =  "1.6.1" // "1.4.1"

lazy val specs2Version = "3.6.4"

// Use `sbt doc` to generate scaladoc, more on chapter 14.8 of "Scala Cookbook"

// show all the warnings: http://stackoverflow.com/questions/9415962/how-to-see-all-the-w
scalacOptions ++= Seq("-feature", "-unchecked", "-deprecation")

// if parallel test execution is not disabled and several test suites using
// SparkContext (even through SharedSparkContext) are running then tests fail randomly
parallelExecution := false

// Could be interesting at some point
// resourceDirectory in Compile := baseDirectory.value / "main/resources"
// resourceDirectory in Test := baseDirectory.value / "main/resources"

// Configure sbt to add the resources path to the eclipse project http://stackoverflow.co
// This is critical so log4j.properties is found by eclipse
EclipseKeys.createSrc := EclipseCreateSrc.Default + EclipseCreateSrc.Resource
```

Imagen 4.3. Muestra de algunas de las bibliotecas usadas.

4.2. Hito 1

La idea general de este hito gira en torno a implementar un generador ScalaCheck desde un fichero en local que contiene casos de tests.

Como ya mencionamos en la descripción del plan de proyecto (ver sección 2.3.1), nos decantamos por el uso de Apache Avro como mecanismo de serialización de los datos debido a que, primero, ofrece un rico modelo de datos valiéndose de JSON para definir los tipos y, segundo, porque permite la serialización compacta a binario facilitando el transporte entre nodos de Hadoop o Spark. Además es soportado por SparkSQL, lo que posibilita una lectura casi inmediata desde ficheros *avro* a RDDs.

Dado que la biblioteca de Apache Avro para Scala no implementa herramientas nativas para Scala sino que usa las de Java se decidió por un acercamiento en dos partes. La creación y lectura de datos *avro* se realizó con la biblioteca Scalavro, esta herramienta permite serializar datos a binario pero carece la posibilidad de leer *schemas* adjuntos en el mismo fichero. Por lo que fue necesario el uso de otra librería llamada 'avrohugger', que dado un *schema* creado (de extensión .avsc) genera en tiempo de compilación una clase en el proyecto con los tipos especificados en el fichero (ver imágenes 4.4 y 4.5 que simulan el *schema* de un *tweet* y su correspondiente clase).

Imagen 4.4. Especificación del *schema.*

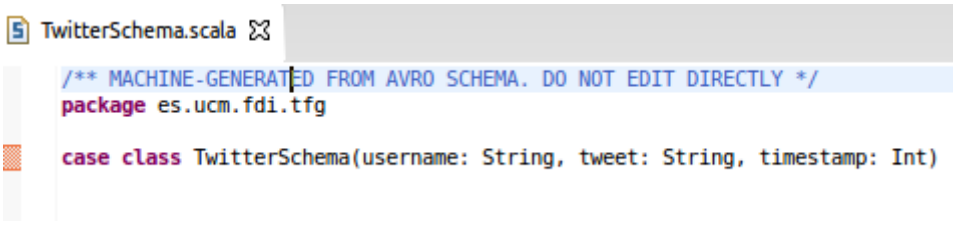


Imagen 4.5. Clase generada automáticamente.

Solventado esto nos enfrentamos al principal problema en este hito, los generadores ScalaCheck no son *stateful*, es decir, carecen de estado y son *sealed traits*, por lo que no se pueden extender, además necesitamos mantener un cursor al siguiente registro que será usado para generar el siguiente caso de test.

Para solucionarlo implementamos otra clase en nuestro caso **FromFileGen** que recibe como parámetros una ruta desde la cual cargar los casos de test y un valor Booleano **defaultToLast** que hará de estado y que determina, una vez alcanzado el último caso de test a generar, si seguir devolviendo este último hasta el final del test o por el contrario devolver un fallo. Para resolver el problema del generador no extensible implementamos una conversión implícita desde **FromFileGen** a **Gen**, usando un objeto compañero en la misma clase y llamando a una función propia de los generadores llamada **Gen.wrap()** que nos permite devolver en forma de generador el caso de test leído. En el caso que nos topemos con el final del fichero los generadores Scalacheck nos permiten devolver un **Gen.Fail** a modo de fallo y que implica que el test ejecutado falle también.

Para probar esta implementación se creó una clase de test, en la cual mediante el uso de la biblioteca specs2 que implementa ScalaCheck definimos un ámbito en el cual proporcionar la ruta al fichero de prueba (que contiene 5 entradas) y definir el valor de la variable **defaultToLast**.

Cuando **defaultToLast** toma el valor *true*, el resultado esperado es que si ya no quedan más casos de test por leer se devuelva el último leído hasta que se satisfaga el test. Para un mínimo de test pasados igual a **10,** el resultado es el indicado en las imágenes 4.6 y 4.7 donde después de devolver el quinto registro, este se devuelve otras cuatro veces.

```
reg = TwitterSchema(human1,hola mundo,1366154481)
reg = TwitterSchema(human2,adios mundo,1366154482)
reg = TwitterSchema(human3,tweet de human3,1366154483)
reg = TwitterSchema(human4,tweet de human4,1366154484)
reg = TwitterSchema(human5,tweet de human5,1366154485)
reg = TwitterSchema(human5,tweet de human5,1366154485)
reg = TwitterSchema(human5,tweet de human5,1366154485)
reg = TwitterSchema(human5,tweet de human5,1366154485)
reg = TwitterSchema(human5,tweet de human5,1366154485)
reg = TwitterSchema(human5,tweet de human5,1366154485)
```

Imagen 4.6. Salida por consola de los casos de test leídos.

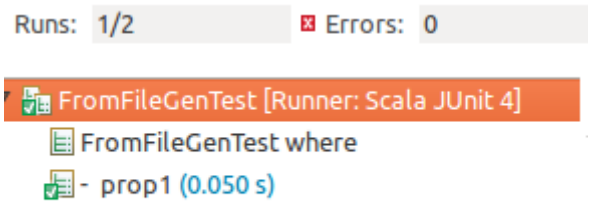


Imagen 4.7. Test superado exitosamente.

Para **defaultToLast** igual a *false* y todos los demás parámetros conservando el mismo valor los resultados se pueden apreciar en las imágenes 4.8 y 4.9 donde después de devolver el quinto registro y no habiendo más que leer, se devuelve un **Gen.Fail**, el test se interrumpe y se da por fallido.

```
reg = TwitterSchema(human1,hola mundo,1366154481)
reg = TwitterSchema(human2,adios mundo,1366154482)
reg = TwitterSchema(human3,tweet de human3,1366154483)
reg = TwitterSchema(human4,tweet de human4,1366154484)
reg = TwitterSchema(human5,tweet de human5,1366154485)
```

Imagen 4.8. Al intentar probar el sexto caso el test falla.

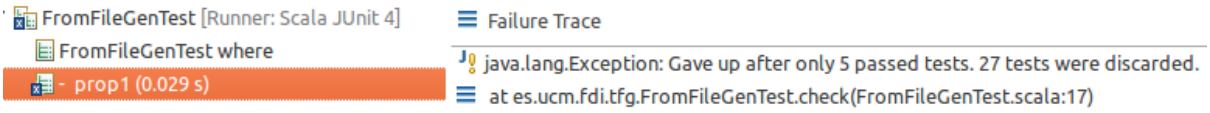


Imagen 4.9. Resultado del test.

4.3. Hito 2

Como segundo hito, y valiéndonos de los resultados obtenidos en el primero, nos centramos en implementar un generador de orden superior ScalaCheck que trabaje con lotes de Spark, es decir que dada una función que recibe un **RDD[A]**, esta devuelve un **Gen[A]**, siendo A un tipo genérico.

Los RDD pueden alcanzar tamaños intratables, especialmente si estamos trabajando en un entorno local, por lo que un generador de casos de test que extrajese datos directamente sería un acercamiento erróneo. Se decidió implementar un primer planteamiento con una clase **FromRDDGen** que se basa en la estructura implementada en el primer hito, pero que además recibe un parámetro entero **bufferSize**.

La idea es, dado un valor entero, utilizar la función **RDD.takeSample()** que implementan los RDDs y que nos permite realizar un sampleo de ese tamaño y guardar esos casos de test en una lista concurrente. Un RDD en ningún caso se modifica ya que es un objeto inmutable, es decir, una vez creado, los elementos que almacena no pueden ser alterados. Una vez hemos obtenido la muestra, implementamos una función que dada una lista de casos de test, los va extrayendo uno por uno y devolviendo envuelto en un **Gen[A]**. Análogamente a lo realizado en el primer hito implementamos también una variable **defaultToLast** que determina si una vez llegado al último caso de test es extraído y devuelto o simplemente devuelto n veces hasta satisfacer el test.

Probamos este acercamiento definiendo un test con specs2. Primero, para poder utilizar Spark hace falta tener un contexto configurado, la herramienta sscheck define ya uno, este contexto define el punto de entrada para todas las funcionalidades de Spark, con el cual podemos especificar la conexión a un clúster y poder crear RDDs. En el test especificamos la ruta que almacena los casos de tests, creamos el RDD y definimos todos los parámetros necesarios para la clase **FromRDDGen** (ver imagen 4.10). Dado que tratamos con ficheros *avro*, es necesario leerlos de manera especial. Para poder leerlos Spark nos ofrece una biblioteca llamada SparkSQL, que nos facilita la lectura desde este tipo de ficheros a *dataFrames* conservando su *schema*. Un *dataFrame* organiza sus datos en columnas y nos permite la conversión a RDD de manera inmediata.

Para **defaultToLast** igual a *true*, un muestreo de 3 casos de test y un mínimo de test pasados igual a **5** el comportamiento esperado es el indicado en las imágenes 4.11 y 4.12 donde, en la primera imagen especificamos los parámetros necesarios para la carga de los casos de test y procedemos a la creación del RDD a partir del fichero *avro* que contiene 10 entradas. En la

segunda imagen se puede observar cómo devuelto el 3 registro se procede a devolver este mismo otras dos veces, se da por bueno el test y el contexto de Spark se detiene automáticamente.

```
/**
*Using SparkSQL to load the data into a  DataFrame , once we have it, we convert it to RDD[A]
*/
val filePath = "src/test/resources/twitter.avro"
val sqlContext = new SQLContext(impSC)
val rdd = sqlContext.read.avro(filePath).rdd
val defaultToLast = false
```

Imagen 4.10. Parámetros de la clase **FromRDDGen.**

```
reg = [DarkTemplar,I strike from the shadows!,1366184681]
reg = [Immortal,I return to serve!,1366175681]
reg = [VoidRay,Prismatic core online!,1366160000]
reg = [VoidRay,Prismatic core online!,1366160000]
reg = [VoidRay,Prismatic core online!,1366160000]

16/06/03 16:10:35 WARN SharedSparkContext: stopping test Spark context
```

Imagen 4.11. Resultado de la ejecución del test para **FromRDDGen.**

Aunque este planteamiento pueda parecer adecuado y hayamos logrado implementar un generador arbitrario, es decir, un generador que en lugar de devolver datos aleatorios devuelve datos a partir de unos casos de test extraídos de un RDD y que conocemos. Surgen algunos problemas a tener en cuenta: si buscamos evaluar estos generadores con futuras propiedades scalaCheck nos veremos en la obligación de realizar muchos muestreos, incluso tal vez del total de los elementos del RDD. La función **RDD.takeSample()**, usada para muestrear los casos nos permite, modificando un parámetro Booleano que recibe llamado **withReplacement**, especificar si queremos que las muestras no se repitan. Pero nos vemos en la misma situación ya que aunque la muestra no contenga casos repetidos el RDD sigue siendo el mismo ya que es inmutable por lo que nos podemos topar con un test que al evaluar una propiedad nos dé un falso positivo.

Para solventar el problema de tener siempre la misma fuente de muestreo realizamos algunos cambios a la clase anterior y creamos otra llamada **FromRDDReplacementGen,** que recibe los mismos parámetros pero en este caso el RDD consiste en una tupla [caso de test, id único]. Esta clase nos permite, en sucesivas muestras crear, un RDD nuevo solo con casos de test no seleccionados. Este RDD es asignado a una variable **rdd** y no a un valor ya que

así podemos reasignar la variable una vez hayamos muestreados los casos de test a un nuevo RDD filtrado. Hacemos esto, primero, porque los valores [val] en Scala son lo que, a una variable **final** supone en Java, una vez asignados no se pueden cambiar y segundo porque un RDD es inmutable y no se puede modificar por lo que debemos crear uno nuevo (ver imagen 4.12).
Esta operación tiene un coste constante ya que un RDD en sí no contiene los datos propiamente dichos sino que almacena los metadatos necesarios para poder tratarlos en memoria y a menos que realizamos un **RDD.collect()** la herramienta no se ve en la necesidad de re-computar todos los datos cada vez que se haga un muestreo lo que supondría que la operación tuviese un coste lineal.

```
/**Collects the ids from the RDD using broadcast and filters the rdd
  *with these values, leaving the rdd only with those registers not sampled
  *It was necessary to extend the Serializable class otherwise a Spark
  *exception was being encountered
 */
if (!withRepl) {
  var ids = impSC.broadcast(sampled.map(_._2))
  rdd=rdd.filter(v => !ids.value.contains(v._2))
}
```

Imagen 4.12. Filtrado del RDD de la clase **FromRDDReplacementGen.**

Una vez modificada esta clase, el test que se implementa con specs2 lee el fichero *avro* con los casos de tests desde la ruta especificada y crea el rdd con las tuplas. Para esto se vale de la función **RDD.zipWithUniqueId()**. Si tenemos un fichero con 10 casos de test , especificamos como 10 también la cantidad de tests pasados y un muestreo de 5, el resultado esperado es que todos los casos de tests serán evaluados. Como se muestra en la imagen 4.11 donde después de devolver los primero 5 registros se avisa que el buffer está vacío y se procede a un nuevo muestreo, extrayendo los registros restantes del RDD sin que ninguno se repita. Una vez finalizado el test se detiene el contexto Spark y se da el test por pasado.

```
[Stage 1:============================>                            (1 + 1) / 2]
                                                                          reg =
reg = [VoidRay,There is no greater void than the one between your ears.,1366176300]
reg = [DarkTemplar,I am the blade of Shakuras!,1366174681]
reg = [Immortal,En Taro Adun!,1366176283]
reg = [VoidRay,Prismatic core online!,1366160000]
Buffer depleted , generating a new batch of samples
reg = [DarkTemplar,I strike from the shadows!,1366184681]
reg = [miguno,Rock: Nerf paper, scissors is fine.,1366150681]
reg = [BlizzardCS,Works as intended.  Terran is IMBA.,1366154481]
reg = [VoidRay,Fire at will, commander.,1366160010]
reg = [Immortal,I return to serve!,1366175681]

16/06/03 17:27:44 WARN SharedSparkContext: stopping test Spark context
```

Imagen 4.13. Muestreo con filtrado.

4.4. Hito 3

Alcanzados los objetivos establecidos en el primer y segundo hito, en este último nos centramos en implementar una propiedad de orden superior, es decir, que reciba otra propiedad y un casos de test como parámetros y que sea capaz de almacenar el caso de test que resulte ser un contraejemplo.

Para ello, creamos una clase **IncisiveProp** y nos centramos en crear una función **forAll[A]** (ver imagen 4.14) que recibe un generador de tipo genérico **Gen[A]** y una ruta con casos de test. La idea es que dada la ruta podamos cargar los casos de test usando los resultados del anterior hito a una variable **mixedGen.** Primero utilizamos los datos que proporciona el **Gen[A]** y una vez que se acaben pasamos a usar los casos de test cargados desde fichero. El objetivo es poder alternar entre un generador que tengamos con casos de test y otros que tengamos guardados en disco.

```
def forAll[A](gl: Gen[A], filePath: String)(f: A => Prop): Prop
```

Imagen 4.14. Función forAll de orden superior.

Una vez hayamos resuelto la fuente de casos de test, creamos un valor de Scala de tipo propiedad llamado *prop* y dado un **mixedGen,** evaluamos el caso de test con una propiedad *x.* Esta propiedad es especificada en el test de prueba mediante el uso de la función **Prop** de ScalaCheck. El resultado, obtenido como una función parcial es resuelta por f, que finalmente lo devuelve como un tipo **Property** (ver imagen 4.15).

```
val prop = Prop.forAll(mixedGen)(f)
```

Imagen 4.15. Scalacheck Prop de orden superior que evalúa el caso de test.

Para poder guardar el contraejemplo en caso de que algún caso de test incumpla la propiedad, redefinimos la función **Prop** de ScalaCheck de forma que, además de evaluar esta propiedad, pueda guardar el contraejemplo que no la satisface. Esto lo logramos valiéndonos del estado de **Result**: si la propiedad no se cumple tomará el valor **Prop.False**, la implementación se muestra como indica la figura 4.16, en la que dado un *if* que distingue casos según su valor, si este resulta ser **Prop.False** llamamos a una función **writeToFile** que guarda el contraejemplo en un fichero.

```
if ((res.status == Prop.False) || (res.status.isInstanceOf[Prop.Exception])) {
  // NOTE using res.args(0) because there is just one argument here
  // NOTE we can safely cast to A because we know the test case has been generated with gen: Gen[A]
  val counterexample: A = res.args(0).origArg.asInstanceOf[A]
  println(s"\tCOUNTEREXAMPLE was: ${counterexample}")
  writeToFile(counterexample, counterExamplePath)
}
res
```

Imagen 4.16. Extracto de la redefinición de Prop, para el guardado del contraejemplo.

Posteriormente se implementaron también sobrecargas de la función **forAll** de manera que pueda aceptar de 2 a 4 generadores y de 2 a 4 rutas de ficheros respectivamente. En cada una de estas situaciones tendríamos tuplas de generadores de casos de test y un contraejemplo se consideraría como tal si una propiedad x no se mantiene para una tupla *y.* Para la imagen 4.17 con 2 generadores y 2 rutas de fichero, se resuelve recursivamente primero la función **forAll** para 1 generador y 1 ruta. El resultado obtenido se evalúa mediante una función parcial con el segundo generador y la segunda ruta.

```
/**
 * This function evaluates for 2 generators and only returns a counterexample if it finds a value
 * that rejects a property based in the tuple
 */
def forAll[A, B](g1: Gen[A], g2: Gen[B], filePath1: String, filePath2: String)
                (f: (A, B) => Prop): Prop = forAll(g1, filePath1)(t => forAll(g2, filePath2)(f(t, _: B)))
...
```

Imagen 4.17. Función forAll para dos generadores y dos rutas.

Finalizada la implementación del generador incisivo, procedemos a crear una clase **IncisivePropTest.** En este test probamos mediante un generador sencillo **Gen.oneOf()**, que nos permite especificar manualmente unos valores como parámetros eligiendo uno aleatoriamente. Definimos dos casos para dos funciones **forAll()** la primera *p2* con 2 generadores y otra *p1* con 1.

Como se puede observar en la imagen 4.18 para dos valores enteros que pueden ser positivos o negativos, elegidos por los generadores, especificamos una propiedad que dado un valor *v*, el producto de *x* e *y* debe ser mayor que *v* para que se satisfaga. Análogamente para *p3,* una propiedad que compara dos *strings.* Especificamos un generador sencillo entre dos palabras. La propiedad se evalúa a *true* o *false* si las palabras coinciden. El valor generado se devuelve almacenado en *sentence* y si es igual a “adios” se satisface la propiedad.

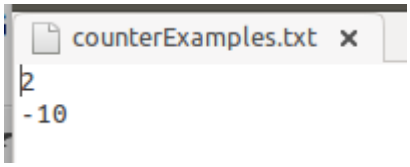
```
  def p2(v: Int) = incisiveProp.forAll(Gen.oneOf(-5, -10), Gen.oneOf(2, -3), "path1.avro", "path2.avro") { (x, y) =>
  println(s"(for v = $v)")
  println(s"x : $x & y : $y")

  x * y must be_>(v)

}.set(minTestsOk = 1).verbose


def p3 = incisiveProp.forAll(Gen.oneOf("hola", "adios"), filePath) { sentence =>
  println(s"sentence: $sentence")
   sentence == "adios"
}.set(minTestsOk = 1).verbose
```

4.18. Implementación de los dos test de prueba.

El resultado de la ejecución se observa en la imagen 4.19. Para *p2,* definimos dos ejecuciones, una con *v* igual a -7 y otra con *v* igual a -3. Como especificamos un mínimo de 1 test pasado, para v igual a -7 la propiedad se cumple con la tupla [-5, -3] y la propiedad se evalúa a *true*. Sin embargo, para *v* igual a -3 y la tupla [2,-10] la propiedad no se cumple, por lo que el resultado es *false*. Esta tupla nos sirve de contraejemplo por lo que la guardamos en el fichero “counterExamples.txt” (ver imagen 4.20). En ambos test para simplicidad del ejemplo establecemos que no hay casos de test que cargar desde disco.

```
No remaining samples left switching to given Gen
(for v = -7)
x : -5 & y : -3
        res Result(True,List(Arg(,-3,0,-3,org.scalacheck.util.Pretty$$anon$1@2dd822c5,org.scalach
        res Result(True,List(Arg(,-5,0,-5,org.scalacheck.util.Pretty$$anon$1@70f1c44,org.scalache
No remaining samples left switching to given Gen
No remaining samples left switching to given Gen
(for v = -4)
x : -10 & y : 2
        res Result(False,List(Arg(,2,0,2,org.scalacheck.util.Pretty$$anon$1@5473eccd,org.scalache
        COUNTEREXAMPLE was: 2
        res Result(False,List(Arg(,-10,0,-10,org.scalacheck.util.Pretty$$anon$1@5fb89726,org.scal
        COUNTEREXAMPLE was: -10
No remaining samples left switching to given Gen
sentence: adios
        res Result(True,List(Arg(,adios,0,adios,org.scalacheck.util.Pretty$$anon$1@728906ab,org.s
16/06/10 15:34:46 WARN SharedSparkContext: stopping test Spark context
```

4.19. Resultado de la ejecución de los tests.



4.20. Fichero donde se guardan los contraejemplos.

4.5. Ejemplo

En esta sección mostramos con un sencillo ejemplo lo alcanzado con la implementación del generador incisivo. Para ello, seguimos la línea del ejemplo usado en la sección 2.1 para describir sscheck y su funcionamiento.

Tenemos un fichero avro con 50 entradas y su respectivo *schema*. Cada una de las entradas alberga una tupla [userID, trustable] que determina si un usuario es de fiar o no, en este caso vamos a definir una propiedad que evalúe la consistencia de los nombres usados por cada usuario. Definimos una propiedad *p1* que dada un generador y una ruta a un fichero avro, evalúa los casos de test cargados a un RDD como se muestra en la imagen 4.21. La propiedad dicta que dado un userID si transformamos todos sus caracteres a mayúscula y luego a minúscula el resultado debería ser igual si directamente cambiamos todos a minúscula.

```
  def p1 = incisiveProp.forAll(rddGen, filePath) { row =>
    println(s"Row: $row")
    row.get(0).toString().toUpperCase().toLowerCase()==row.get(0).toString().toLowerCase()
}.set(minTestsOk = 30).verbose
```

Imagen 4.21. Implementación de la propiedad p1.

Para un mínimo de 30 test pasados y realizando muestreos del RDD de 5 casos de test por vez, el resultado es el que se muestra en la imagen 4.22. Después de 11 ejemplos evaluados nos encontramos con el nombre de un usuario que tiene un carácter Unicode que comparte con muchos otros el mismo carácter en mayúscula. Esto supone que la propiedad no se mantenga, el test falle y se detenga el contexto Spark no sin antes guardar el contraejemplo en el fichero.

```
Row: [BriggsKirk,true]
        res Result(True,List(Arg(,[BriggsKirk,true],0,[BriggsKirk,true],org.scalacheck.util.Pretty
Row: [BeulahKelly,true]
        res Result(True,List(Arg(,[BeulahKelly,true],0,[BeulahKelly,true],org.scalacheck.util.Pret
Row: [LittleIrwin,false]
        res Result(True,List(Arg(,[LittleIrwin,false],0,[LittleIrwin,false],org.scalacheck.util.Pr
Row: [LetitiaNewton,true]
        res Result(True,List(Arg(,[LetitiaNewton,true],0,[LetitiaNewton,true],org.scalacheck.util.
Row: [KittyTownsend,true]
        res Result(True,List(Arg(,[KittyTownsend,true],0,[KittyTownsend,true],org.scalacheck.util.
Buffer depleted , generating a new batch of samples
Row: [FernSharp,false]
        res Result(True,List(Arg(,[FernSharp,false],0,[FernSharp,false],org.scalacheck.util.Pretty
Row: [LucyGilmộ̊,false]
        res Result(False,List(Arg(,[LucyGilmộ̊,false],0,[LucyGilmộ̊,false],org.scalacheck.util.Prett
        COUNTEREXAMPLE was: [LucyGilmộ̊,false]

16/06/10 18:46:15 WARN SharedSparkContext: stopping test Spark context
```

Imagen 4.22 Resultado de la ejecución del test.

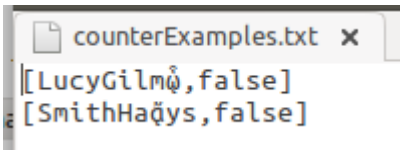


Imagen 4.23 Fichero que almacena los contraejemplos.

5. Conclusiones

En esta sección concluyo la presente memoria añadiendo una valoración personal sobre el proyecto en el primer punto y el posible trabajo futuro que se podría realizar a partir de la extensión de la herramienta sscheck en el segundo punto.

5.1. Valoración personal

Partiendo de los requisitos iniciales, que consistían en desarrollar una extensión de la herramienta sscheck que permitiese el testeo de programas Spark y Spark Streaming con la carga de casos de test desde ficheros avro y el guardado de contraejemplos en un fichero persistente, el proyecto ha cumplido los requisitos básicos establecidos al inicio del proyecto.

Sin embargo, la funcionalidad del proyecto tiene algunas limitaciones, ya que por simplicidad se decidió guardar los datos en un fichero simple en local y debido a la curva de aprendizaje, que resultó ser más pronunciada de la esperada por el alumno, hubo que suprimir ciertas implementaciones: no se pudo probar en un entorno de flujo continuo de datos que simulase el uso de Spark Streaming. Todos los ejemplos realizados en este proyecto se basaron en la creación de lotes de datos desde Spark Core. Tampoco se pudo probar en un entorno real con datos reales y que pudiesen poner a prueba la utilidad de la herramienta.

Como punto final, creo que este proyecto supone para mí un buen inicio de cara a introducirme de lleno en el mundo *big data*. He tenido la oportunidad de trabajar con un *framework* que cada vez está teniendo más aceptación por parte de muchas empresas, dado que, en ciertos entornos, es mucho más rápido que el entorno usado de facto, Hadoop[18]. Además, el haber tenido que aprender un lenguaje como Scala, sin ninguna noción de programación funcional, supuso poner a prueba los conocimientos adquiridos durante la carrera y también ver mi capacidad de adaptación y aprendizaje. Por último, el proyecto significó una primera toma de contacto con herramientas de testeo, creo que es algo a tener en cuenta, ya que como tal no se nos enseña en ninguna asignatura del grado y considero que es un valor añadido para mi currículo y de cara a enfrentarme al mundo laboral.

[18] http://www.datastax.com/dev/blog/how-much-faster-is-spark-than-hadoop-in-datastax-enterprise

5.2. Trabajo futuro

La realización de este proyecto abre las puertas al desarrollo de una herramienta mucho más completa y compleja.

Posibles ampliaciones del proyecto serían que el generador incisivo alternase con cierto determinismo la prueba de casos de test cargados desde fichero con el generador pasado por parámetro pero por falta de tiempo no se pudo realizar. Esto supondría otorgar más aleatoriedad a los tests y que pudiésemos alternar entre distintas fuentes con mayor frecuencia.

Ampliar la lectura casos de test a RDD desde ficheros JSON o CSV (valores separados por comas). Esta extensión otorgaría mucha más versatilidad a la herramienta, ya que muchos ficheros obtenidos desde otras fuentes fuera del marco Spark puede que no ofrezcan una forma de estructurar datos con tipos definidos.

Por último, también sería interesante pulir la implementación de forma que pase a ser parte de la siguiente versión de sscheck. Para esta mejora será también necesario generar la documentación necesaria en forma de Scaladoc para que pueda ser utilizada por otros desarrolladores y publicarla.

5. Conclusions

This section concludes the project memory by, first, presenting a personal assessment about the work done and, second, describing future extensions for the sscheck tool.

5.3. Personal assessment

Focusing in the basic requirements, which consisted in developing an extension to the sscheck tool that could let us do property-based testing on Spark-based programs by loading test cases from avro files and saving the resulting counterexamples in a local file, the project has met the expected results.

Nevertheless, the tool has some limitations. For simplicity it was decided to store the counterexamples found in a local file, which was not serialized. The learning curve was harder than expected by the student so it was necessary to suppress some implementations: the extension could not be tested in a continuous data stream environment that could simulate a Spark Streaming usage. All the tests were done in Spark Core and it was not possible to use real data to evaluate what the tool was capable of.

Finally, I consider this project draws a good start for me as an introduction to the big data ecosystem. I had the chance to work with a framework, Spark, that is being widely accepted by more and more companies migrating to big data, in fact, it shows to be faster than Hadoop in certain environments.[19]

Moreover, having the opportunity of learning a language such as Scala without previous notions in functional programming put to test my abilities and all that I had learn in the faculty, as well as my adaptation capabilities and my ability to learn new concepts. Lastly, the fact that I had my initial contact with testing tools is a valuable experience, because it is something that is not being taught in the computer science degree and I consider it will be important in my career.

[19] http://www.datastax.com/dev/blog/how-much-faster-is-spark-than-hadoop-in-datastax-enterprise

5.4. Future work

The realization of this project opens a path for a larger and more complex tool development.

Possible implementations would be an incisive generator intermingling deterministically the test cases given by the generator specified as parameter and the ones loaded from the local file. It was impossible to implement it due to the time constraints. This could give more randomness to the tests switching between test case sources more frequently.

It would also be interesting to extend to JSON or CSV (comma-separated values) the file extensions accepted by the RDD, hence giving more versatility to the tool. This is because the test case files could be generated from various sources outside the Spark framework and they might not offer a rich data structure with defined types.

We would also like to polish the implementation so it could be part of the next sscheck release. Finally, we are also interested on generating the Scaladoc documentation and publishing it, so it can be used by other developers.

6. Bibliografía

[1] http://navint.com/images/Big.Data.pdf : *Why is BIG Data so important.* Navint. 2012

[2] Needham, Jeffrey. *Disruptive possibilities: how big data changes everything*. "O'Reilly Media, Inc." 2013.

[3] T. White. *Hadoop: The definitive guide*. "O'Reilly Media, Inc." 2012.

[4] H. Karau, A. Konwinski, P. Wendell and M. Zaharia. *Learning Spark: Lightning-Fast Big Data Analysis*. O'Reilly Media, 2015.

[5] Claessen, Koen, and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices* 46.4 (2011): 53-64.

[6] R. Nilsson. *Scalacheck: The Definitive Guide*. IT Pro. Artima Incorporated, 2014.

[7] https://github.com/juanrh/sscheck : *Repositorio de la herramienta sscheck*.

[8] A. Riesco and J.Rodriguez-Hortalá. *Temporal Random Testing for Spark Streaming*. In E. Abraham and M. Huisman, editors, Proceedings of the *12th International Conference on integrated Formal Methods (iFM 2016)*.

[9] Blackburn, Patrick, Johan FAK van Benthem, and Frank Wolter. *Handbook of modal logic*. Elsevier, 2006.

[10] Odersky, Martin, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008.