Generadores escalables en Apache Flink: implementación y aplicaciones en testing y pruebas de carga
Scalable Generators in Apache Flink: Implementation and Application in Testing and Load Functions

Trabajo de Fin de Máster
Curso 2019–2020

Autor
Antonio Barral Gago

Director
Enrique Martín Martín

Colaborador
Juan Rodríguez Hortalá

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Generadores escalables en Apache Flink: implementación y aplicaciones en testing y pruebas de carga
Scalable Generators in Apache Flink: Implementation and Application in Testing and Load Functions

Trabajo de Fin de Máster en Ingeniería Informática
Departamento de Sistemas Informáticos y Computación

Autor
Antonio Barral Gago

Director
Enrique Martín Martín

Colaborador
Juan Rodríguez Hortalá

Convocatoria: *Julio 2020*
Calificación: 10 - Sobresaliente

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

21 de julio de 2020

Agradecimientos

A mis tutores Enrique y Juan por su constante apoyo semana tras semana, reunión tras reunión. A toda mi familia por su interés en mi progreso, su apoyo y su paciencia conmigo. A mi otra familia en Madrid por cuidarme todos los días, por su cariño y su complicidad.

Resumen

Generadores escalables en Apache Flink: implementación y aplicaciones en testing y pruebas de carga

Debido al auge del procesamiento masivo de datos y al surgimiento de motores capaces de realizar esta tarea, se ha visto la necesidad de cambiar la manera en la que se testean las aplicaciones que se usan para dichos motores. La comprobación mediante test unitarios es insuficiente para poder probar la funcionalidad de un programa que manipula millones de datos, sin tener que dedicar una gran cantidad de tiempo para implementar dichos test. Existe una metodología llamada *Property-based testing* (PBT), derivada del *Random testing* que genera entradas de datos de manera aleatoria, pudiendo así crear test genéricos. Actualmente, se podría considerar que algunos de los motores más usados son Apache Spark y Apache Flink, pudiendo ambos procesar tanto flujos continuos de datos, como datos en lotes. Apache Spark ya cuenta con aplicaciones basadas en PBT para ambos tipos de procesamiento, mientras que Apache Flink solo tiene bibliotecas para usar PBT en flujos de datos. Así pues, el objetivo principal de este proyecto es desarrollar una herramienta que permita usar PBT en grandes cantidades de datos procesadas en lotes, teniendo en cuenta que dicha herramienta debe de implementar algunas de las características principales de los motores de procesamiento masivo de datos (escalabilidad, distribución y tolerancia a fallos).

Palabras clave

Apache Flink, ScalaCheck, Testeo basado en propiedades, Tolerancia a fallos, Escalabilidad, Specs2, Generadores, DataSet, Scala.

Abstract

Scalable Generators in Apache Flink: Implementation and Application in Testing and Load Functions

Due to the massive data processing boom and the rise of engines capable of executing this task, there is a need of changing how applications have been tested in these engines. Verification using unitary tests is insufficient to test the functionality of a program that deal with millions of data, without spending too much time in the process. There exists a methodology, born from *Random testing*, called *Property-based testing* (PBT) which allows the programmer to create random input data, so that generic tests can be created. Nowadays, some of the most relevant engines are Apache Spark and Apache Flink, being both of them capable of doing stream data processing and batch data processing. Apache Spark has libraries based on PBT for both types of processing, whereas Flink only has libraries for stream processing. Therefore the main purpose of this project is to create a tool that allows the implementation of PBT in programs that process a large amount of data in batches, considering some of the main features of massive data processing engines (scalability, distribution, and fault-tolerance).

Keywords

Apache Flink, ScalaCheck, Property-based testing, Fault-tolerance, Scalable, Specs2, Generators, DataSet, Scala.

Índice

1. Introducción **1**
1.1. Descripción . . . 1
1.2. Objetivos . . . 2
1.3. Plan de trabajo . . . 2
1.4. Organización de la memoria . . . 6

2. Preliminares **9**
2.1. *Testing* . . . 9
2.1.1. Test unitarios . . . 9
2.1.2. *Property-based testing* . . . 10
2.2. Sistemas distribuidos . . . 11
2.3. Apache Flink . . . 12
2.3.1. Características . . . 12
2.3.2. Arquitectura de Flink . . . 13
2.3.3. Apache Spark. Diferencias con Flink . . . 14
2.4. Scala . . . 16
2.4.1. SBT . . . 17
2.4.2. ScalaCheck . . . 17
2.4.3. Specs2 . . . 19
2.4.4. ScalaMeter . . . 20
2.5. *Testing* en motores de procesamiento de datos . . . 20
2.5.1. Apache Hadoop . . . 20
2.5.2. Apache Spark . . . 21
2.5.3. Apache Flink . . . 22

3. Generadores de `DataSet` y `Table` **25**
3.1. Estructura general . . . 25
3.2. Escalabilidad . . . 27
3.3. Tolerancia a fallos . . . 28
3.4. Ejemplo de ejecución mediante *cluster standalone* de Apache Flink . . . 30
3.5. Generadores de `Table` . . . 32
3.5.1. Estructura general . . . 32

4. *Property-based testing* usando generadores **35**
4.1. *WordCount* . . . 35
4.2. TPCH10 . . . 37
4.3. *K-Means* . . . 39
4.4. ETL . . . 41
4.5. Propiedades para el generador de `Table` . . . 44
4.6. Resultados . . . 45

5. Implementación y uso de objetos `Matcher` de flink-check **47**
5.1. `Matcher` proporcionados por flink-check . . . 47
5.2. `Matcher` implementados . . . 50
5.3. Resultados . . . 52

6. Elaboración de funciones de carga usando ScalaMeter **53**
6.1. Estructura de las funciones de carga . . . 53
6.2. Eficiencia sobre generadores . . . 55
6.2.1. *SmallGeneratorBenchmark* . . . 56
6.2.2. *AverageGeneratorBenchmark* . . . 57
6.2.3. *LargeGeneratorBenchmark* . . . 58
6.2.4. *HugeGeneratorBenchmark* . . . 58
6.3. Eficiencia de programas Flink . . . 59
6.3.1. Función lineal . . . 60
6.3.2. Función cuadrática . . . 61

7. Conclusiones **65**
7.1. Dificultades encontradas . . . 65
7.2. Trabajo futuro . . . 66

8. Introduction **69**
8.1. Description . . . 69
8.2. Objectives . . . 70
8.3. Working plan . . . 70
8.4. Paper structure . . . 74

9. Conclusions **75**
9.1. Difficulties . . . 75
9.2. Future work . . . 76

Bibliografía **77**

A. Manual de usuario **81**
A.1. Instalación y uso de Intellij + SBT . . . 81

Índice de figuras

1.1. Uso de Trello. Los colores definen la prioridad de la tarea 3

2.1. Ejemplo de testeo unitario usando JUnit en Scala 10
2.2. Ejemplo de test PBT usando ScalaCheck 11
2.3. Interacciones entre componentes 13
2.4. Comparación del flujo de datos lógico con el flujo de datos físico . . . 15
2.5. Página principal del panel de Apache Flink 16
2.6. Ejemplo de uso `Prop.forAll` 18
2.7. Uso de pre-condiciones en una propiedad 18
2.8. Generador personalizado de una clase Cliente 19
2.9. Testeo unitario en Apache Flink 23

3.1. Función de creación de generador para DataSet 26
3.2. Ejemplo de uso de creación de generadores ScalaCheck 27
3.3. Flujo del trabajo enviado al cluster, mostrado por el panel de Flink . 30
3.4. Información general sobre la repartición de la tarea 31
3.5. Información sobre cada tarea desglosada 31
3.6. Información sobre el reparto de las tareas entre los *TaskManagers* . . 32
3.7. Función de creación de generador para Table 32

4.1. Definición de propiedad básica en *WordCount* 36
4.2. Consulta SQL TPCH10 utilizada para testear 38
4.3. Propiedad TPCH10 39
4.4. Elaboración de la propiedad para *K-Means* 41
4.5. Validación de la herramienta ETL 43
4.6. Lanzamiento de excepción usando el Matcher proporcionado por specs2 43
4.7. Programa *WordCount* para la API Table de Flink 44
4.8. Propiedad para el programa *WordCount* implementado para API Table de Flink 45

5.1. Matcher para comprobar si un DataSet es subconjunto de otro 48
5.2. Función de resta entre conjuntos, aplicada a DataSet 49
5.3. Matcher para comprobar si dos DataSet son iguales 50

5.4. Propiedad para comprobar el funcionamiento de beSubDataSetOf y beEqualDatasetTo 51

6.1. Información mostrada en la interfaz web creada por ScalaMeter . . . 55
6.2. Gráfica con funciones de carga con un rango de elementos pequeño . . 56
6.3. Gráfica con funciones de carga con un rango de elementos medio . . . 57
6.4. Gráfica con funciones de carga con un rango de elementos medio . . . 59
6.5. Gráfica con funciones de carga con un rango de elementos medio . . . 60
6.6. Código para crear la función lineal y la función cuadrática 61
6.7. Gráfica con funciones de carga de complejidad lineal 62
6.8. Gráfica con funciones de carga de complejidad cuadrática 63

8.1. How Trello was used. Colours define the priority if every task 71

A.1. Versiones de Scala JDK y SBT usadas 82
A.2. Estructura del proyecto en Intellij 82
A.3. Ejemplo de ejecución de WordCountTestSpecs 82

Índice de tablas

1.1. Planificación del desarrollo del proyecto 6

2.1. Comparativa entre Apache Flink y Apache Spark 17

8.1. Project work plan 74

Capítulo 1

Introducción

En este capítulo se presentará el contexto del proyecto y una motivación. A continuación, se expondrán los objetivos principales enumerándolos por hitos. El último apartado tratará sobre la planificación en la que se explica el seguimiento que se ha realizado del proyecto, exponiendo los diferentes hitos completados.

1.1. Descripción

Hoy en día, la cantidad de información que se genera y manipula es inmensa, creciendo el volumen de la misma de manera exponencial en los últimos años. Esto se debe a la constante mejora del hardware y de las infraestructuras de los sistemas informáticos. El resultado se puede ver actualmente en algoritmos que emplean grandes empresas para conocer los gustos de sus clientes (desde Netflix recomendando sus series, hasta Google ubicando publicidad en webs de productos, en función de una conversación de una persona con otra).

Para poder tratar un volumen ingente de datos, ya no son suficientes las herramientas que se han utilizado hasta hace unos años para almacenar y manejar datos como las bases de datos relacionales. Ahora es necesario usar motores de procesamiento de datos, pensados para poder realizar dicha tarea de la manera mas eficiente posible. Es importante remarcar que estos motores se diseñan para utilizarse de manera distribuida, es decir, varios servidores están conectados entre sí para poder llevar a cabo una tarea de la manera más eficiente posible.

El trabajo distribuido implica una mayor complejidad a la hora diseñar y programar una tarea, lo que conlleva a una mayor facilidad de equivocación por parte del programador. Otro problema añadido, es la dificultad de encontrar un error en el programa debido a que se ejecuta de manera distribuida. Debido a esta problemática, es fundamental realizar un testeo exhaustivo que pueda reproducir un error que aparezca en un programa.

Apache Flink es uno de estos motores, relativamente nuevo en comparación con sus competidores (Apache Spark, Apache Hadoop) pero ya utilizado por empresas conocidas como Amazon, King (creadores de Candy Crush) o Alibaba entre otros listados en [1]. Es capaz tanto de procesar flujos de datos en tiempo real como

conjuntos de datos separados en lotes. Actualmente ya existen herramientas para testear los flujos de datos creados por Apache Flink, como por ejemplo la biblioteca flink-check [2]. Como se explicará en la sección 2.5.3, flink-check es una biblioteca que implementa PBT usando fórmulas de lógica temporal.

La idea de este trabajo es crear una herramienta que permita crear grandes conjuntos de datos mediante la funcionalidad de tratamiento de datos por lotes de Apache Flink. Los datos creados se utilizarán para poder testear tareas en este motor mediante el uso de un tipo de testeo que se llama *Property Based Testing* (PBT a partir de ahora) y que se explicará en detalle en esta memoria (ver sección 2.1.2).

La herramienta debe de ser distribuida, escalable y, para una mayor facilidad a la hora de reproducir test fallidos, tolerante a fallos.

1.2. Objetivos

A raíz del resumen del proyecto y su introducción, se definen los siguientes objetivos:

1. Crear una herramienta capaz de crear generadores de conjuntos de datos en Apache Flink que sean capaces de actuar de manera:
   a) **distribuida**, ofreciendo así la posibilidad que se pueda ejecutar en un *cluster*, como propone la arquitectura de Apache Flink. Este punto también implica que los generadores tienen que ser serializables, ya que es necesario el paso de información en tiempo de ejecución entre los diferentes trabajadores que esten realizando la tarea.
   b) **escalable**, para que si el número de elementos a generar crece, la carga de trabajo se distribuya equitativamente.
   c) **tolerante a fallos parciales** mediante reintentos, consiguiendo de esta manera reproducir el mismo generador en el caso de que se caiga alguno de los servidores que esté calculando su parte del generador.
2. Utilizar la herramienta para probar diferentes programas Apache Flink utilizando el paradigma PBT.
3. Implementar generadores del tipo `Table` de Apache Flink.
4. Implementar varios `Matcher` para poder comparar objetos `DataSet` de manera flexible y eficiente.
5. Basado en los generadores, desarrollar una biblioteca para realizar pruebas de carga sobre programas Apache Flink y mostrar los resultados de manera gráfica.

1.3. Plan de trabajo

Para llevar a cabo los objetivos citados, se ha realizado un plan de trabajo que consta de reuniones cada, aproximadamente, 15 días con los tutores del proyecto.

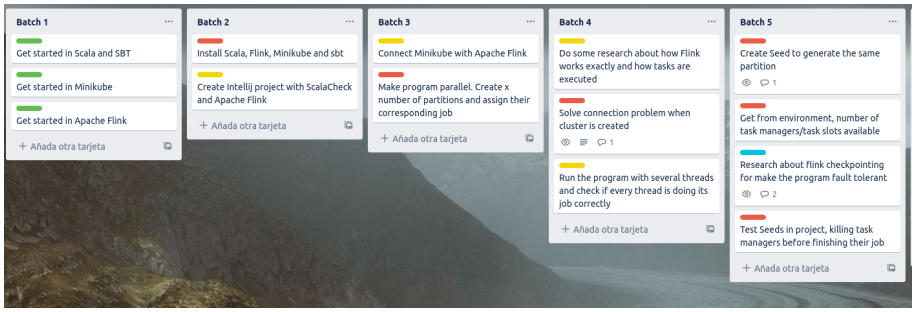


Figura 1.1: Uso de Trello. Los colores definen la prioridad de la tarea

En dichas reuniones se revisaban las tareas propuestas para los 15 días entre la reunión anterior y la actual, y se proponían nuevas tareas en función de los objetivos conseguidos. En caso de que algunas de las tareas realizadas necesitasen una revisión, ya fuese porque se encontraron errores o porque en la reunión se proponían ideas mejores, se volvían a proponer como tareas para la siguiente reunión. De esta manera se ha simulado de la manera más precisa posible una metodología kanban [3].

Los objetivos previamente mencionados, se agruparon en los siguientes hitos sobre los que, posteriormente, se desglosaron los objetivos y las diferentes tareas que se definieron en cada reunión. Dichos hitos son:

1. Investigación sobre las tecnologías principales a utilizar para el proyecto (Scala, Apache Flink, ScalaCheck).

2. Implementación de una herramienta de generadores distribuidos de `DataSet`, escalables y tolerantes a fallos en Apache Flink.

3. Uso de generadores en PBT con ScalaCheck para testear su funcionalidad y enseñar posibles usos para los mismos.

4. Incremento de la funcionalidad de la herramienta añadiendo un generador para el tipo `Table` de Apache Flink.

5. Realización de funciones de carga usando el *framework* ScalaMeter, para comprobar el comportamiento de la herramienta para crear generadores de diferentes cantidades de elementos, usando un número variable de particiones.

Para realizar este seguimiento de manera mas sencilla, se han utilizado las siguientes herramientas:

- Correo electrónico como contacto principal entre tutores y alumno.

- Google Meets para realizar reuniones cuando no se podía realizar este acto de manera presencial.

- Trello como herramienta para realizar seguimiento del trabajo entre reunión y reunión mediante la implementación de un tablón, simulando el de kanban, como se puede ver en la figura 1.1.

En la tabla 1.1 se describe de manera más detallada los diferentes hitos realizados durante el desarrollo del proyecto. En cada hito se muestran las diferentes tareas finalizadas. Las fechas proporcionadas son estimaciones.

Iteraciones	**Tareas**
Investigación sobre herramientas principales (1/1/20 a 15/1/20)	▪ Aprendizaje de Scala y SBT. ▪ Aprendizaje de Apache Flink.
Creación de entorno de trabajo y continuación de investigación (16/1/20 a 30/1/20)	▪ Instalación de Flink, Scala y SBT en Intellij. ▪ Aprendizaje de ScalaCheck.
Implementación de la funcionalidad básica del generador (1/2/20 a 15/2/20)	▪ Creación de generador de `DataSet` dado un generador de elementos, un número de elementos y el número de particiones que se quieren utilizar. ▪ Simulación del uso del generador en un *cluster* simulado mediante el *standalone cluster* de Apache Flink para comprobar su escalabilidad.
Incorporación de semillas al generador para añadir la tolerancia a fallos (16/2/20 a 29/2/20)	▪ Integración del uso de una semilla en el generador para generar siempre el mismo conjunto de datos, aunque falle Flink en el proceso. ▪ Integración de specs2 al proyecto. ▪ Realización de un test para comprobar la funcionalidad de la semilla, consistiendo este en escribir los datos en ficheros temporales. ▪ Primera refactorización del código.

Creación de test usando ScalaCheck sobre ejemplos de Apache Flink (1/3/20 a 15/4/20)	▪ Implementación del código *WordCount* de Apache Flink. ▪ Creación de test mediante ScalaCheck ejecutando el código de *WordCount* usando datos creados con el generador de `DataSet`. ▪ Implementación del código TPCH10 de Apache Flink. ▪ Creación de test mediante ScalaCheck ejecutando el código de TPCH10 usando datos creados con el generador de `DataSet`. ▪ Implementación de test simulando una herramienta ETL mediante el uso de generadores de `DataSet`. ▪ Implementación del código KMeans de Apache Flink. ▪ Creación de test mediante ScalaCheck ejecutando el código de KMeans usando datos creados con el generador de `DataSet`.
Extensión de la herramienta de generación de `DataSet` para soportar objetos `Table` (22/3/20 a 7/4/20)	▪ Segunda refactorización del código. ▪ Implementación de un generador de `Table` mediante el uso del generador de `DataSet`. ▪ Integración de test de comparación entre `DataSet` generados usando semillas. ▪ Extensión de los test realizados para el código *WordCount* con una implementación para el generador de `DataSet`.

Adaptación e implementación de varios `Matcher` de la biblioteca flink-check (8/4/20 a 21/4/20)	▪ Actualización de los `Matcher` de la biblioteca a las nuevas versiones de Scala y Apache Flink. ▪ Implementación de un `Matcher` para comprobar si dos `DataSet` son exactamente iguales. ▪ Integración de un `Matcher` que compare `DataSet` incluyendo duplicados (estricto) o sin incluirlos.
Implementación de funciones de carga mediante el uso de ScalaMeter (22/4/20 a 7/5/20)	▪ Aprendizaje de la biblioteca ScalaMeter. ▪ Implementación de una función que devuelve una propiedad de ScalaCheck sobre la que ejecutar el *benchmarking*. ▪ Implementación de varias pruebas con diferente número de elementos generados y número de particiones que crean el generador.
Redacción de la memoria y optimización del código (8/5/20 a 21/5/20)	▪ Tercera y última refactorización del código. ▪ Integración de comentarios siguiendo el estándar de Scaladoc en el código del proyecto. ▪ Redacción de la memoria.

Tabla 1.1: Planificación del desarrollo del proyecto

1.4. Organización de la memoria

Esta memoria está dividida en varios capítulos, tratando cada uno de ellos un tema en específico. Los capítulos son los siguientes:

- **Preliminares**: se explican las diferentes herramientas, bibliotecas y metodologías aplicadas para este proyecto.
- **Generadores de `Dataset` y `Table`**: se explica el desarrollo básico de la funcionalidad de la herramienta, que son las funciones para crear generadores de `Dataset` y `Table`.
- ***Property-based testing* usando generadores**: se explica la aplicación de estos generadores en propiedades PBT usando diferentes programas de Apache

Flink.

- **Implementación y uso de objetos `Matcher` de flink-check**: se explica la implementación de dichos objetos con el propósito de comparar de manera más eficiente objetos `DataSet`.
- **Elaboración de funciones de carga usando ScalaMeter**: se muestran varias funciones en las que se muestra el comportamiento y eficiencia de los generadores tanto de manera individual, como en programas Flink, en función del número de elementos que se generen.
- **Conclusiones**: se exponen las conclusiones junto con las principales dificultades encontradas durante el proyecto y posible trabajo futuro para expandir la herramienta.

Todo el código implementado se encuentra en el siguiente repositorio de GitHub: `https://github.com/AntonioBarral/TFM-flink-dataset-generators`.

Capítulo 2

Preliminares

En este capítulo se expondrán los conceptos necesarios para poder comprender el proyecto, junto con las herramientas que se han utilizado en el mismo. Por último se hablará del estado del arte. En el estado del arte, se mencionará brevemente otros entornos de cómputo masivo de datos como Apache Spark y Apache Hadoop, además de herramientas y técnicas de testeo, tanto en dichos entornos, como en Apache Flink.

2.1. *Testing*

El testeo es la actividad que busca evaluar una sección o función de un programa de software, para poder obtener el resultado deseado de dicho código, de la manera más eficiente posible. El testeo es un proceso que va más haya de la depuración, ya que el objetivo del testeo, no solo es el de encontrar errores de código, sino también el de verificar y validar las operaciones realizadas en la porción de código y su resultado [4].

Existe una gran variedad de algoritmos y métodos para realizar testeo sobre una funcionalidad en un programa. En este proyecto se ha utilizado principalmente un tipo de testeo automático, llamado testeo basado en propiedades o *Property-based testing* (PBT).

Para explicar PBT antes se va a introducir uno de los métodos más usados (y más enseñados en universidades), que es el test unitario o *Unit testing* y la razón por la que es preferible usar PBT antes que un test unitario.

2.1.1. Test unitarios

Un test unitario es aquel que ejecuta una pequeña porción de código (ya sea una clase o una función) con unos valores de entrada predeterminados y comprueba si la salida que produce la ejecución del código es la deseada o no [5, p. 18].

Uno de los *frameworks* más conocidos para programar test unitarios es JUnit [6]. Principalmente, Junit es un *framework* que funciona en Java, aunque también tiene soporte para Scala, ya que este lenguaje es compatible con Java y Junit utiliza la

```
1  import org.junit.Test
2  import org.junit.Assert._
3
4  class JunitTest {
5
6    @Test def testMax() {
7      val z = Math.max(1,3)
8      assertEquals(3, z)
9    }
10
11   @Test def testMaxTwo() {
12     val z = Math.max(1,0)
13     assertEquals(1, z)
14   }
15
16   @Test def testMaxThree() {
17     val z = Math.max(20,20)
18     assertEquals(10, z)
19   }
20
21   @Test def testMaxFour() {
22     val z = Math.max(-35,0)
23     assertEquals(0, z)
24   }
25 }
```

Figura 2.1: Ejemplo de testeo unitario usando JUnit en Scala

máquina virtual de Java o *Java Virtual Machine* (JVM a partir de ahora).

El código en la figura 2.1 muestra un ejemplo de cómo testear de manera unitaria la función *Math.max* de Scala, adaptando el ejemplo de [5, p. 18]. Como se puede observar, es necesario indicar varios ejemplos manualmente (en este caso, uno por método) lo cual produce que se generen muchas lineas de código para poder testear la función *Math.max*. En caso de necesitar evaluar funcionalidades más complejas que la expuesta en la figura 2.1 mediante el uso del testeo unitario, puede convertir el testeo en una tarea tediosa. Además, dependiendo la complejidad del código, se pueden generar ficheros con una gran cantidad de líneas de código.

2.1.2. *Property-based testing*

Al contrario que con los test unitarios, PBT utiliza propiedades para representar un test, en vez de utilizar varios test unitarios para poder testear una funcionalidad. En este caso, para testear una funcionalidad se utilizan generadores de datos para generar diferentes entradas de datos a la hora de ejecutar una propiedad. Como el valor de los datos para poder comprobar una propiedad es variable en cada comprobación de una propiedad, es necesario definir una validación para la propiedad genérica (lo que se denomina *assertions*) que sea suficiente para cualquier tipo de valor de los datos utilizado en la propiedad. El concepto de PBT fue inventado

```scala
import org.scalacheck.Properties
import org.scalacheck.Prop.forAll

object MathProps extends Properties("Math") {
   property("max") = forAll { (x: Int, y: Int) =>
      val z = java.lang.Math.max(x, y)
      (z == x || z == y) && (z >= x && z >= y)
   }
}
```

Figura 2.2: Ejemplo de test PBT usando ScalaCheck

mediante la creación de la biblioteca Quickcheck para Haskell [7]. En el paper al que se hace referencia, se crea dicho concepto partiendo de lo que se llama *Random testing* que, como su propio nombre indica, consiste en la creación de test de manera aleatoria.

De esta manera, es posible poder testear un algoritmo sin necesidad de implementar varios test para poder comprobar la validez del código. Otras ventaja es que, al usar datos de manera aleatoria o pseudo-aleatoria, el test cubrirá de manera más fiable todas las posibilidades de la función a verificar que si se hiciese manualmente. Asimismo, el código es más sencillo de mantener debido a que la cantidad de líneas es menor [5, p.22-23]. Por contra, la programación de propiedades es una tarea más abstracta y complicada de llevar a cabo.

En la figura 2.2 se muestra el ejemplo de [5, p. 20]. En el código se puede ver cómo la cantidad de líneas de código es menor y cómo se define una función genérica en vez de varias pruebas con valores fijos. La aserción se comprueba que z será igual a x o y y que, además, z siempre tendrá que ser mayor o igual que x y mayor o igual que y. Con esta aserción, independientemente de lo que valgan x o y, la propiedad se cumplirá siempre. Este ejemplo se ha programado utilizando ScalaCheck, que es la biblioteca usada en Scala para PBT. Esta biblioteca es uno de los pilares del proyecto y se describirá con una mayor profundidad en la sección 2.4.

2.2. Sistemas distribuidos

Se define como un sistema distribuido, aquel en el que varios componentes de hardware y software están ubicados en una red, y dichos componentes son capaces de comunicarse entre ellos mediante el paso de mensajes [8]. Esto genera que en un sistema distribuido se den las siguientes características:

- La ejecución de un programa se realiza de manera concurrente (i.e. se realizan operaciones en paralelo entre los diversas máquinas).
- A la hora establecer una comunicación entre los componentes de un sistema distribuido, una de las arquitecturas más populares, y la que utiliza Apache Flink por debajo, es la que integra una comunicación mediante intercambio de mensajes.

- En cuanto al control de fallos, otro de los modelos más comunes y que usa Apache Flink, es mediante una arquitectura primario-secundario, en la que si un nodo secundario deja de funcionar, el nodo primario asigna el trabajo que tenía que realizar este nodo a otro nodo secundario.

Es necesario este pequeño apartado para resumir brevemente qué es un sistema distribuido, ya que, como se mencionará en la sección 2.3, Apache Flink se diseñó para funcionar en un *cluster*. Aunque existe diversidad de opinión sobre que diferencias hay entre un *cluster* y un sistema distribuido [9, 10], estas disimilitudes se asocian a cómo se distribuyen los recursos y cómo se ejecuta la tarea. Aun así, las bases son comunes tanto en un sistema distribuido como en un *cluster*.

2.3. Apache Flink

Apache Flink es un *framework* y un motor de procesamiento de datos distribuido *open-source*, que realiza cálculos sobre flujos de datos, tanto limitados como ilimitados. Apache Flink está diseñado para soportar los entornos de *cluster* más comunes [11]. Flink ha sido desarrollado por la Fundación de Software Apache. Flink esta programado en Java y en Scala y utiliza la JVM para ejecutar sus procesos.

2.3.1. Características

Las principales características de Apache Flink expuestas en [12, p. 24] son las siguientes:

- Permite el tratamiento de flujos de datos basado en eventos temporales. Flink utiliza unas semánticas para los eventos temporales, que proveen de una gran consistencia y precisión en los datos a la hora de devolverlos.
- Garantizada la consistencia de los datos en cada estado, mediante el uso de *check-points*. Este suceso se denomina *exactly-once state consistency*.
- Las aplicaciones en Flink pueden escalar para ejecutarse hasta en un millón de núcleos, además de poder estar ejecutandose 24/7 (algunos ejemplos de integración son Kubernetes, Apache Mesos y YARN).
- Flink posee una API para poder usar sus diversas funcionalidades:
    * `DataStream` API: se encarga de realizar ejecuciones sobre flujos de datos.
    * `DataSet` API: se encarga de realizar ejecuciones sobre grupos de datos estáticos.
    * `Table` y SQL API: nivel más alto de la API. Se utiliza para optimizar y validar consultas a nivel relacional. Se puede integrar con las 2 API anteriores.

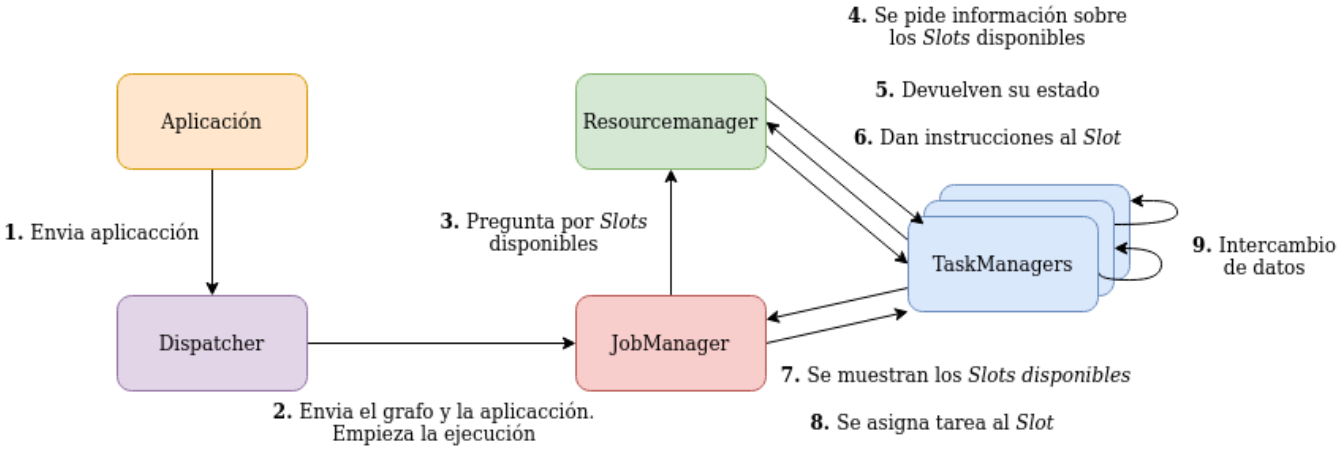


Figura 2.3: Interacciones entre componentes

Esta API funciona por debajo mediante un sistema de comunicación basado en paso de mensajes, como se comentó en la sección 2.2. Está diseñada de tal manera que el usuario pueda programar aplicaciones complejas de manera sencilla, gracias a un alto nivel de abstracción de la comunicación por paso de mensajes implementada en la API.

- Sistema de monitorización que permite hacer seguimiento a las diferentes tareas en ejecución. Dicho sistema contiene herramientas que permiten identificar posibles problemas antes de que ocurran.

- Aunque la funcionalidad principal de Flink es el tratamiento de flujos de datos, el motor tiene un procesador para tratar lotes de datos (mediante la API de `DataSet`).

De todas las funcionalidades que tiene Flink, se hará hincapié en que están relacionadas con el uso de las API de `DataSet` y `Table`, ya que en el desarrollo del mismo no se han utilizado flujos de datos en Flink. La capacidad de Flink de ser tolerante a fallos también se tratará, ya que consta como uno de los objetivos de este proyecto.

2.3.2. Arquitectura de Flink

La arquitectura de Flink está preparada para soportar procesamiento en paralelo de manera escalada. Dicha arquitectura está formada por muchos componentes, teniendo cada uno de ellos tareas muy bien definidas. Flink se diseñó para ser ejecutado en un *cluster*, así que la idea principal es que los diferentes procesos se repartan entre las máquinas que forman el *cluster* [12, p. 53]. A continuación se describirán los componentes principales de Flink en detalle y la manera en la que se conectan. Un vistazo rápido y general a la arquitectura se puede visualizar en la figura 2.3 [12, p. 54-55].

JobManager

Por cada aplicación ejecutada en Flink existe un solo *JobManager*. Este proceso recibe, por una parte la aplicación a ejecutar (e.g. Un fichero JAR) y por otra un grafo con un flujo de datos lógico llamado *JobGraph*. Este grafo contiene la información sobre la ejecución de la aplicación y este grafo es transformado por el *JobManager* en un grafo físico en el que, a cada tarea, se le asignara un *TaskManager* que se encargue de ejecutarla en paralelo. En la figura 2.4 se pueden observar ambos flujos.

Antes de entregar cada tarea a los *TaskManagers*, el *JobManager* se encarga de pedir y asignar los recursos que son necesarios para cada tarea. Por último, el *JobManager* es responsable de coordinar acciones comunes (como puede ser el uso de los *check-points*).

TaskManagers

Son los procesos que actúan como *workers*. Por cada aplicación, pueden existir uno o más *TaskManagers*, y cada *TaskManager* tiene una o varias unidades de procesamiento llamadas *Task Slots*. Una vez un programa se está ejecutando, los diferentes *TaskManagers* intercambian datos entre ellos cuando sea necesario, dentro de la misma aplicación. Cada *Task Slot* se encarga de ejecutar la tarea que le asigne el *TaskManager*.

ResourceManagers

Se encargan de indicar al *JobManager* cómo asignar las tareas a los *TaskManagers* en función de los espacios disponibles que estos tengan y los recursos que puede usar Flink en la aplicación a la que pertenecen estos procesos. Cuando un espacio de un *TaskManager* acaba de ejecutar su tarea, el *ResourceManager* se encarga de liberar los recursos del sistema que ocupa dicho espacio.

Dispatcher

El *Dispatcher* provee de una interfaz REST para poder enviar las aplicaciones y que se ejecuten en un *cluster* de Flink. La comunicación entre las distintas máquinas pertenecientes al *cluster* es llevada a cabo por el *Dispatcher* actuando como un punto de entrada HTTP. Por último, mediante el *Dispatcher*, Flink proporciona un panel para poder visualizar las diferentes tareas que se están ejecutando, las tareas finalizadas y el estado de finalización de estas como se puede observar en la figura 2.5.

2.3.3. Apache Spark. Diferencias con Flink

Apache Spark es un motor de procesamiento de datos distribuido como Flink. Este, reside dentro del conjunto de software perteneciente a Apache, al igual que Flink (Spark fue desarrollado por la Universidad de California en Berkeley). Spark es un software mucho más veterano que Flink, y por ende, la cantidad de aplicaciones,

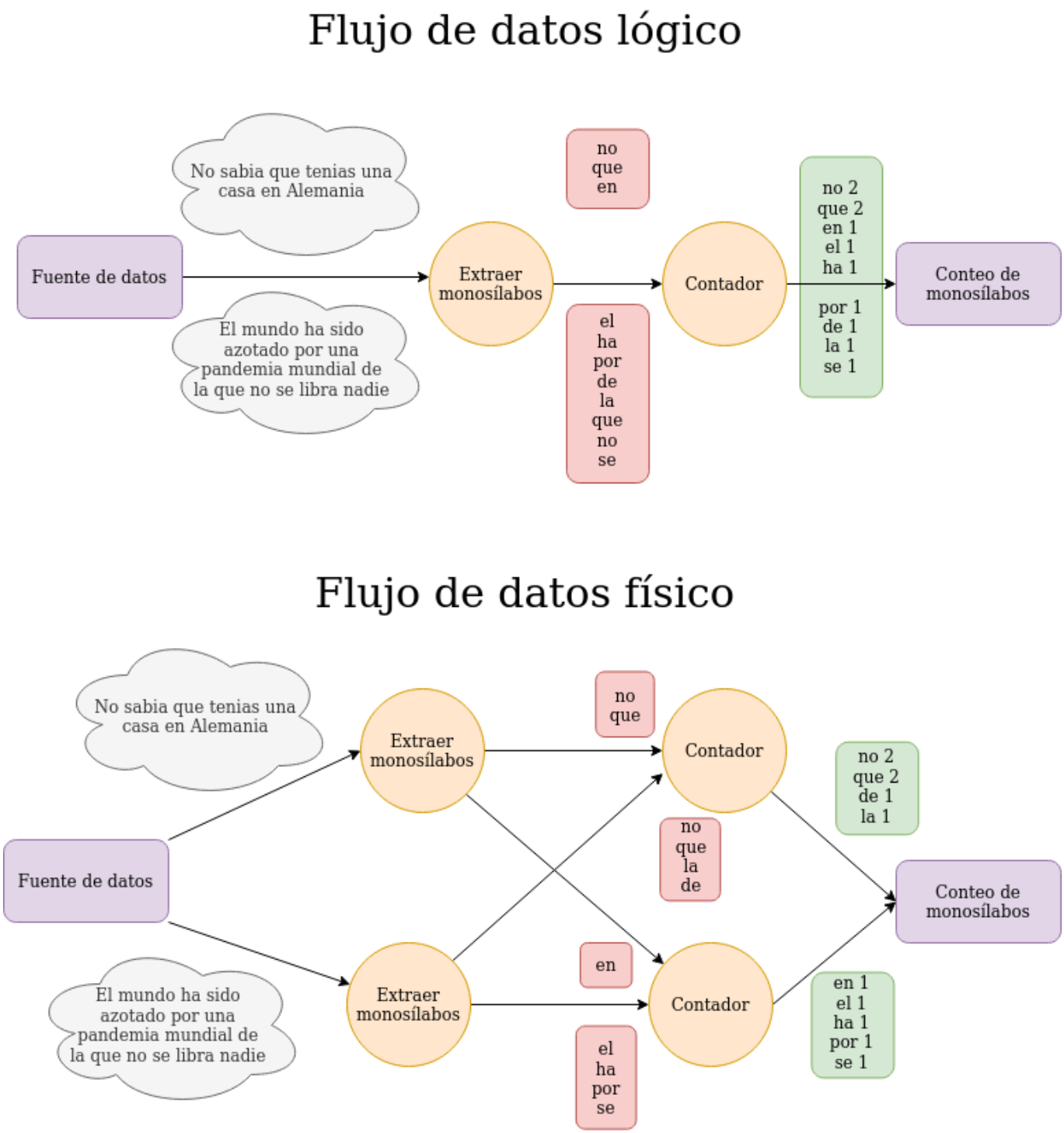


Figura 2.4: Comparación del flujo de datos lógico con el flujo de datos físico

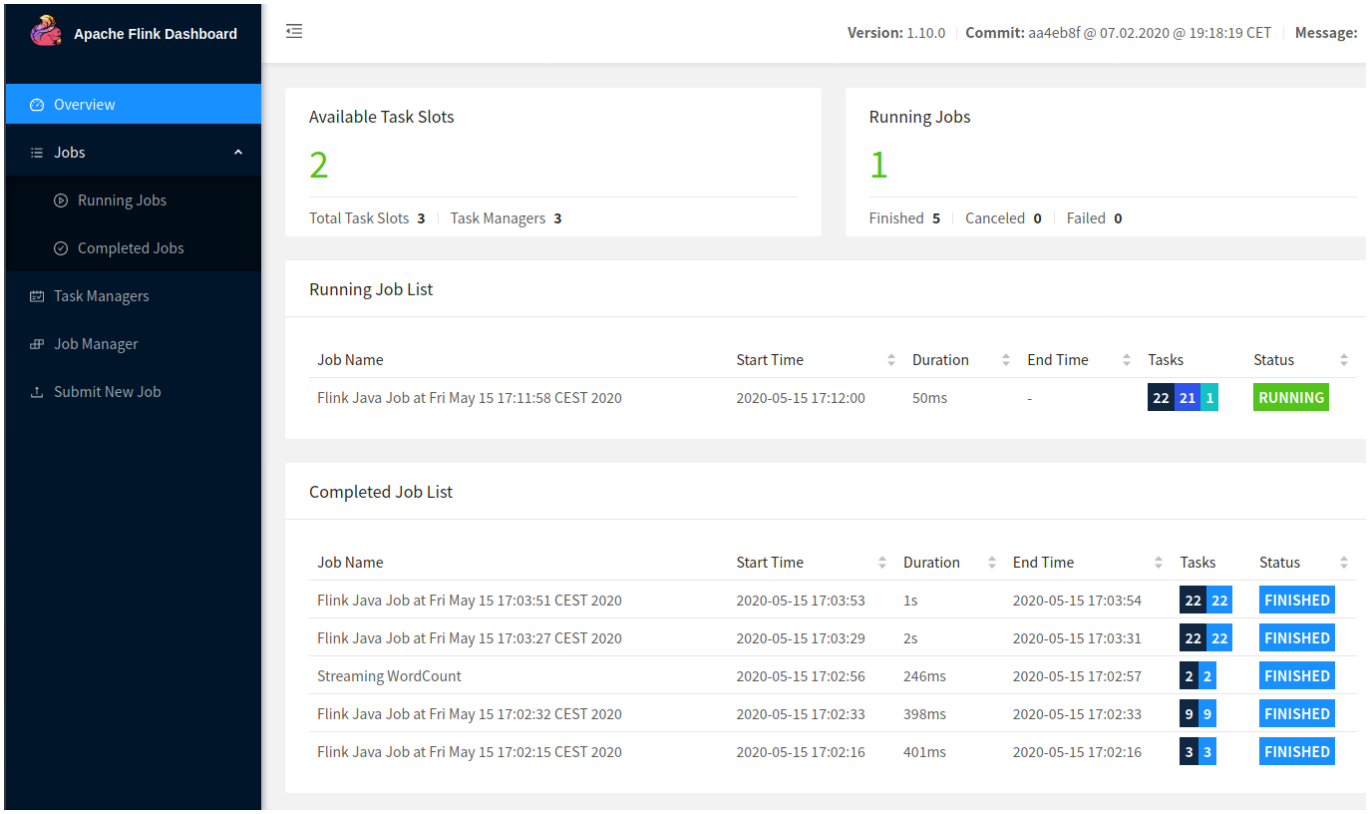


Figura 2.5: Página principal del panel de Apache Flink

amplitud de documentación y tamaño de comunidad es mucho mayor que la de Flink. A su vez, existen una mayor cantidad de bibliotecas de testeo basado en PBT (ver sección 2.4.2), una de las razones por las que se decidió crear una herramienta en Flink en vez de en Spark. La tabla 2.1 muestra una comparativa entre ambos motores [13, 14].

Apache Flink es más joven que Spark, e implementó directamente un sistema de procesamiento nativo de flujo de datos para no tener los mismos problemas que ha tenido Apache Spark. Como Flink es más reciente, hay una menor cantidad de software y bibliotecas que lo implementen, crear herramientas nuevas es más sencillo, como se ha hecho con este proyecto.

2.4. Scala

Scala [15, 16] es un lenguaje que combina la programación funcional y la programación orientada a objetos. Este lenguaje se ejecuta en la JVM, lo que provee al lenguaje con las bibliotecas que también posee Java. Esto significa que Scala tiene un amplio ecosistema de bibliotecas al que acceder. A su vez, las bibliotecas de Scala son compatibles en Java.

Se ha escogido Scala como lenguaje de programación para este proyecto debido a que Apache Flink esta programado principalmente en Scala. Esto hecho produce que toda la funcionalidad de Flink en sus diversas capas de la API este disponible

	Apache Flink	**Apache Spark**
Procesamiento	Puede procesar flujos de datos de manera real y adapta este procesamiento para implementar procesamiento por lotes.	Realiza procesamiento por microlotes para simular un procesamiento de flujos de datos.
Plan de ejecución	Implementa ciclos de flujos de datos, evitando así, cálculos extras.	Implementa ciclos DAG. (*Directed Acyclic Graph*).
Memoria	Implementación propia más eficiente que usar la JVM.	Utiliza principalmente la memoria de la JVM.
Comunidad	Flink es más usado en Asia.	Spark es más usado en Europa y EE.UU.
Rendimiento	En cuanto al procesamiento con flujos de datos, el rendimiento es mayor que en Spark.	Mayor rendimiento en procesamiento por lotes.
Latencia	Muy baja latencia.	Baja latencia.

Tabla 2.1: Comparativa entre Apache Flink y Apache Spark

para Scala.

En los siguientes sub-apartados se describirán las bibliotecas y herramientas principales que se han usado en este proyecto.

2.4.1. SBT

SBT (*simple build tool*) es una herramienta para construir proyectos en Scala y Java, añadiendo las librearías que sean necesarias en cualquier momento del desarrollo del código [17].

En este proyecto se ha utilizado SBT para añadir todas las bibliotecas necesarias para poder ejecutar desde la terminal los test realizados mediante las bibliotecas Specs2 y ScalaMeter, como se refleja en el capítulo 4.

SBT tiene una funcionalidad mucho mayor a la descrita en esta sección. Para poder acceder a más documentación y a algunos ejemplos, consultar [18].

2.4.2. ScalaCheck

ScalaCheck es una biblioteca escrita en Scala, usada para la creación de test automáticos basados en propiedades, es decir, para PBT [5] [19]. El origen de esta biblioteca viene de QuickCheck [7], que es la primera biblioteca que se creó de PBT, en Haskell.

Para el desarrollo del proyecto se han utilizado una gran cantidad de funcionalidades de ScalaCheck. Las principales han sido las siguientes.

Propiedades

Una propiedad es la unidad más pequeña de testeo en ScalaCheck. Viene definida por la clase `org.scalacheck.Prop` y contiene funciones para definir diferentes tipos de

```
1 import org.scalacheck.Properties
2
3 val commutative = Prop.forAll {x1: Int, x2: Int =>
4    x1 + x2 == x2 + x1
5 }
```

Figura 2.6: Ejemplo de uso `Prop.forAll`

```
1 import org.scalacheck.Properties
2
3 val sumProp = Prop.forAll {x1: Int, x2: Int =>
4    (x1 != 0 && x2 != 0) ==>
5       val sum = x1 + x2
6       sum > x1 && sum > x2
7 }
```

Figura 2.7: Uso de pre-condiciones en una propiedad

propiedades. La más usada es `Prop.forAll` que indica que el código de la propiedad debe de cumplirse para cualquier conjunto de valores dado. Otro tipo de propiedad común es `Prop.exists` que, a diferencia de `Prop.forAll`, en cuanto un conjunto de valores de entrada cumple la propiedad, el test es superado con éxito. En la figura 2.6 muestra un pequeño ejemplo con `Prop.forAll` sobre la propiedad conmutativa en la suma.

Una vez se ha definido una propiedad, para poder ejecutar el test y ver si esa propiedad se cumple, es necesario utilizar el comando `check` (en el ejemplo definido en la figura 2.6 sería `commutative.check()`).

Otro concepto importante de las propiedades y usado en el desarrollo de este proyecto, es la capacidad de generar unas pre-condiciones a una propiedad. Se puede usar como ejemplo otra propiedad sencilla (figura 2.7) en la que se quiera afirmar que la suma de dos números sea siempre mayor que ambos por separado. Este caso es cierto siempre y cuando ninguno de los números valga cero.

Generadores

Los generadores son un concepto básico del testeo basado en propiedades, ya que se encargan de producir valores de un tipo concreto de manera aleatoria o pseudo-aleatoria. Un generador de datos toma unos parámetros para definir qué tipo de datos se van a generar y de qué manera [5, p. 39-40]. La clase en ScalaChek es `org.scalacheck.Gen`.

En función del tipo de la variable existen varias funciones para crear generadores. Los más usados en este proyecto son los siguientes:

- `Gen.choose()`: recibe como parámetro dos números y el generador devuelve un valor en el rango creado entre ambos números. `Gen.chooseNum()` realiza la misma función.
- `Gen.posNum()`: devuelve un número positivo.

```
1 import org.scalacheck.Gen
2
3  val genCliente: Gen[Cliente] = for {
4      nombre <- Gen.oneOf("Enrique", "Juan", "Antonio", "Ramon", "Olaya",
           "Carla", "Pedro", "Virginia", "Maria", "Cecilia")
5      mayoria_edad <- Gen.oneOf(true, false)
6      gastos <- Gen.chooseNum(0.0, 1000.0)
7    } yield Cliente(nombre, nacionalidad, gastos)
```

Figura 2.8: Generador personalizado de una clase Cliente

- `Gen.negNum()`: devuelve un número negativo.
- `Gen.alphaStr()`: genera una cadena de caracteres aleatoria.
- `Gen.oneOf()`: este generador es válido con cualquier tipo de variables. Su funcionalidad consiste en seleccionar un valor, de un conjunto de valores pasados por parámetro. Los tipos entre los valores usados pueden diferir.
- `Gen.listOf()`: el generador creado devuelve listas de longitud variable con valores creados por un generador pasado como parámetro.
- `Gen.listOfN()`: misma funcionalidad que el generador anterior, pero creando listas de tamaño fijo. El tamaño se pasa como parámetro al igual que el generador.

Además, se pueden crear generadores personalizados. Un ejemplo es el mostrado en la figura 2.8 sobre la clase `Cliente` creada.

Una vez se tiene almacenado en una variable el generador, para obtener un valor se llama a la función `Gen.apply()` la cual devuelve un objeto de tipo `Option`. Esto se debe a que un generador puede llegar a fallar. En caso de que falle devuelve `None` o el valor que especifique el programador, si no desea dicho valor. Para obtener el valor real encapsulado en `Option`, se aplica el método `Option.get()`.

Por último, es necesario explicar el significado de las semillas (*Seeds*) en un generador de Scalacheck, ya que esta funcionalidad se ha utilizado en la herramienta creada en este proyecto. ScalaCheck contiene un objeto `org.scalacheck.rng.Seed` que se puede aplicar a un generador. Esto produce que si se crean 2 generadores iguales con la misma semilla, los datos que crearán serán los mismos siempre.

2.4.3. Specs2

Specs2 [20] es un *framework* que permite estructurar los test, de manera que sea mas sencilla su implementación, y provee una serie de funcionalidades que permiten elaborar test de la manera más completa posible. Se ha usado en este proyecto debido a su compatibilidad con ScalaCheck y con SBT, pudiendo así ejecutar grupos de propiedades desde la terminal (también compatible con otros tipos de testeo).

Specs2 funciona creando especificaciones, en las cuales se debe de escribir los test necesarios para evaluar una porción de código. La unidad básica para medir

el comportamiento es mediante objetos `Matcher`. Estos objetos permiten comparar una gran cantidad de tipos y objetos que necesitan testear. Desde una igualdad entre dos números (`1 must beEqualTo(1)`) hasta la posibilidad de lanzar excepciones (`throwA[ExceptionType](message = "Mensaje de error")`).

Se ha usado esta biblioteca en el proyecto debido a que es compatible con ScalaCheck y ofrece muchas facilidades a la hora de implementar test. Asimismo, se han usado algunos objetos `Matcher` de la biblioteca flink-check [2].

2.4.4. ScalaMeter

ScalaMeter [21] es un *framework* que realiza un *microbenchmarking* al ejecutar una sección de código, devolviendo los tiempos de ejecución en dicha sección. La biblioteca se ejecuta sobre la JVM. En este proyecto se ha usado para realizar funciones de coste sobre el generador de `DataSet` de Apache Flink. ScalaMeter permite generar gráficas usando HTML y JavaScript de manera automática, si se le indica en el código. Una de sus funcionalidades más interesantes es que permite mostrar un intervalo de confianza sobre las gráficas creadas. El intervalo de confianza se forma mediante un número máximo y otro mínimo entre los cuales, se encuentran los diferentes valores obtenidos en un muestreo. También se proporciona una media. Es importante saber estos datos, ya que obtener solo la media de todos los datos de un muestreo a veces no es suficiente información, mientras que con el intervalo de confianza se puede ver cuanto han variado los valores de la muestra.

2.5. *Testing* en motores de procesamiento de datos

En esta sección se tratarán bibliotecas y proyectos ya existentes de testeo en motores de procesamiento de datos. Se analizarán y comentarán estas alternativas en Apache Hadoop, Apache Spark e incluso en el propio Apache Flink, en los siguientes apartados.

2.5.1. Apache Hadoop

Apache Hadoop es un *framework* desarrollado por la *Apache Software Foundation* para poder procesar grandes cantidades de datos de manera distribuida. Esta biblioteca fue diseñada para detectar y manejar errores en la capa de aplicación, ofreciendo de esta manera una alta disponibilidad en dicha capa [22]. Apache Hadoop es el sistema con el menor número de aplicaciones orientadas al testeo de los motores propuestos. Su biblioteca más significativa es MRUnit [23]. Esta biblioteca está escrita en Java, y se utiliza para la creación de test unitarios sobre trabajos *MapReduce*.

MapReduce [24] es el modelo de programación que utiliza Hadoop para manejar grandes cantidades de datos. El concepto de este método es partir de una entrada de valores clave/valor y devolver una salida, también con el formato de clave/valor. Este proceso se divide en 2 fases obligatorias y una opcional a definir por el desarrollador:

- ***Map***: de una entrada con datos en formato clave/valor, se transforman dichos datos siguiendo el código que haya diseñado el desarrollador para devolver listas con el formato clave/valor (i.e. $k1, v1 \rightarrow list(k2, v2)$).

- ***Reduce***: recibe una lista de valores por cada clave, y se realiza una operación sobre dichos valores. El resultado que se devuelve es la clave recibida por la entrada de la función *reduce* y el valor obtenido de la operación realizada (i.e. $(k2, list(v2)) \rightarrow list(k2, v2)$).

- ***Combine***: es la función que es opcional. Se ejecuta después de la función *map* y antes de la función *reduce*. Esta operación suele realizar los mismos pasos que *reduce* para aligerar el coste de esta, teniendo que procesar una menor cantidad de datos.

Dicho esto, el funcionamiento de MRUnit tienen un funcionamiento muy parecido a cualquier tipo de test unitario, aunque en este caso, aplicado al modelo *MapReduce*. Una vez definidas las funciones de *map* y *reduce*, se definen funciones de test en las que se especifica una entrada para el *map*, el *reduce* o la combinación de ambas funciones, y la salida deseada. Uno de los aspectos más testeados en estos casos no es el valor exacto de la salida (un dato complicado de obtener si se prueba con conjuntos de datos muy altos) sino el tipo y formato del resultado. Para ver un ejemplo de test ir al apartado de tutorial en [23]. MRUnit se notificó como obsoleta en 2016.

Este sistema de testeo es poco práctico debido a que es bastante complicado realizar test con conjuntos de datos muy grandes y, una vez más, al ser necesario definir varios test con varias entradas definidas manualmente, se convierte en un proceso tedioso. Por otra parte, esta es una biblioteca que está obsoleta para un sistema que hace tiempo que se ha visto superada por competidores como Apache Spark o Apache Flink.

2.5.2. Apache Spark

En lo referente a bibliotecas para poder testear Apache Spark, existe un mayor número de alternativas en comparación con Hadoop, y además usando PBT, en vez de testing unitario. En esta sección se van a tratar, por una parte la biblioteca *spark-testing-base*, y por otra parte la biblioteca sscheck.

Spark-testing-base

Esta biblioteca diseñada por Holden Karau [25] utiliza ScalaCheck para crear diversas herramientas para poder implementar un testeo basado en propiedades en Scala. Sus principales características son las siguientes:

- Proporciona un contexto especial de Spark para usar en el testeo (`SharedSparkContext`).

- Existen funcionalidades que permiten establecer comparaciones entre las diferentes estructuras de datos de Spark, como `RDD`, `DataFrame` y `DataSet`.

- Incluye la capacidad de crear generadores de los mismos tipos comentados en el punto anterior.
- Permite una simulación la manera en la que Spark trata los flujos de datos mediante la clase *StreamingSuiteBase*.
- Permite simular un trabajo en un *cluster* HDFS usando test locales.

En la wiki del proyecto de GitHub existen varios ejemplos del funcionamiento de esta biblioteca en Spark. Partes de esta biblioteca se han usado para crear diversas alternativas en Apache Spark como la que se explicará en el siguiente apartado. El *testing* en spark-testing-base es a nivel de lotes y no de flujos de datos.

Sscheck

Sscheck [26, 27] se basa en la biblioteca spark-testing-base, previamente comentada. La aportación de Sscheck es el *testing* sobre flujos de datos. Esta biblioteca fue diseñada por Adrián Riesco y Juan Rodríguez Hortalá, y utiliza lógica temporal para escribir propiedades de ScalaCheck en programas que usan flujos de datos de Apache Spark. Esta lógica se llama Lógica Lineal Temporal para Sistemas en Streaming (*LTLss*) y se utiliza para expresar cómo dependen unos eventos de otros, a la hora de establecer las condiciones que tiene que cumplir una propiedad.

Este trabajo hecho en Spark se adaptó en un futuro para los flujos de datos en Flink, como se explicará en la siguiente sección.

2.5.3. Apache Flink

Referente a Apache Flink se van a tratar dos sistemas de testeo diferentes. El primero es el sistema propio que tiene Apache Flink, basado en test unitarios. Es interesante ver las propias herramientas que tiene el sistema para poder testear su funcionalidad. El segundo es la biblioteca flink-check basado en PBT sobre flujos de datos.

Testing unitario en Apache Flink

El sistema de testeo que tiene integrado Apache Flink es unitario. El sistema se basa en testear funciones de Flink (como pueden ser `map`, `flatMap`, etc.) cuando se han redefinido por el usuario. Estas funciones [28] se pueden analizar en función de tres tipos de operadores:

- **Operadores sin estado:** son las operaciones más sencillas de testear ya que no se tiene en cuenta estados anteriores al que se va a testear. Suponiendo una función *map* en la que se incrementa en 1 el entero que se pasa por parámetro, se crea una clase de test que comprueba que si se ejecuta dicha función pasando un 2 como parámetro, el resultado final debería ser 3, como se indica en la figura 2.9 (la función `IncrementMapFunction` incrementa en 1 el valor que se le pase por parámetro).

```
1  class IncrementMapFunctionTest extends FlatSpec with Matchers {
2
3     "IncrementMapFunction" should "increment values" in {
4        val incrementer: IncrementMapFunction = new IncrementMapFunction()
5
6        incremeter.map(2) should be (3)
7     }
8  }
```

Figura 2.9: Testeo unitario en Apache Flink

- **Operadores con estado:** a diferencia del ejemplo anterior, en este caso es necesario considerar que el estado actual es el correcto. Para ello, Flink permite especificar diferentes entradas de datos en diferentes eventos temporales, y así poder desarrollar test en los que se asignen unos valores en unos estados concretos.
- **Operadores de procesos temporizados:** similar al ejemplo anterior. En este caso es necesario proveer marcas de tiempo a los eventos, a diferencia del testeo usando operadores con estado. Esto significa que es necesario controlar el tiempo en la aplicación contra la que se realicen los test.

Esta herramienta integrada en Flink está pensada para usarse principalmente con flujos de datos y mediante test unitarios, en vez de usar lotes de datos y test basado en propiedades, que es la funcionalidad que se implementa en este proyecto.

Flink-check

Flink-check [2] es una biblioteca que extiende la funcionalidad implementada en sscheck, adaptándola a Apache Flink. Con lo cual, flink-check permite PBT mediante el uso de lógica temporal lineal usando propiedades de ScalaCheck en programas que tratan flujos de datos de Apache Flink.

Esta biblioteca solo cubre los programas que usan flujos de datos (i.e. `DataStream`), con lo cual los programas que utilizan procesamiento de datos por lotes (i.e `DataSet`) no están contemplados por esta biblioteca. Los generadores de `DataSet` desarrollados en este Trabajo Fin de Máster persiguen complementar la funcionalidad de *flink-check* y permitir PBT sobre programas que procesan lotes en lugar de flujos.

Capítulo 3

Generadores de `DataSet` y `Table`

En la sección anterior se han detallado las diferentes herramientas y tecnologías que se utilizan en el desarrollo de este proyecto. Además, se han planteado, y brevemente explicado, las alternativas existentes en lo referente al testeo en motores de procesamiento masivo, tanto de flujos de datos, como de datos en lotes.

En los siguientes capítulos del proyecto se incluirá una explicación detallada de cómo se han cumplido los diferentes hitos planteados en la sección 1.3, las diferentes decisiones tomadas para ello, y los principales problemas encontrados mientras se implementaban las diferentes funcionalidades. Se puede acceder al código del proyecto mediante el siguiente repositorio de GitHub: `https://github.com/AntonioBarral/TFM-flink-dataset-generators`.

El primer paso de este proyecto fue el desarrollo del propio generador. Dicho generador tenía que cumplir las características definidas en la sección 1.2, que se explicarán una a una en las siguientes sub-secciones.

3.1. Estructura general

El primer paso antes de explicar cada funcionalidad del generador en detalle, es explicar cómo es su estructura general, y cómo se puede usar. Para ello es conveniente definir qué parámetros son necesarios para llamar a la función del generador y cual es la salida que devuelve. La función en cuestión se llama `generateDataSetGenerator` y los parámetros que necesita son:

- `numElements`: este parámetro es el número de elementos que se va a generar por cada partición.
- `numPartitions`: con este valor se define el número de particiones entre las que se va a dividir el trabajo en Flink a la hora de crear el generador de `DataSet`.
- `g`: este parámetro es el generador de los elementos individuales del `DataSet`.
- `seedOpt`: es el valor de la semilla que se le asigna al generador para poder reproducir el mismo `DataSet`. Es un parámetro opcional y por defecto vale `None`.

```
1  def generateDataSetGenerator[A: ClassTag : TypeInformation]
2   (numElements: Int, numPartitions: Int, g: Gen[A], seedOpt: Option[Int] = None)
3   (implicit env: ExecutionEnvironment =
        ExecutionEnvironment.getExecutionEnvironment)
4     : Gen[DataSet[A]] = {
5
6     val seedGen: Gen[Int] = if (seedOpt.isDefined) Gen.const(seedOpt.get) else
          Arbitrary.arbitrary[Int]
7     for {
8       seed <- seedGen
9       seeds = Gen.listOfN(numPartitions,
            Arbitrary.arbitrary[Int]).apply(Parameters.default,
            Seed.apply(seed)).get
10    } yield
11      env.fromCollection(seeds)
12      .rebalance()
13      .flatMap { xs =>
14        val elements: List[A] = Gen.listOfN(numElements,
              g).apply(Parameters.default, Seed.apply(xs)).getOrElse(Nil)
15        elements
16      }
17      .setParallelism(numPartitions)
18   }
```

Figura 3.1: Función de creación de generador para `DataSet`

La función devuelve un objeto `Gen[DataSet[A]` siendo `A` el tipo de valor reproducido por el generador `g`. Las figura 3.1 muestra el código de la función en sí. En los siguientes puntos se referenciará esta figura para explicar los puntos más importantes de la misma.

Además de los parámetros mencionados, es necesario añadir como parámetro implícito el `ExecutionEnvironment` de Flink. Un parámetro implícito en Scala es aquel que se obtiene del contexto en el que se llama a la función que tiene declarado dicho parámetro. Esto significa que desde el programa en el que se llama al método `generateDataSetGenerator`, se ha declarado como implícita la variable `ExecutionEnvironment`.

Por otra parte, el `ExecutionEnvironment` es el entorno de ejecución que usa Flink en sus programas. Dicho entorno se puede configurar manualmente teniendo en cuenta que un programa se ejecute en un entorno local (teniendo en cuenta los recursos del mismo) o teniendo en cuenta que se va a ejecutar en un *cluster*. La última de las opciones, y la usada en los diferentes programas de testeo creados en este proyecto, es la asignación de recursos en función del contexto. Esto significa que Flink sabrá distinguir si debe de lanzar el programa de manera local o si el trabajo se debe de realizar en un *cluster*. Como en el desarrollo de este proyecto no se ha utilizado ningún *cluster*, se ha declarado el entorno para que Flink elija qué recursos va a necesitar en función del trabajo a ejecutar (llamando a la función `getExecutionEnvironment`).

La razón por la que este parámetro es implícito, es que se necesita obtener del contexto del programa que llama a `generateDataSetGenerator` qué recursos quiere

```
1 import org.scalacheck.Gen.choose
2    val myGen = for {
3    n <- choose(1, 50)
4    m <- choose(n, 2*n)
5 } yield (n, m)
```

Figura 3.2: Ejemplo de uso de creación de generadores ScalaCheck

usar, en vez de declarar unos fijos en el método, lo que no sería adecuado.

Respecto al tipo genérico de dato del `DataSet` que se quiere reproducir con el generador (se ha nombrado `A` en la función) es importante explicar un par de conceptos necesarios para poder definir dicho tipo en Apache Flink.

- Como hasta que no se ejecuta el programa Flink no sabe el tipo del valor que reproduce el generador `Gen[A]` definido en `g`, y por ende, el que se va a devolver en el `Gen[DataSet[A]`, es necesario proporcionarle dicha información a Flink. Para ello en la declaración de la función es necesario asignarle a `A` un `TypeInformation`. La clase `TypeInformation` es la que contiene la información básica de los tipos en Flink, además de encargarse de generar los serializadores y comparadores entre valores del mismo tipo, entre otras especificaciones [29].

- Por otra parte, como va a ser necesario operar con dicho tipo genérico, es necesario añadir también `ClassTag`. En este caso, es necesario darle la información sobre el tipo que se va a utilizar en el programa, a la JVM en tiempo de ejecución, y esa es la función de `ClassTag`.

Para finalizar este apartado es importante mostrar como se construye un generador. Para ello se utiliza una comprensión usando los comandos `yield` y `for` como en el ejemplo de la figura 3.2 mostrado en [5, p. 41].

Todos los puntos descritos en esta sub-sección referentes a la estructura se reflejan en la cabecera de función `generateDataSetGenerator` expuesta en las líneas 1 y 2 de la figura 3.1

3.2. Escalabilidad

Aunque las pruebas solo se han podido llevar a cabo en un solo ordenador, uno de los objetivos más importantes a la hora de crear el generador es que este fuese escalable y que la tarea se repartiese de manera homogénea entre los trabajadores. Para poder llevar a cabo una repartición equitativa entre los recursos que van a intervenir, se utiliza el parámetro referente al número de particiones en las que se va a repartir el trabajo a realizar para crear el generador.

La idea es que cada partición genere un generador de listas partiendo del generador `g` con un número de elementos igual a `numElements`, luego dichas particiones se asignen de manera balanceada entre los recursos disponibles, y por último dichas listas se junten en el `Gen[DataSet[A]]` final. Este proceso se puede ver reflejado en el comando `flatMap` de las líneas 12 a la 15 de la figura 3.1 (de momento se va a

obviar el código referente a la semilla, ubicado en el segundo parámetro de la función `Gen.listOfN`).

Para indicar a Flink el número de particiones que se tienen que crear y cómo repartirlas entre los recursos disponibles en el entorno son necesarias las siguientes funciones:

- `setParallelism` [30], invocada en la línea 16 de la figura 3.1, permite indicar a Apache Flink el nivel de paralelismo que se requiere (i.e. el número de particiones). A esta función se la puede llamar en cualquier sección de código, siempre y cuando haya una referencia al entorno de ejecución de Flink en el contexto. La razón de realizar la llamada dentro de la estructura de creación del generador (es decir, dentro del `for/yield`) es para que solo se aplique dicha configuración en este trabajo de Flink, sin que afecte a otras secciones de código.

- `rebalance` [31], invocada en la línea 11 de la figura 3.1, es la función encargada de redistribuir las particiones en las que se divide la tarea que ejecuta `flatMap` entre los recursos disponibles del entorno, de manera balanceada. Se planteó el uso de la función `partitionByHash`, la cual, en función de una clave o un id, asigna los elementos con misma clave o identificador al mismo *Task Slot*, pero al final se implementó `rebalance` ya que a la hora de crear el generador no se puede saber de cuántos recursos se dispone, siendo más sencillo que Flink se ocupe de distribuir balanceadamente.

Es importante clarificar que, aunque la creación del generador sea una tarea escalable, no siempre todos los *Task Slots* que utilice el entorno de ejecución van a recibir el mismo número de particiones, ya sea porque la división entre *Task Slots* y particiones no es exacta o porque el número de elementos a generar por partición es lo suficientemente pequeño como para que Flink decida asignar más particiones a un *Slot*.

3.3. Tolerancia a fallos

Al no poder controlar fallos debidos a una perdida de conexión puntual o a que uno de los *Task Slots* falle en medio de la ejecución del programa, se ha diseñado la función de creación de generadores para que, en caso de que ocurra uno de estos fallos y se vuelva a reintentar la tarea de Flink, el generador de `DataSet` va a reproducir los mismos elementos que se estaban generando en el intento anterior. La tolerancia a fallos que se ha implementado es parcial, ya que en caso de que la tarea falle un número superior de veces al establecido en la configuración de Flink, el programa fallaría y se perderían las referencias a la semilla y al generador.

Para poder lograr este objetivo, se ha decidido implementar la funcionalidad que proporciona la clase `Seed` de la librería ScalaCheck. Como se mencionó en la sección 2.4.2, al aplicar una semilla en un generador, se consiguen reproducir siempre los mismos valores para dicho generador. En el caso de la función de creación de generadores expuesta en la figura 3.1, son necesarias dos semillas:

1. La primera semilla es la que se define con el parámetro `seedOpt`, que está encapsulada en un `Option`. La semilla en cuestión es un entero pasado por parámetro, de manera opcional, por el desarrollador que ejecute la función. Un solo entero no es suficiente para crear de manera óptima el generador ya que si se especifica un número de particiones superior a uno y se aplica la semilla en cada partición, se reproducirán los mismos valores en cada partición. Para solucionar este problema, esta semilla se aplicará para obtener una lista de semillas (tantas como particiones haya), definida en la línea 8. Si no se pasa por parámetro un valor para `seedOpt`, se genera un entero aleatorio mediante el generador de la línea 5.

2. La segunda semilla es, en realidad, una lista de semillas generada en la línea 8 aplicando la semilla explicada en el punto anterior. Dentro del código implementado en el `yield`, cada partición genera un conjunto de elementos diferente al resto de particiones, porque se le asigna una semilla de la lista de semillas, en función del número de la partición.

Para comprobar el correcto funcionamiento de la implementación de semillas, se creó un test usando specs2 en el que se forzaba a fallar el trabajo ejecutado en Apache Flink, y una vez se superaba el número de reintentos permitidos, se comprobaba que los datos reproducidos por el generador eran iguales en todos los reintentos. El código escrito para este test realiza los siguientes pasos:

1. El primer paso es llamar a la función para crear un generador d `DataSet`, en este caso, de enteros.

2. Sobre el generador, se llama a una `RichMapPartitionFunction`, que es una función `mapPartition` personalizada [31]. Esto es necesario para poder obtener en cada partición el identificador de la partición y el intento en el que se encuentra la ejecución.

3. Una vez se han obtenido tanto el identificador como el intento, se escriben los datos asignados a dicha partición en el intento correspondiente, en un fichero temporal con la estructura "partition_*numPartición*_attempt_*numIntento-idAutogenerado*".

4. Cuando se han almacenado los datos en el fichero temporal se lanza manualmente una excepción para forzar el reintento de Flink.

5. Una vez se han realizado todos los reintentos se realizan 2 test para comprobar que los datos son iguales.

   a) Se comprueba que cada partición ha generado siempre los mismos datos en cada intento

   b) Se comprueba, uniendo los datos generados por todas las particiones de cada intento, para obtener todos los elementos que contendría el `DataSet` resultante. Luego se compara que los conjuntos formados en cada intento son iguales entre ellos.

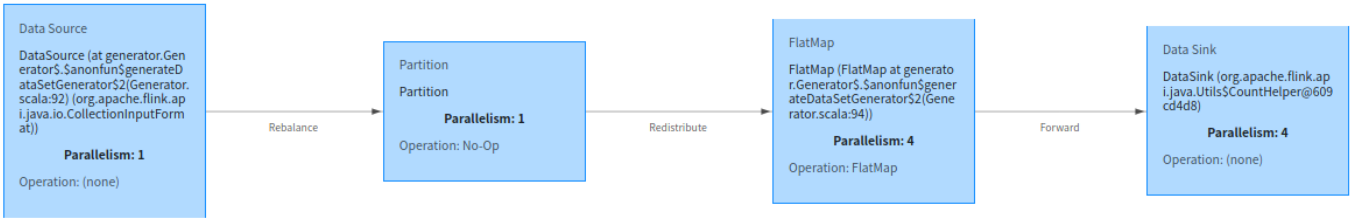


Figura 3.3: Flujo del trabajo enviado al cluster, mostrado por el panel de Flink

Este test se realizó probando con diferentes números de particiones, tanto pasando una semilla por parámetro como sin pasarla y usando un número de elementos pequeño (inferior a 100) ya que el objetivo no era medir la escalabilidad de la función, sino su corrección.

3.4. Ejemplo de ejecución mediante *cluster standalone* de Apache Flink

El propósito de este sub-apartado es mostrar un ejemplo de llamada a la función y mostrar el procesamiento que realiza un *cluster standalone* de Flink mediante el uso de su interfaz REST.

Para el ejemplo se va a crear un generador de `DataSet` de 1 millón elementos enteros, que sean mayores de 0 y menores de 500 y se van a usar 4 particiones. El mapeo de estos valores a los parámetros de la función sería el siguiente:

- `numElements = 250`.
- `numPartitions = 4`.
- `g = Gen.choose(0,500)`.
- `seedOpt` en este caso no es necesaria una semilla así que no se le asigna valor.

El *cluster* está compuesto de un *JobManager* y tres *TaskManagers*, cada uno con un *TaskSlot*. La figura 3.3 muestra el flujo que ha seguido la ejecución del programa para crear el generador. Los recuadros más importantes son el segundo y, sobretodo, el tercero. El segundo recuadro solo indica que se va a balancear equitativamente la carga de trabajo entre los recursos del entorno, mediante la función `rebalance`. El tercer recuadro, como se puede observar, ya aplica un paralelismo igual al número de particiones que se ha indicado en el ejemplo. Dicho recuadro es el que ejecuta la función `flatMap` en donde se crea el generador. El primer recuadro representa la preparación del entorno para realizar el trabajo, mientras que el último recuadro indica la recogida de los datos (en este ejemplo mediante la ejecución de una función `count`).

Para obtener una mayor cantidad de información sobre qué ha pasado, es necesario clicar cada uno de los recuadros. En este caso, el que más información contiene

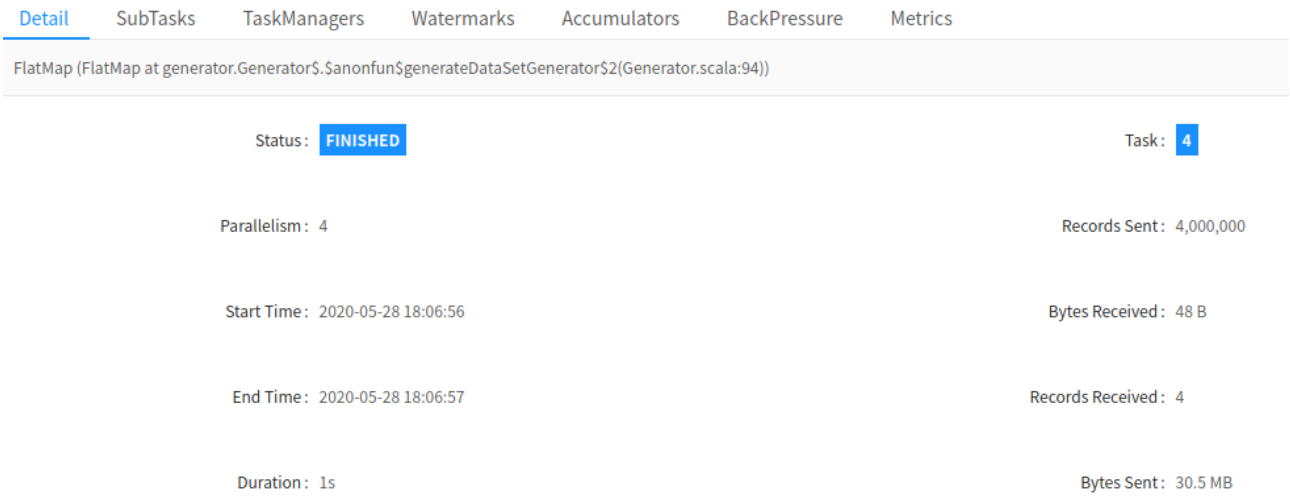
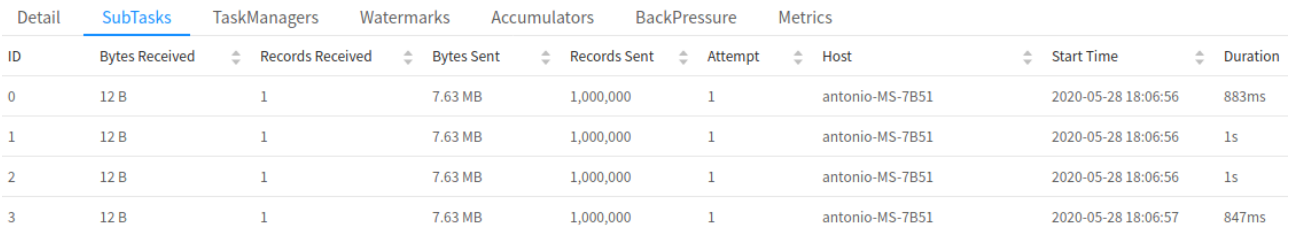


Figura 3.4: Información general sobre la repartición de la tarea



ID	Bytes Received	Records Received	Bytes Sent	Records Sent	Attempt	Host	Start Time	Duration
0	12 B	1	7.63 MB	1,000,000	1	antonio-MS-7B51	2020-05-28 18:06:56	883ms
1	12 B	1	7.63 MB	1,000,000	1	antonio-MS-7B51	2020-05-28 18:06:56	1s
2	12 B	1	7.63 MB	1,000,000	1	antonio-MS-7B51	2020-05-28 18:06:56	1s
3	12 B	1	7.63 MB	1,000,000	1	antonio-MS-7B51	2020-05-28 18:06:57	847ms

Figura 3.5: Información sobre cada tarea desglosada

es el tercero, se mostrará su información en las siguientes figuras. Una vez se accede, aparecerá la ventana recogida por la figura 3.4. En esta imagen se puede observar información básica como el nivel de paralelismo, fechas de inicio y fin, número de registros recibidos, etc.

La información más relevante se encuentra en las pestañas de *SubTasks* mostrada en la figura 3.5 y *TaskManagers*, mostrada en la figura 3.6. Estos apartados contienen las siguiente información:

- ***SubTasks***: contiene información sobre cada tarea individual (i.e. partición), en la que se puede observar que, como se indicó por parámetro al llamar a la función, cada partición ha generado 1 millón de elementos. Además contiene otro tipo de información relevante, como el número de intentos necesarios, el tamaño de bytes enviados, la duración, etc.

- ***TaskManagers***: este es el apartado más interesante, ya que muestra cómo se han distribuido las tareas en función de los *TaskManagers* disponibles. Como se puede observar en la figura 3.6, existen tres *TaskManagers*, como se explicó en el ejemplo. Al ser 4 particiones las enviadas, uno de los *TaskManagers* ha tenido que procesar una partición extra, pero en ningún caso una de las particiones se ha llevado todo el trabajo.

Como se ha podido observar en las imágenes, se ha conseguido implementar la

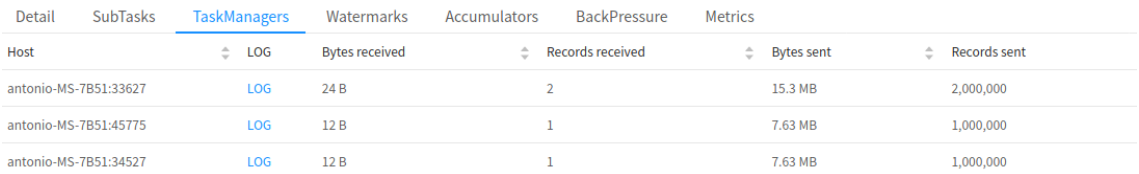


Figura 3.6: Información sobre el reparto de las tareas entre los *TaskManagers*

```
def generateDataSetTableGenerator[A: ClassTag : TypeInformation]
   (numElements: Int, numPartitions: Int, g: Gen[A], seedOpt: Option[Int] =
       None, auto_increment: Boolean = false)
   (implicit tEnv: BatchTableEnvironment, env: ExecutionEnvironment =
       ExecutionEnvironment.getExecutionEnvironment): Gen[Table] = {
for {
 d <- generateDataSetGenerator(numElements, numPartitions, g, seedOpt)
} yield
 if (auto_increment)
   tEnv.fromDataSet(d.zipWithIndex '_1, '_2).orderBy('_1)
 else
   tEnv.fromDataSet(d)
}
```

Figura 3.7: Función de creación de generador para `Table`

escalabilidad y el balanceo de carga de manera exitosa. Además, la interfaz que provee Flink es muy cómoda y proporciona una gran cantidad de información, la cual sería mucho más complicada de encontrar si se tuviese que usar una terminal para dicho proceso.

3.5. Generadores de `Table`

Una vez implementados los generadores de `DataSet`, se decidió extender esta herramienta para que pudiese soportar generadores de tipo `Table` [32]. Es interesante añadir esta funcionalidad, ya que como se explicó previamente, proporciona una mayor abstracción al usuario y se asemeja al lenguaje SQL, el cual es ampliamente conocido. Además, los objetos `Table` son compatibles con los `DataSet`, pudiendo transformar de un tipo a otro y viceversa.

En las siguientes sub-secciones se incidirá en la estructura de la función para crear los generadores de tipo `Table` y sus características.

3.5.1. Estructura general

La implementación del generador de `Table` es relativamente sencilla una vez se ha creado el generador de `DataSet`. La estructura de la función se muestra en la figura 3.7.

La definición de la función es muy parecida a la del generador de `DataSet`, ya que todos los parámetros que se le pasaban a la función de `DataSet` se le pasan también

a la función de `Table`. La diferencia reside en que para el generador de `Table` hay dos parámetros nuevos:

- `auto_increment` indica a la función si el generador de `Table` va a añadir una columna con un identificador auto-incremental en los objetos `Table` generados o no.
- `tEnv` al igual que `env`, es un parámetro implícito que representa el entorno necesario realizar operaciones con objetos `Table`. Entre otras funcionalidades permite crear UDF para este tipo de objetos, transformar de `Table` a `DataSet` y viceversa o registrar objetos `Table`.

El proceso que se realiza en la función es muy sencillo ya que, primero se obtiene un `DataSet` mediante un generador de `DataSet` como se muestra en la línea 5, y luego dicho `DataSet` se transforma a un objeto de tipo `Table` usando el entorno definido en `tEnv`. Por último, en caso de que `auto_increment` valga `true`, se asignará una columna a modo de identificador auto-incremental mediante el uso de la función `zipWithIndex` [33] y se ordenarán el resto de columnas en función de dicha columna, ascendentemente. Una vez más, se utiliza la estructura `for`/`yield` para que el resultado sea un generador. La función devuelve un objeto `Gen[Table]`.

Aunque en el código de la figura 3.7 no se utilice la variable `ExecutionEnvironment` definida como implícita, es necesaria porque la función `generateDataSetGenerator` sí que la utiliza.

Capítulo 4

Property-based testing usando generadores

Una vez implementada la funcionalidad del generador de `DataSet`, el siguiente paso es comprobar su adecuación para la definición de propiedades PBT considerando diferentes ejemplos. Para ello se han usado los programas del repositorio de GitHub de Apache Flink [34]. En concreto se ha implementado el código del repositorio de *WordCount*, TPCH10 y *K-Means* (con pequeñas modificaciones) y se han programado varios test basados en propiedades con ScalaCheck, usando los generadores. Además, se ha implementado otro ejemplo simulando el comportamiento de una herramienta ETL (*Extract, Transform, Load*) usando la estructura de datos definida por el código desarrollado para TPCH10.

Para cada programa se ha creado una especificación de specs2, la cual contiene varias propiedades usando ScalaCheck.

4.1. *WordCount*

La funcionalidad de *WordCount* consiste en, dado un `DataSet` de cadenas de texto, devolver una lista de tuplas en las que se indica la frecuencia de cada palabra del `DataSet` original. Se podría decir que este programa es el equivalente al "Hola mundo" de los motores de procesamiento masivo de datos. Es un programa sencillo, pero adecuado para empezar a diseñar las primeras propiedades y entender su funcionamiento.

Para las propiedades que se han diseñado sobre el programa de *WordCount*, se han creado cuatro generadores diferentes:

1. Generador de `DataSet` de cadenas de texto `"antonio"` mediante el uso de un `Gen.const("antonio")`. La variable se llama `genDatasetAntonio`.

2. Un generador como el del primer punto pero usando `Gen.const("juan")`. La variable se llama `genDatasetJuan`.

3. Un generador como el del primer punto pero usando `Gen.const("enrique")`. La variable se llama `genDatasetEnrique`.

```
1 "Count total elements joining 3 datasets and check it is the same compared to
      the flink wordcount program" >>
2    Prop.forAll(genDatasetAntonio, genDatasetEnrique, genDatasetJuan){
3    (d1: DataSet[String], d2: DataSet[String], d3: DataSet[String]) =>
4      val totalCount = d1.count() + d2.count() + d3.count()
5      val allDatasets = d1.union(d2.union(d3))
6      WordCount.wordCountDataSetCalc(allDatasets).map{case (_,t2) => t2}.sum
           must_== totalCount
7
8  }.set(minTestsOk = 50)
```

Figura 4.1: Definición de propiedad básica en *WordCount*

4. Por último, se ha creado un generador de `DataSet` de cadenas aleatorias, usando como parámetro un `Gen.alphaStr`. La variable se llama `genDatasetRandomStrings`.

Se ha llamado a la función `generateDataSetGenerator` para crear estos generadores, y se han usado diferentes valores para los parámetros relacionados con el número de elementos y el número de particiones.

Aunque la intención de esta sección es explicar las diferentes propiedades que se han creado para comprobar cómo se puede abarcar el testeo de un programa, en vez de mostrar su código, se expondrán algunas figuras con las propiedades más interesantes. Antes de eso, es necesario mostrar una propiedad sencilla para explicar la estructura que van a tener todas las propiedades, tanto de *WordCount* como del resto de programas. Para ello, la figura 4.1 muestra una propiedad sencilla sobre el programa *WordCount* usando los tres generadores que reproducen `DataSet` de cadenas constantes.

El funcionamiento de la propiedad es muy sencillo. Se pasan por parámetro los tres generadores, que reproducen para cada test de la propiedad un `DataSet` (los parámetros definidos como `d1`, `d2` y `d3` en la figura). Se obtiene la suma total de sus tamaños, sumando el resultado de la función `count` en cada uno de ellos. Una vez obtenido este total, se crea un `DataSet` resultante de la unión de los anteriores, el cual se pasa como parámetro a la función `wordCountDataSetCalc`. Esta función es la responsable de ejecutar el código *WordCount* sobre el `DataSet`, devolviendo una lista de tuplas con cada cadena de texto y su frecuencia (en este caso, para las cadenas: `"enrique"`, `"juan"` y `"antonio"`). A partir de esta lista de tuplas, se suman las repeticiones de las tres palabras, y por último, se comprueba que ese número es siempre igual al cálculo realizado antes aplicando la función `count`, mediante el `Matcher must_==` que proporciona specs2.

A nivel estructural, esta propiedad no es muy diferente al resto. En la línea 1 de la figura 4.1, está definido el texto que explica la funcionalidad que va a medir la propiedad. Cuando se ejecuta el test, se escribe por consola dicha cadena de texto para indicar que la propiedad ha sido exitosa. En la línea 2 se pasa como parámetro de `Prop.forAll` los generadores que va a usar la propiedad. La siguiente línea contiene las variables `d1`, `d2` y `d3` que son variables generadas por los generadores de la línea 2 (i.e: de un `Gen[DataSet[String]` se obtiene un `DataSet[String]`). Los datos de estas variables cambiarán en cada test de la propiedad, consiguiendo así que los test sean

con valores aleatorios. La función `Prop.forAll` debe devolver un objeto `Prop`. Para ello se utiliza un `Matcher` en la línea 6. Por último, la línea 8 define cuántos test tienen que ser superados para que se cumpla la propiedad.

La siguiente propiedad implementada es, realmente, una sutil adaptación de la propiedad de la figura 4.1. En esta propiedad se realizan exactamente los mismos pasos, pero a la hora de comprobar el total de elementos obtenidos por la función `wordCountDataSetCalc` con el valor de `totalCount`, se le resta a `totalCount` el número de elementos de una de las particiones de un generador y se utiliza el `Matcher must_!=`, que indica que los valores tienen que ser siempre diferentes.

La última propiedad creada para esta especificación es la más interesante de todas. Su funcionalidad radica en el uso del generador `genDatasetRandomStrings` para obtener diferentes cadenas de texto de diferentes longitudes. Una vez generado el `DataSet[String]`, se eliminan posibles duplicados usando la función de Flink `distinct` y, por cada cadena de texto, se genera esa misma cadena tantas veces como número de caracteres tiene (es decir, tantas cadenas como la longitud de la misma). Primero se utiliza la función `map` para crear una lista con el cálculo explicado por cada cadena de texto, y después, usando la función `flatmap` se realiza una transformación de `DataSet[Seq[String]]` a `DataSet[String]`. De esta manera, cuando se llama a la función `wordCountDataSetCalc` usando este `DataSet`, el resultado tiene que ser una lista de tuplas en las cuales, el valor numérico de la frecuencia tiene que ser igual a la longitud de la cadena de texto que tiene asociada.

4.2. TPCH10

TPCH [35] es un *benchmark* que utiliza consultas SQL en las que se accede y se modifican datos de manera concurrente. Es un *benchmark* diseñado para usarse con grandes volúmenes de datos.

La versión de este programa que contiene el repositorio de GitHub [34] es una versión simplificada del *benchmark* y trata la consulta número 10. Dicha consulta es la que se refleja en la figura 4.2. Como se puede observar en la cláusula `FROM` de la figura 4.2, existen cuatro tablas. Para simular su uso con el generador de `DataSet`, primero es necesario crear clases para cada una de esas tablas. La estructura de cada una de ellas es la siguiente:

- `case class Customer(name: String, address:String, nationId: Long, acctBal: Double).`
- `case class Order(custId: Long, orderDate: String).`
- `case class Lineitem(extPrice: Double, discount: Double, returnFlag: String).`
- `case class Nation(nation: String).`

Una vez se han definido las clases, es necesario crear un generador personalizado para cada una de ellas, y así poder pasar dicho generador como parámetro a la función del generador de `DataSet`. Para ello se utilizó una estructura similar a la

```
1  SELECT
2     c_custkey,
3     c_name,
4     c_address,
5     n_name,
6     c_acctbal
7     SUM(l_extendedprice * (1 - l_discount)) AS revenue,
8  FROM
9     customer,
10    orders,
11    lineitem,
12    nation
13 WHERE
14    c_custkey = o_custkey
15    AND l_orderkey = o_orderkey
16    AND YEAR(o_orderdate) > '1990'
17    AND l_returnflag = 'R'
18    AND c_nationkey = n_nationkey
19 GROUP BY
20    c_custkey,
21    c_name,
22    c_acctbal,
23    n_name,
24    c_address
```

Figura 4.2: Consulta SQL TPCH10 utilizada para testear

presentada en la figura 2.8.

Como se puede observar en la figura 4.2 existen campos de las tablas que no se han definido en las clases de Scala. Estos elementos son los identificadores de cada tabla (*c_custkey*, *o_orderkey*, *l_orderkey* y *n_nationkey*). Esto se debe a que estos valores son auto-incrementales y deben de ser únicos, lo que es complicado de gestionar mediante un generador. La solución a este problema se resuelve al obtener los `DataSet` generados en las propiedades, añadiéndole un identificador auto-incremental a cada `DataSet`.

El programa de TPCH10 que está implementado en el repositorio de GitHub de Flink se calcula mediante diferentes operaciones sobre el `DataSet` hasta obtener los datos que especifica la clausula `SELECT` de la figura 4.2. La propiedad que se ha creado parte de cuatro `DataSet` obtenidos de los generadores de las clases previamente expuestas. A dichos `DataSet` se les asigna un id auto-incremental, mediante la función `zipWithUniqueId` [33] para replicar completamente la estructura de datos que requiere el programa TPCH10 de Flink. Para comparar el resultado que devuelve dicho programa, se simula la consulta de TPCH10 mediante la API `Table` que proporciona Flink. En vez de mostrar la propiedad entera, en la figura 4.3 se puede observar el cálculo realizado para poder comparar el resultado del programa de Flink (se llama a dicha función en la línea 1) con el cálculo realizado convirtiendo los `DataSet` a tipo `Table`. Los objetos `DataSet` corresponden a las variables `dCustomerId, dOrdersId, dLineItemId` y `dNationsId` mientras que sus homónimos de tipo `Table` son

```
1   val resultTPCH = TPCHQuery10.TPCHQuery10Calc(dCustomerId, dOrdersId,
        dLineItemId, dNationsId)
2
3   val tableApiResult = tableCustomers
4     .join(tableOrders)
5     .where('c_id === 'custId)
6     .join(tableLineItems).where('o_id === 'li_id)
7     .join(tableNations).where('nationId === 'n_id)
8     .where(toIntFunct('orderDate.substring(0, 5)) > 1990 && 'returnFlag ===
          "R")
9     .groupBy('c_id, 'name, 'acctBal, 'nation, 'address)
10    .select('c_id, 'name, 'address, 'nation, 'acctBal, ('extPrice *
          ('discount - 1).abs()).sum.ceil() as 'revenue)
11
12  val datasetApiResult = tEnv.toDataSet[(Long, String, String, String,
        Double, Double)](tableApiResult)
13
14  datasetApiResult must flink.DataSetMatchers.beEqualDataSetTo(resultTPCH)
```

Figura 4.3: Propiedad TPCH10

las variables `tableCustomers, tableOrders, tableLineItems` y `tableNations`.

En las líneas 4, 5, 6 y 7 se aplican las uniones necesarias entre las tablas, como se indica en las líneas 14, 15 y 18 de la figura 4.2. En la línea 8 se realiza la cláusula `WHERE` en la que se comprueba que el año sea mayor a 1990 y que la bandera valga 'R'. Como se puede observar, existe una función `toIntFunct` que es una función definida por el usuario (UDF) necesaria para aplicar una transformación a la cadena para que sea de tipo entero. Por último, en las líneas 9 y 10 se aplican las cláusulas `GROUP BY` y `SELECT`, con la operación necesaria para calcular `revenue`. Como se puede observar, la sintaxis es muy parecida a la del lenguaje SQL, lo que facilita al usuario la programación ya que la abstracción es alta.

En la línea 12 se transforma `tableApiResult` para que sea un `DataSet` y en la última línea se aplica el `Matcher` para comparar ambos `DataSet`. El `Matcher` utilizado es `flink.DataSetMatchers.beEqualDataSetTo`, que es uno de los `Matcher` creados en el proyecto (ver sección 5).

4.3. *K-Means*

K-Means [36, 37] es un algoritmo iterativo de *clustering* basado en las distancias, que consiste en lo siguiente. Dado conjunto de puntos y un conjunto inicial de `k` centroides, cada punto es asignado a su centroide más cercano. Un centroide se podría definir como el centro de un *cluster*. Sin entrar mucho en detalle, dado un número de centroides (`k`) iniciales, los pasos que ejecuta el algoritmo son los siguientes:

1. Del conjunto de datos, seleccionar `k` puntos para que sean los centroides de manera aleatoria.

2. Asignar cada punto a su centroide más cercano.

3. Volver a calcular los centroides en función de la asignación de los puntos.

4. Repetir los pasos 2 y 3 el número de veces asignado en el programa o hasta que converja la asignación de puntos.

El programa implementado en la biblioteca de GitHub de Flink [34] opera con puntos bidimensionales, así que la propiedad implementada se ha creado siguiendo esta condición. Además, el programa define las clases `Point` y `Centroid`, teniendo ambas como atributo las coordenadas `x` e `y`. La clase `Centroid` también posee un atributo `id`, el cual corresponde al identificador del *cluster* al que representa. Por último, para calcular la distancia entre puntos se utiliza la distancia euclídea.

La propiedad diseñada se basa en los siguientes puntos:

1. Crear `k` centroides a lo largo de una recta del plano con una distancia `d` entre centroide y centroide.

2. Una vez creados los centroides, se crea alrededor de cada uno de ellos un conjunto de puntos, siendo la distancia más grande entre un punto y su centroide inicial `d/2` (siendo `d/2` exclusivo). De esta manera, los puntos creados siempre van a ser más cercanos dicho centroide, consiguiendo así que la separación entre los *clusters* sea evidente.

3. Cuando ya se hayan creado todos los puntos y centroides, se llama al programa de Flink para que recalcule los centroides.

4. Debido a que el conjunto de datos en su totalidad se ha creado para que los *cluster* estén lo suficientemente alejados unos de otros, la propiedad comprueba que los centroides iniciales siempre pertenecerán a su clase inicial al final de la ejecución del programa.

Para crear la propiedad se han usado los siguientes generadores:

- `initialCentroidGen` es un generador que reproduce los centroides iniciales.
- `numCentroidsGen` es un generador de enteros que especifica el número de centroides que va a utilizar cada test de la propiedad.
- `iterationsGen` genera el número de iteraciones que ejecutará el programa *K-Means* de Flink.
- `centroidsDistanceGen` genera la distancia entre centroides.

Como el código de la propiedad implementada para *K-Means* es más matemático que las propiedades previamente expuestas, y por ende, no contiene nada remarcable respecto funciones de Flink y ScalaCheck, la figura 4.4 muestra de manera gráfica los pasos que sigue el código de la propiedad, en vez del código en si.

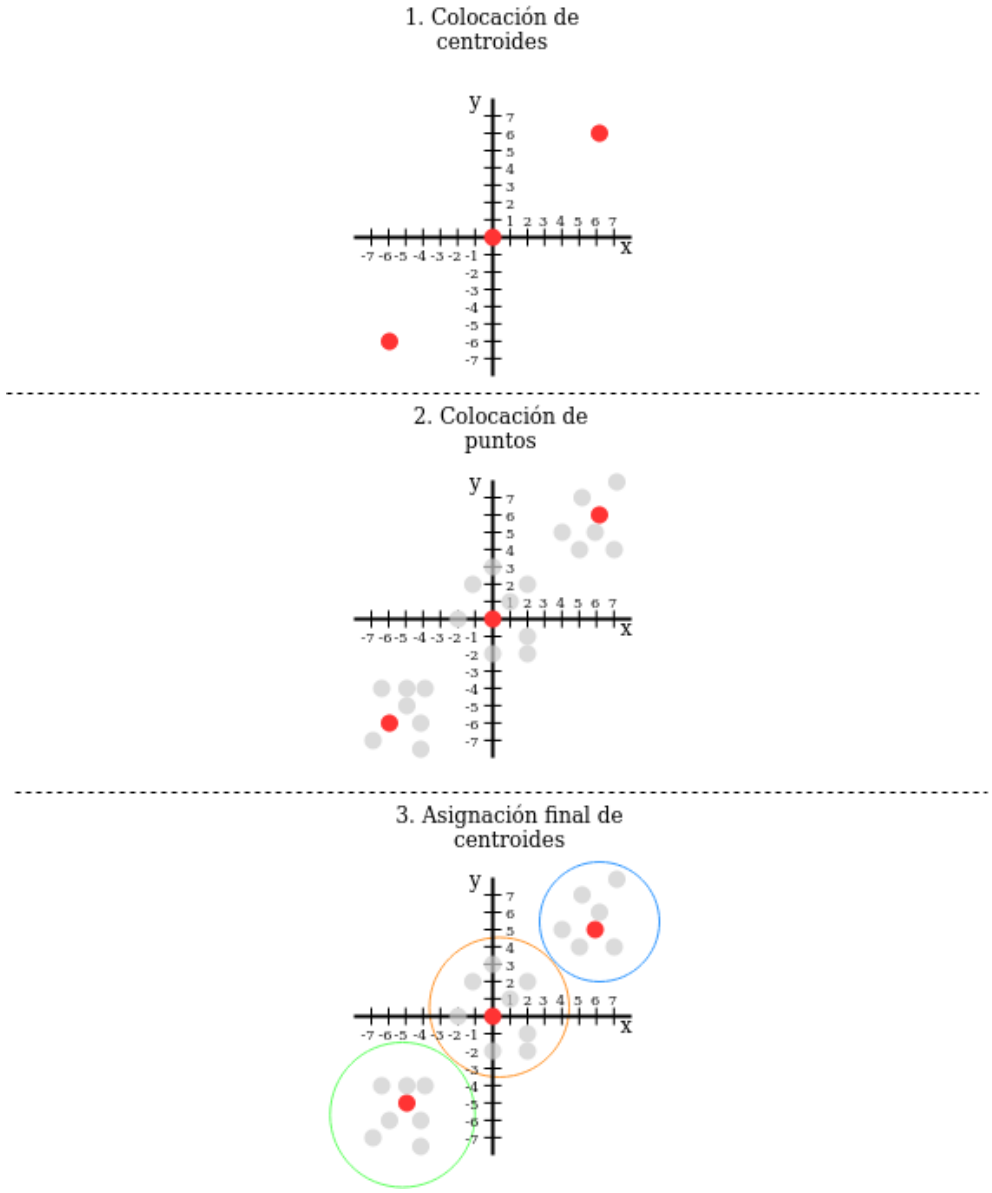


Figura 4.4: Elaboración de la propiedad para *K-Means*

4.4. ETL

El concepto de ETL [38] (*Extract, Transform, Load*) se aplica a las herramientas que recogen datos de diferentes fuentes, los procesan y, por último, los almacenan en una base de datos centralizada.

Esta sección difiere de las anteriores, ya que las propiedades creadas no se basan en un programa de la biblioteca de Flink, sino que se ha simulado el comportamiento de dicha herramienta partiendo de la clase `Customer` definida para la sección 4.2. La herramienta ETL que se ha simulado tiene las siguientes características:

- La fuente de datos de la que se parte son ficheros CSV siendo cada columna uno de los atributos de la clase `Customer`.

- Es necesario excluir las cabeceras de los ficheros CSV.
- Es necesario que el número de datos por cada fila sea igual a cuatro (`name`, `address`, `nationId` y `acctBal`).
- Los tipos de los datos tienen que ser correctos (e.g. `name` tiene que ser una cadena de texto y no un valor numérico).
- Es necesario excluir las filas de datos en las que alguno de los campos numéricos, su valor sea `NaN` o `NA`.
- Los valores de `acctBal` deben tomar un valor entre 0.0 y 1000.0.

Para simular este comportamiento se han seguido los siguientes pasos:

1. Crear un generador de `DataSet` de objetos `Customer` con datos válidos para la ETL. El generador se llama `validCustomerGen`.
2. Crear varios generadores para reproducir datos no válidos para la ETL. Esos generadores son los siguientes:
   - `invalidCustomerGen` genera `DataSet` con valores para `acctBal` que no cumplen el rango establecido.
   - `headersNumberGen` genera un número aleatorio que indica cuántas cabeceras se van a añadir (existe la posibilidad de que una ETL lea más de un CSV, asi que existirán varias cabeceras). La cabecera tiene el siguiente formato: `"Name, Address, NationId, AcctBal"`.
   - `incorrectFieldNumberGen` genera un entero especificando cuántas filas de datos van a tener más o menos campos de los requeridos.
   - `parseErrorGen` genera una lista a partir del generador `Gen.oneOf("NaN", "NA", "Double", "Int")`. Por cada elemento de esa lista se genera una fila de datos que tiene una estructura incorrecta. En caso de que una posición valga `NaN` o `NA`, la fila de datos generada será `"Nombre,Calle,NaN,NaN"` o `"Nombre,Calle,NA,NA"` respectivamente. En caso de valer `Double`, se asignará un número decimal al campo `NationId` cuando este debería ser entero. Por último, si la posición vale `Int`, se generará una fila de datos en la que el campo `acctBal` sea un entero.
3. Transformar los `DataSet` de `Customer` a `DataSet` de `String` para simular el formato CSV (e.g: `"Mario, Calle Santo Domingo, 4, 45.65"`).
4. Aplicar la función `union` para unir todos los `DataSet` en uno solo.
5. Aplicar el código definido por la función representada en la figura 4.5 que se encarga de eliminar todas las filas de datos que no cumplan las condiciones establecidas y transformar cada cadena de texto en un objeto `Customer`. La función devuelve un `DataSet` de objetos `Customer`.

```
1    def checkCustomerETLProperties(customerStringDataset: DataSet[String],
         rangeValidation: Boolean = true, dataTypeValidation : Boolean = true):
         DataSet[Customer] = {
2
3    customerStringDataset
4      .filter {
5       _ != "Name, Address, NationId, AcctBal"
6      }
7      .map { xs => xs.split(',') }
8      .filter { xs => if (dataTypeValidation) xs.length == 4 && validTypes(xs)
           else xs.length == 4 }
9      .map{ xs => Customer(xs(0), xs(1), xs(2).toInt, xs(3).toDouble) }
10     .filter { xs => if (rangeValidation) validRanges(xs) else true }
11   }
```

Figura 4.5: Validación de la herramienta ETL

```
1  checkCustomerETLProperties(wholeDataset, dataTypeValidation=false)
2     .collect() must throwA[Exception]
```

Figura 4.6: Lanzamiento de excepción usando el `Matcher` proporcionado por specs2

Para este proceso se han creado tres propiedades. La diferencia entre cada una de ellas radica en la función definida por la figura 4.5. La primera propiedad realiza la validación al completo, es decir, se eliminan todas las filas de datos no válidas. Para ello primero se elimina la cabecera, como se indica en la función `filter` de las líneas 4-6. Una vez eliminados los datos que corresponden a la cabecera, se separan los campos, como indica la línea 7. En la línea 8 se eliminan todas las filas de datos que tengan un número de campos diferente a 4 y las que no cumplan la función `validTypes`. Esta función realiza la comprobación de que los valores tienen el tipo adecuado. Por último, una vez transformada la cadena de texto a un objeto de tipo `Customer`, se comprueba que los rangos de `acctBal` se cumplen llamando a la función `validRanges`. La propiedad compara el `DataSet` obtenido con el que se obtiene del generador `validCustomerGen`. Ambos `DataSet` deberían ser iguales ya que el resto de datos no válidos deben haber sido eliminados.

Las dos propiedades restantes omiten las validaciones de tipo y de rango, respectivamente. Para ello se utilizan los parámetros `rangeValidation` y `dataTypeValidation` (línea 1 de la figura 4.5). En el caso de la propiedad que no válida los rangos se comprueba que el `DataSet` resultante es diferente al generado por `validCustomerGen`.

Es más interesante la propiedad que no válida los tipos, ya que al intentar transformar uno de los atributos de la clase `Customer` al tipo correspondiente, el programa fallará y lanzará una excepción. Specs2 permite controlar este tipo de situaciones. Para ello la propiedad se cumple cuando al recoger los datos generados por la función `checkCustomerETLProperties` se lanza una excepción como se muestra en la figura 4.6.

```
1  def wordCountTableCalc(tableSample: Table, desired_frequency: Long)(implicit
       tEnv: BatchTableEnvironment): DataSet[WC] = {
2
3    val result = tableSample
4      .groupBy('word)
5      .select('word, 'frequency.sum as 'frequency)
6      .filter('frequency === desired_frequency)
7      .toDataSet[WC]
8
9    result
10  }
11
12 case class WC(word: String, frequency: Long)
```

Figura 4.7: Programa *WordCount* para la API `Table` de Flink

4.5. Propiedades para el generador de `Table`

Para testear la funcionalidad del generador de objetos `Table` se han implementado dos propiedades ScalaCheck.

La primera de ellas es para comprobar que el uso de semillas, tanto en el generador de `DataSet` como en el generador de `Table`, funcionan de la misma manera y, en caso de usar la misma semilla para crear un generador de ambos tipos, los datos generados serán idénticos aunque la estructura sea diferente. Esto se debe a que los generadores de objetos `Table` se forman a partir de los generadores de `DataSet`. Para esta propiedad se utiliza un generador de enteros, el cual generará semillas en cada test de la propiedad. Dicha semilla se usará en un `Gen[DataSet[A]` y un `Gen[Table]` devolviendo un objeto `DataSet[A]` y un objeto `Table`. Este último se transforma en un `DataSet[A]`, siendo `A` el tipo formado por todas las columnas, y se comparan ambos `DataSet`. Para ello se utiliza el `Matcher flink.DataSetMatchers.beEqualDataSetTo` que, como se explicará en detalle en el capítulo 5, es un `Matcher` personalizado para comparar `DataSet` de manera eficiente.

La segunda propiedad es sobre otro programa de *WordCount* extraído del repositorio de ejemplo de Flink [34], pero en esta ocasión, la versión diseñada para la API `Table` que muestra la figura 4.7. El ejemplo es una pequeña variante del ejemplo típico de *WordCount*. En este caso, dado un objeto `Table` con dos columnas (`word` y `frequency`), y una variable `desired_frequency`, se calcula la frecuencia de cada palabra y se devuelve un `DataSet` de objetos `WC` con todas las palabras que tengan una frecuencia igual al valor de `desired_frequency`.

La propiedad (figura 4.8) se basa en la competición entre dos objetos `Table` reproducidos por dos `Gen[Table]` creados con la función explicada en este capítulo. La propiedad recibe por parámetro dos generadores de números enteros, que van a indicar el número de elementos creados por cada partición, de cada objeto `Table`. Como se puede observar en las líneas 4 y 5 de la figura 4.8, los objetos tienes como palabras `"foo"` y `"bar"` respectivamente.

Una vez creadas las variables `Table`, se unen mediante la función `unionAll` y se escoge como palabra ganadora la que tenga un mayor número de elementos, como se

```
1  "The Table val with a greater number of elements will be the one returned by
       Word Count program " >> Prop.forAll(genElementst1, genElementst2) {
2    (elementsT1: Int, elementsT2: Int) =>
3     (elementsT1 != elementsT2) ==> {
4       val t1 = createTableGenerator("foo", elementsT1).sample.get
5       val t2 = createTableGenerator("bar", elementsT2).sample.get
6       val tUnion = t1.unionAll(t2)
7       var greater = "foo"
8       if (elementsT1 < elementsT2){
9         greater = "bar"
10      }
11
12      val winnerWordDataSet = WordCount.wordCountTableCalc(tUnion,
            Math.max(elementsT1, elementsT2)*partitions)
13      winnerWordDataSet.collect().head.word == greater
14
15    }
16  }
```

Figura 4.8: Propiedad para el programa *WordCount* implementado para API `Table` de Flink

puede observar en las líneas 6-10. Por último, se llama a la función `wordCountTableCalc` pasando la variable con los objetos `Table` unidas y como frecuencia deseada, el máximo entre los números enteros generados multiplicado por el número de particiones usadas para crear los generadores. Como resultado tiene que devolver siempre la palabra con mayor frecuencia, almacenada en la variable `greater`.

4.6. Resultados

Una vez creadas y testeadas todas las propiedades definidas en las secciones anteriores, podemos extraer las siguientes conclusiones:

- Se han podido definir todas las propiedades de manera exitosa usando los generadores. Se han creado tanto propiedades positivas (en las que se establece una igualdad) y negativas (como lanzar una excepción o la diferencia entre objetos), con lo cual, son una buena herramienta para usar con propiedades ScalaCheck.
- La eficiencia de las propiedades creadas no es la mejor cuando es necesario comparar el contenido de dos `DataSet`, ya que es necesario usar el comando `collect`. La solución a este problema son los `Matcher` presentados en el capítulo 5. De todas formas, las propiedades no se ejecutan de manera inmediata, en general, ya que cada una de ellas realiza muchos test, lo que conlleva crear nuevos datos de entrada por cada uno de ellos.
- El uso del generador es muy flexible para poder definir propiedades, ya que se pueden generar datos de los tipos que se desee, siempre y cuando sean serializables.

Por último, es importante indicar que para ejecutar cada ejemplo con SBT, es necesario ejecutar el siguiente comando: `testOnly "nombreClase"` (e.g: `testOnly WordCountTestSpecs`).

Capítulo 5

Implementación y uso de objetos `Matcher` de flink-check

Muchas de las propiedades de ScalaCheck implementadas en el capítulo 4 se definen mediante la comparación entre `DataSet` usando la función `collect`. Si el tamaño de los `DataSet` es grande, utilizar esta función es muy costoso debido a que se cargan todos los elementos en una lista, fuera del *cluster* de Flink (se desaconseja el uso en la propia documentación de Apache Flink).

Para solucionar este problema, se ha usado una funcionalidad que proporciona specs2 para crear `Matcher` personalizados en la sección *Matchers* de su documentación [20]. Para poder crear un `Matcher` es necesario importar el contenido de `org.specs2.matcher.MatchersImplicits._` y que la función devuelva una tupla formada por un valor booleano y una o más cadenas de texto. Si es solo una cadena de texto, su contenido debe de indicar la razón del fallo (si lo hubiese) del `Matcher`. Si se definen dos cadenas de texto, la primera contiene el mensaje de fallo (la razón por la que no se ha cumplido la condición establecida en la función del `Matcher`) y la segunda la del mensaje de éxito (que puede contener una información más detallada sobre la condición que se ha cumplido).

En el repositorio GitHub del proyecto flink-check [2] existe un fichero que contiene algunas funciones que devuelven variables `Matcher` que se pueden usar sobre objetos `DataSet`. Aun así, dichos `Matcher` no eran suficientes para este proyecto y se implementaron nuevos `Matcher` personalizados. Tanto los `Matcher` proporcionados por la biblioteca flink-check, como los `Matcher` implementados, se describirán en las siguientes secciones.

5.1. `Matcher` proporcionados por flink-check

En el fichero que contiene los `Matcher` del proyecto flink-check existen varios `Matcher` personalizados definidos en funciones. De todos ellos, se han usado en el proyecto los siguientes:

- `beEmptyDataSet` y `beNonEmptyDataSet` comprueban si un `DataSet` esta vacío o no.

```
1  def beSubDataSetOf[T : TypeInformation : ClassTag : Ordering]
2     (other: DataSet[T], strict: Boolean = true): Matcher[DataSet[T]] = {
3    data: DataSet[T] =>
4      val failingElements = new FlinkCheckDataSet(data).minus(other,
           strict).first(numErrors)
5      (
6       failingElements.count() == 0,
7       "this data set is contained in the other",
8       s"these elements of the data set are not contained in the other
9         ${failingElements.collect().mkString( ", " )} ..."
10     )
11 }
```

Figura 5.1: `Matcher` para comprobar si un `DataSet` es subconjunto de otro

- `beSubDataSetOf` comprueba si el `DataSet` que se pasa por parámetro contiene al `DataSet` que invoca al `Matcher`.

Esta última función es la más importante, ya que es la más cercana al objetivo de comprobar que dos `DataSet` son iguales, ya que la igualdad entre dos `DataSet` se puede establecer a que son iguales si, y solo si, el primero contiene al segundo y el segundo contiene al primero. Para entender cómo se implementó el `Matcher` de igualdad, primero es necesario entender el `Matcher beSubDataSetOf` expuesto en la figura 5.1.

En la primera línea, además de añadir `TypeInformation` como `ClassTag` para el tipo genérico `T`, es necesario también `Ordering`. Esto se debe a que, como se explicará en otra figura más adelante, es necesario establecer un orden entre los elementos de los `DataSet` para poder indicar si uno el `DataSet data` es subconjunto de `other`. La otra variable pasada como parámetro es `strict`, la cual está relacionada con el tipo de comparación que se quiere realizar. Se explicará de manera más detallada en la sección 5.2 ya que fue uno de los cambios implementados sobre el fichero que contiene los `Matcher` de flink-check.

En la línea 4 se convierte la variable `data` a un tipo de la clase `FlinkCheckDataSet` definida en el fichero. Dicha clase contiene la función `minus`, que a su vez llama a `minusWithInMemoryPartition`, y esta simula la operación de diferencia entre conjuntos usando `data` y `other` respectivamente. El contenido de esta función es la parte más importante, ya que es la encargada de optimizar las operaciones necesarias para comprobar si se cumple la operación. Esta función devuelve cuántos elementos deberían estar incluidos en `data` o vacío en caso de que se cumpla la condición. La variable `numErrors` es una variable global que define cuantos elementos almacenar de la respuesta de la función, debido a que si el número es muy alto, es muy caro llamar a `collect`.

Por último, la línea 6 comprueba si la función `minus` ha devuelvo elementos o no y, en función de la respuesta, crea el `Matcher`. Si a `data` le faltan elementos para poder ser subconjunto de `other`, dichos elementos se devuelven en forma de cadena de texto al usuario.

La figura 5.2 muestra la función `minusWithInMemoryPartition`. El primer paso es

```
1  private[this] def minusWithInMemoryPartition(other: DataSet[T], strict:
       Boolean): DataSet[T] = {
2    val selfMarked: DataSet[(T, Boolean)] = self.map((_ : T, true))
3    val otherMarked: DataSet[(T, Boolean)] = other.map((_: T, false))
4    val all = selfMarked.union(otherMarked)
5      .partitionByHash(0)
6
7    all.mapPartition[T] { (partitionIter: Iterator[(T, Boolean)], collector:
         Collector[T]) =>
8
9      val sortedPartition = {
10       val partition = partitionIter.toArray
11       util.Sorting.quickSort(partition)
12       partition
13     }
14
15     val notContainedValues =
16        if (strict) strictComparison(sortedPartition)
17        else nonStrictComparison(sortedPartition)
18
19     notContainedValues.foreach(collector.collect)
20   }
21 }
```

Figura 5.2: Función de resta entre conjuntos, aplicada a `DataSet`

transformar ambos `DataSet` para que contenga una tupla con su valor `T` y un booleano, para poder así distinguir qué elemento pertenece a cada `DataSet`. En las líneas 4 y 5 se juntan ambos `DataSet` en uno, y luego se ejecuta la función `partitionByHash` separando así los diferentes elementos en particiones entre los diferentes *Task Slots* que utilice el entorno, pero asegurándose de que los elementos que sean iguales se manden siempre a la misma partición. Esta operación se realiza para que se distribuya la carga de trabajo que va a suponer comparar los elementos de ambos `DataSet`.

Usando una `RichMapPartitionFunction` anónima (líneas 7-19) se le asigna a cada partición dos tareas:

1. Ordenar todas las tuplas de cada partición mediante `quicksort`, de menor a mayor `T`, y en caso de que sea igual, los valores `T` acompañados de un valor booleano igual a `false` se colocarán antes que los que valgan `true`. Es en esta operación definida en las líneas 9-13 donde se necesita utilizar la clase `Ordering` para que el método de ordenación sepa como ordenar un tipo. Si el tipo es una instancia de una clase creada, es necesario sobre-escribir la función `compare` para establecer la manera de ordenación.

2. Una vez que los valores se han ordenado, en función de si se quiere realizar una comparación estricta o no, se llama a una función u otra. Antes de modificar el código del fichero solo existía la implementación de la función `nonStrictComparison`. Esta función no realiza una comparación estricta, porque el código implementado en ella no tiene en cuenta los duplicados. Esto

```
1  def beEqualDataSetTo[T : TypeInformation : ClassTag: Ordering]
2     (other: DataSet[T], strict: Boolean = true): Matcher[DataSet[T]] = {
3       data: DataSet[T] =>
4         var failingElements = new FlinkCheckDataSet(data).minus(other,
              strict).first(numErrors)
5         failingElements = failingElements.union(
6            new FlinkCheckDataSet(other).minus(data, strict).first(numErrors)
7         )
8
9         (
10          failingElements.count() == 0,
11          "this data set is equal to the other",
12          s"these elements of the data set are not contained in the other
13         ${failingElements.collect().mkString( ", " )} ..."
14        )
15    }
```

Figura 5.3: `Matcher` para comprobar si dos `DataSet` son iguales

quiere decir que si por ejemplo, se parte de un conjunto `(2, 2, 3)` y se quiere comprobar si es subconjunto de `(2, 3, 3)`, la función indicará que el primero conjunto sí es subconjunto del segundo, ya que no contempla ni el `2` duplicado del primer conjunto, ni el `3` duplicado del segundo conjunto.

Independientemente de cual de las dos funciones se ejecute, el resultado final que devuelve es un `DataSet` con los elementos que le faltan al primer `DataSet` para ser subconjunto del segundo. Si el `DataSet` es vacío significa que se cumple la condición de que el primer `DataSet` es subconjunto del segundo.

5.2. `Matcher` implementados

El objetivo real de esta sección es poder crear un `Matcher` para comparar dos `DataSet` en su totalidad. Se ha explicado el `Matcher beSubDataSetOf` debido a que, como se comentó en el apartado anterior, la funcionalidad de este es necesaria para poder implementar el `Matcher beEqualDataSetTo`, ya que dos `DataSet` son iguales si y solo si, el primero es subconjunto del segundo y viceversa, como se expone en las líneas 3 y 4 de la figura 5.3.

Una vez implementado el `Matcher`, surgió el problema de cómo se realizaba la comparación entre `DataSet` en la función `minusWithInMemoryPartition`. El problema radica en que si alguno de los elementos de un `DataSet` estaba duplicado, no se tenía en cuenta. Por tanto, si se pasaban por parámetro dos `DataSet` con los valores `(2, 3, 3)` y `(2, 2, 3)` respectivamente, el `Matcher` devolvía como resultado que ambos `DataSet` son iguales cuando realmente no es así.

Como en ningún momento se puede asegurar la ausencia de duplicados en los `DataSet` que se pasen como parámetros, se decidió implementar una función que comprobase de manera más exhaustiva y teniendo en cuenta la presencia de duplicados en los `DataSet`. Dicha función se llama `strictComparison` y cuenta el número de elementos iguales que existen en cada `DataSet`, devolviendo error en caso de que

```
"Checking beSubDataSetOf and beEqualDataSetTo" >> {
   val d1 = env.fromElements(2,3,3)
   val d2 = env.fromElements(3,2,2)
   val d3 = env.fromElements(2,3,2)
   val xs = env.fromCollection(1 to 10)
   val ys = env.fromCollection(5 to 15)
   val zs = env.fromCollection(1 to 5)

   println("Starting tests for beSubDataSetOf and beEqualDatasetTo")
   emptyDataSet must flink.DataSetMatchers.beEqualDataSetTo(emptyDataSet)
   emptyDataSet must flink.DataSetMatchers.beSubDataSetOf(xs)
   xs must flink.DataSetMatchers.nonBeEqualDataSetTo(emptyDataSet)
   zs must flink.DataSetMatchers.beSubDataSetOf(xs)
   xs must flink.DataSetMatchers.beSubDataSetOf(xs)
   emptyDataSet must flink.DataSetMatchers.beSubDataSetOf(xs)
   xs must flink.DataSetMatchers.nonBeEqualDataSetTo(emptyDataSet)
   xs must flink.DataSetMatchers.nonBeSubDataSetOf(ys)
   ys must flink.DataSetMatchers.nonBeSubDataSetOf(xs)
   d1 must flink.DataSetMatchers.nonBeEqualDataSetTo(d2)
   d2 must flink.DataSetMatchers.nonBeEqualDataSetTo(d1)
   d3 must flink.DataSetMatchers.beEqualDataSetTo(d2)
   d2 must flink.DataSetMatchers.beEqualDataSetTo(d3)
   d1 must flink.DataSetMatchers.beEqualDataSetTo(d2, strict = false)
   d2 must flink.DataSetMatchers.beEqualDataSetTo(d1, strict = false)
   d3 must flink.DataSetMatchers.beEqualDataSetTo(d2, strict = false)
   d2 must flink.DataSetMatchers.beEqualDataSetTo(d3, strict = false)
   println("Done tests for beSubDataSetOf and beEqualDatasetTo")
   ok
 }
```

Figura 5.4: Propiedad para comprobar el funcionamiento de `beSubDataSetOf` y `beEqualDatasetTo`

el `DataSet` que tiene que ser subconjunto, contenga un número de elementos de un valor concreto diferente al otro `DataSet`.

Por completitud y claridad de la funcionalidad de los `Matcher`, también se implementaron las funciones negativas de los `Matcher` `beSubDataSetOf` y `beEqualDataSetTo`:

- `nonBeSubDataSetOf` devuelve un valor booleano que es verdadero en caso de que falten elementos del `DataSet` que tiene que ser subconjunto sobre el otro `DataSet`.
- `nonBeEqualDataSetOf` devuelve un valor booleano que es verdadero en caso de que los `DataSet` sean diferentes.

Por último, todos las funciones contienen el parámetro opcional `strict`, que es un booleano que indica qué tipo de comparación se quiere realizar, ya que puede existir algún caso en el que se quiera eliminar los duplicados a la hora de comprobar una igualdad.

5.3. Resultados

Al implementar los `Matcher` explicados en este capítulo para poder comparar `DataSet` más eficientemente, se ha conseguido evitar el uso de la función `collect`, la cual es muy ineficiente debido a que carga todos los datos a una lista y la comparación no se realiza en el entorno de Flink. Con el desarrollo actual, la comparación se realiza de manera distribuida, repartiendo los diferentes elementos de ambos `DataSet` entre los *Task Slot* disponibles en el entorno de Flink. Es importante remarcar que en caso de que el número de elementos del `DataSet` no sea grande, en algunos casos puede ser menos eficiente usar el `Matcher` que la función `collect` para comparar las listas que devuelve dicha función.

El funcionamiento se ha probado en las propiedades explicadas en el capítulo 4, superando exitosamente todas en las que se ha invocado a los `Matcher`. Es interesante el hecho de que se puedan usar estas propiedades para poder estimar si el funcionamiento de los `Matcher` es el óptimo, debido a que los datos van variando en cada test realizado en cada propiedad. Además, se ha implementado una pequeña porción de código con casos más específicos para comprobar que se resuelve el problema de la comparación entre `DataSet` con elementos duplicados. Dichas pruebas se pueden observar en la figura 5.4 que contiene código adaptado del fichero *FlinkMatchersSpec* del proyecto flink-check. Entre otras condiciones, se comprueba el caso puesto como ejemplo en las secciones anteriores, mediante el uso de los `DataSet` definidos en las líneas 1 y 2, comprobando que no son iguales si se comparan de manera estricta (condición de la línea 19) pero sí lo son en caso de que no se comparen estrictamente (condición de la línea 23).

Capítulo 6

Elaboración de funciones de carga usando ScalaMeter

Para finalizar el desarrollo de este proyecto se ha querido analizar el comportamiento de los generadores implementados, variando el valor de los parámetros que tienen las funciones para crear dichos generadores. Para ello se ha utilizado la herramienta ScalaMeter, explicada en la sección 2.4.4. Con esta herramienta se han podido implementar unas funciones de carga que muestran el comportamiento de los generadores, en función del tiempo y del número de elementos que contiene cada `DataSet` generado. Además, se han creado otras funciones de carga para medir la eficiencia de programas Flink, los cuales usan los generadores que crea la herramienta

Este capítulo se divide en tres secciones. La primera parte consiste en una explicación de la estructura del código implementado para poder generar unas gráficas que muestren información sobre estas funciones de carga. La segunda parte se basa en diferentes pruebas realizadas sobre este código hecho en ScalaMeter para comprobar la eficiencia de generadores. Por último, se mostrará la aplicación de estas funciones de carga para probar la eficiencia de algunos programas de Apache Flink.

6.1. Estructura de las funciones de carga

Uno de los aspectos más positivos de ScalaMeter es que, de manera sencilla, permite configurar una gran cantidad de parámetros. Una de las principales razones por las que se escogió esta herramientas es porque permite crear fácilmente un fichero HTML que contiene tanto gráficas lineales, como de barras pudiendo además, mostrar un intervalo de confianza en ambos tipos. Para que se creen dichas gráficas solo es necesario extender de `Bench.OfflineReport` en el test con el que se quieran mostrar las gráficas.

Extender `Bench.OfflineReport` no solo provee al usuario de gráficas, sino que también configura el programa para que se ejecute en una JVM separada, consiguiendo que las pruebas que se realicen no se vean afectadas (en medida de lo posible) por otros procesos que se estén ejecutando mediante la JVM. De esta manera, los resultados serán más fiables.

Otras características que se han configurado para realizar las funciones de carga son las siguientes:

- `independentSamples` es el número de JVM usadas entre las que se reparte la ejecución del código. Se ha configurado para que se usen tantas como particiones se vayan a usar.
- `minWarmupRuns` representa el número de ejecuciones que va a realizar la JVM para calentar (lo que se denomina *warmup*). El calentamiento se realiza para que la información que se carga en tiempo de ejecución ya haya sido cargada en el calentamiento, y se pueda acceder a ella más rápidamente en las siguientes ejecuciones. Se han asignado dos ejecuciones para el calentamiento.
- `benchRuns` es el número de ejecuciones que se realiza por cada test. A la hora de establecer el intervalo de confianza se utilizan los valores generados por cada ejecución para obtener el máximo, el mínimo y la media. Se ha configurado para que se ejecute 30 veces cada test.

Existen muchos más parámetros configurables que se pueden encontrar en la documentación del repositorio de GitHub de ScalaMeter [39] (no es recomendable acceder a la documentación de la web de ScalaMeter ya que está desactualizada).

Cada función de carga realiza varios test sobre diferentes generadores, en función del número de elementos y particiones utilizadas, y cada test es ejecutado tantas veces como especifique el parámetro de configuración `benchRuns`. Los parámetros usados para definir las particiones y los elementos son:

- `initElements` indica el número inicial de elementos del `DataSet` generado de la función de carga.
- `incrementElements` indica el incremento de elementos del `DataSet` de cada test realizado en la función de carga.
- `iterations` indica el número de iteraciones que se van a realizar por función de carga.
- `maxPartitions` indica el número de particiones máximo que se va a usar. Cada función de carga se ejecuta con un número concreto de particiones.
- `rangePartitions` indica el aumento de particiones entre cada función de carga.

Para medir el tiempo de ejecución se ha diseñado una propiedad de ScalaCheck bastante sencilla. Esta propiedad, dado un generador de enteros entre 1 y el número de elementos que se le pase por parámetro, comprueba que el número de datos del `DataSet` eliminando sus duplicados, será siempre menor o igual al número de elementos dividido por el número de particiones que se hayan usado.

Un ejemplo de la interfaz que se ha usado para representar las funciones es la que muestra la figura 6.1. Como se puede observar, en el punto 1 se muestra cada función calculada con su respectivo número de particiones usado. El número 2 define el tipo de gráfica que se quiere visualizar y si se quiere aplicar el intervalo de confianza.

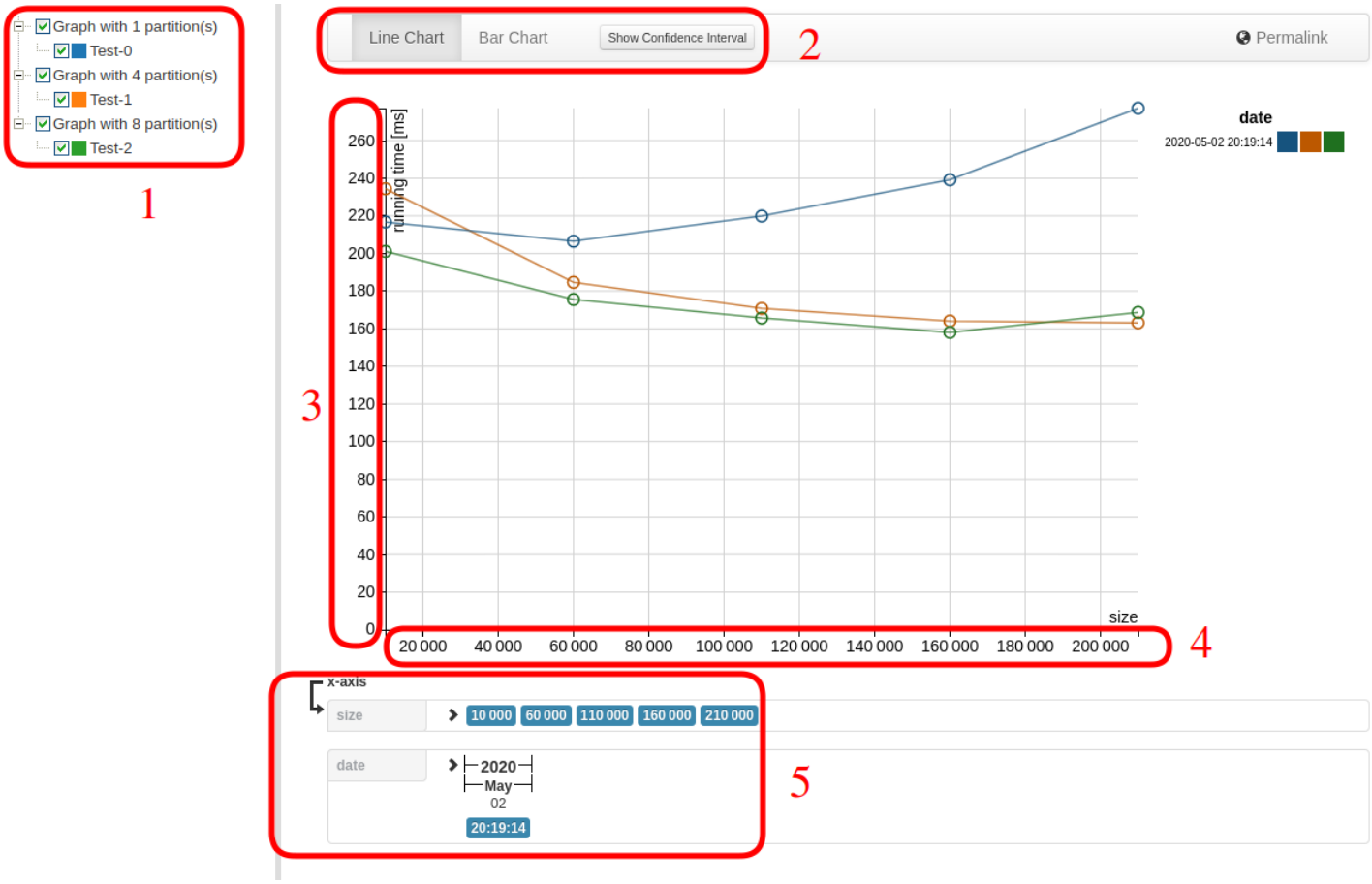


Figura 6.1: Información mostrada en la interfaz web creada por ScalaMeter

Los números 3 y 4 muestran los ejes `y` (muestra el tiempo de ejecución en ms) y `x` (muestra el número de elementos) respectivamente. Por último el cuadrante número 5 presenta valores de ambos ejes por los que se puede filtrar.

6.2. Eficiencia sobre generadores

Se han implementado cuatro test diferentes, cada uno con diferentes valores para los parámetros comentados en la sección anterior y divididos en función del número de elementos que se utiliza en cada test. En los siguientes párrafos se mostraran dichos test, los valores que se han utilizado y una descripción del resultado obtenido.

La máquina que se ha utilizado para las pruebas tiene las siguientes especificaciones:

- **Sistema operativo**: Ubuntu 18.04.
- **CPU**: Intel(R) Core(TM) i5-9600K CPU @ 3.70GHz. Contiene 6 núcleos de un hilo.
- **RAM**: 16GB.

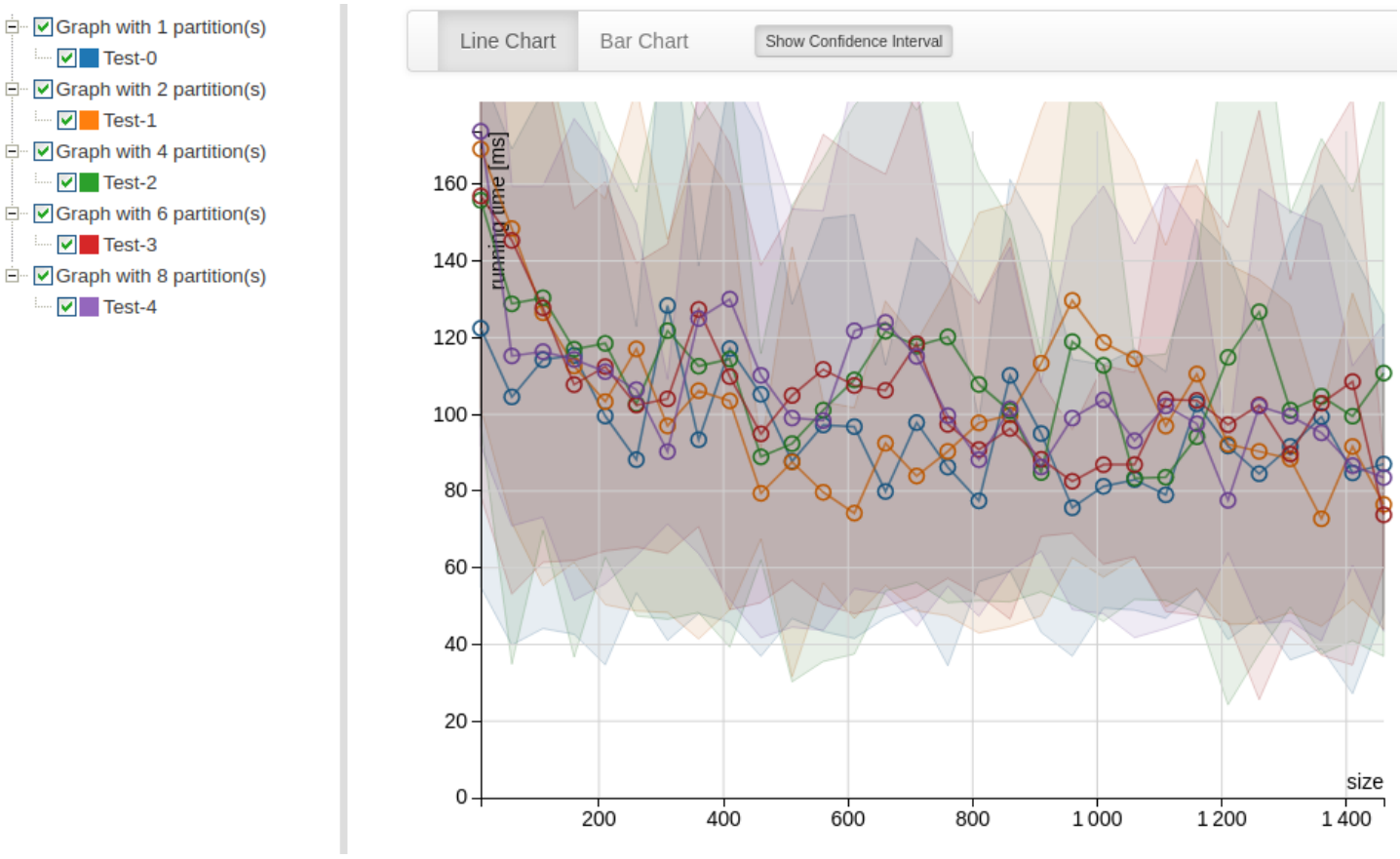


Figura 6.2: Gráfica con funciones de carga con un rango de elementos pequeño

- **Versión de Apache Flink**: 1.10.

6.2.1. *SmallGeneratorBenchmark*

Se utiliza un rango de elementos pequeño. El valor de los parámetros es el siguiente:

- `initElements` = 10
- `incrementElements` = 50
- `iterations` = 30
- `maxPartitions` = 8
- `rangePartitions` = 2

La figura 6.2 muestra una gráfica de líneas mostrando el intervalo de confianza. Al ser un número de elementos pequeños (la última medición no llega a 1.500 elementos), se puede observar como, independientemente del número de particiones, los tiempos son muy parecidos en todos los casos. Aun así, en la mayor parte de las

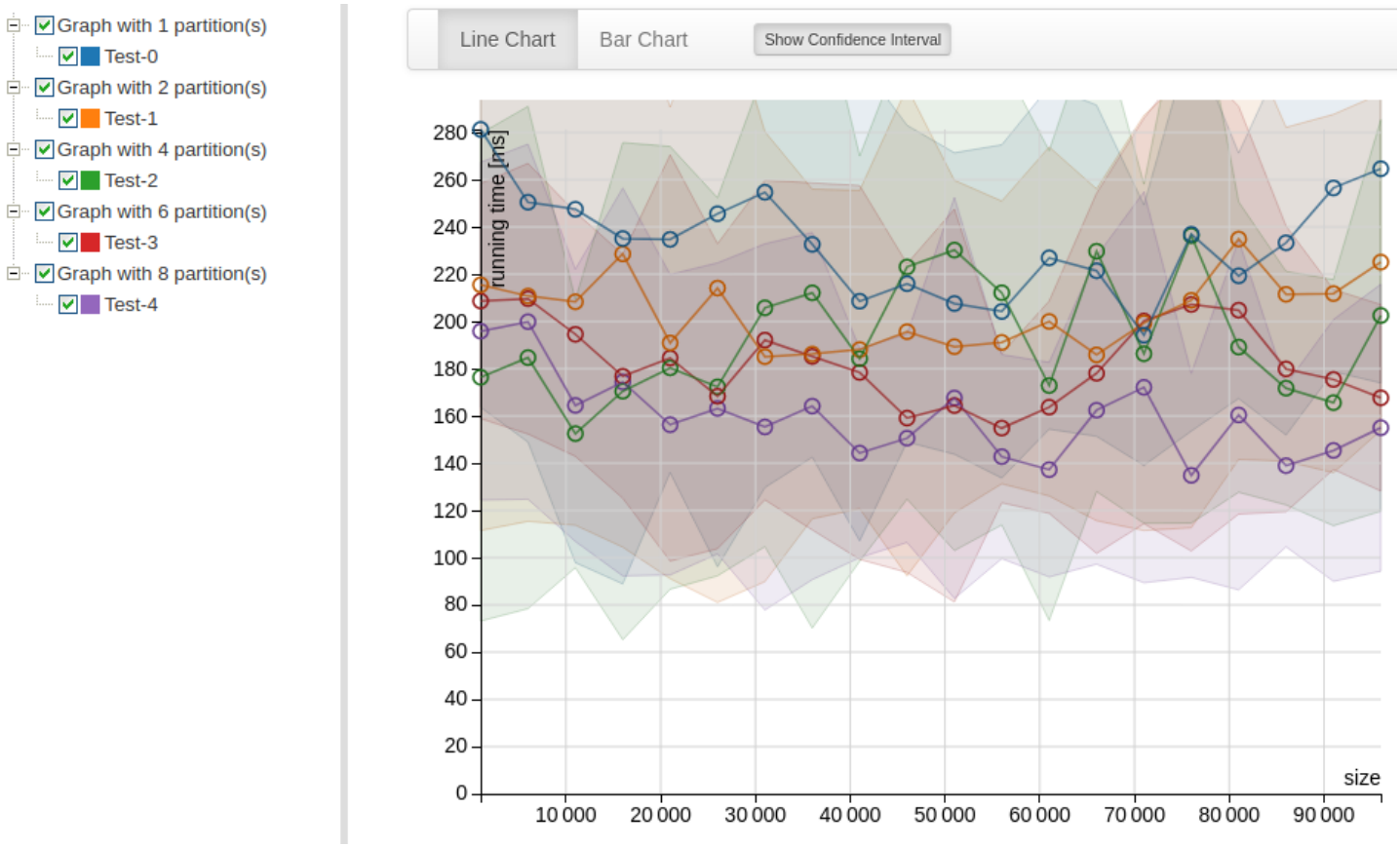


Figura 6.3: Gráfica con funciones de carga con un rango de elementos medio

mediciones que se pueden observar, la línea de color azul (equivalente a una partición) es la que menos tiempo necesita para ejecutar la propiedad en la mayoría de las muestras. Esto se debe que, al ser tan pequeño el conjunto de valores a crear, el reparto de tareas en diferentes particiones es más costoso que el propio cómputo.

Respecto al intervalo de confianza, este muestra una gran diferencia entre su máximo y su mínimo en algunas mediciones. Esto es normal debido a que el tiempo que se tarda en ejecutar la propiedad es muy pequeño. Como se verá en los test realizados con un mayor número de elementos, este intervalo se estabiliza.

6.2.2. *AverageGeneratorBenchmark*

Se utiliza un rango de elementos medio. El valor de los parámetros es el siguiente:

- `initElements` = 1.000
- `incrementElements` = 5.000
- `iterations` = 20
- `maxPartitions` = 8
- `rangePartitions` = 2

Respecto a la figura 6.3, se puede observar que las primeras mediciones (1.000, 6.000 y 11.000 elementos) todavía no son un número suficientemente alto de elementos como para que sea eficiente repartir la tarea en particiones. El intervalo de confianza sigue siendo demasiado alto, no pudiendo garantizar que alguna de las opciones vaya a ser más rápida que otra. En cambio, en las últimas mediciones (81.000, 86.000 y 91.000 elementos) los tiempos son ligeramente menores cuanto mayor número de particiones se utilice. Aun así, todavía no se puede asegurar que la ejecución usando más de una partición vaya a ser más eficiente con estos tamaños (los tiempos máximos de las gráficas con más de una partición son superiores al mínimo de la gráfica de una partición).

6.2.3. *LargeGeneratorBenchmark*

Se utiliza un rango de elementos grande. El valor de los parámetros es el siguiente:

- `initElements` = 100.000
- `incrementElements` = 50.000
- `iterations` = 20
- `maxPartitions` = 8
- `rangePartitions` = 2

En la gráfica de la figura 6.4 se puede observar una gran diferencia entre las primeras mediciones y las últimas. A partir del umbral del medio millón de elementos ya se puede observar un aumento de eficiencia de, en torno, al 50 % entre la función realizada con una partición y las funciones realizadas con seis y ocho particiones. A partir de la medición con 350.000 elementos, el mínimo del intervalo de confianza de la función realizada con una partición es superior al máximo de la siguiente función más lenta (la que usa dos particiones), así que se podría afirmar que a partir de este número de elementos merece la pena distribuir la carga de trabajo. Por otra parte, los tiempos usando cuatro, seis y ocho particiones son muy parecidos en todas las medidas tomadas en esta gráfica.

6.2.4. *HugeGeneratorBenchmark*

Se utiliza un rango de elementos muy grande. El valor de los parámetros es el siguiente:

- `initElements` = 1.000.000
- `incrementElements` = 2.000.000
- `iterations` = 5
- `maxPartitions` = 8

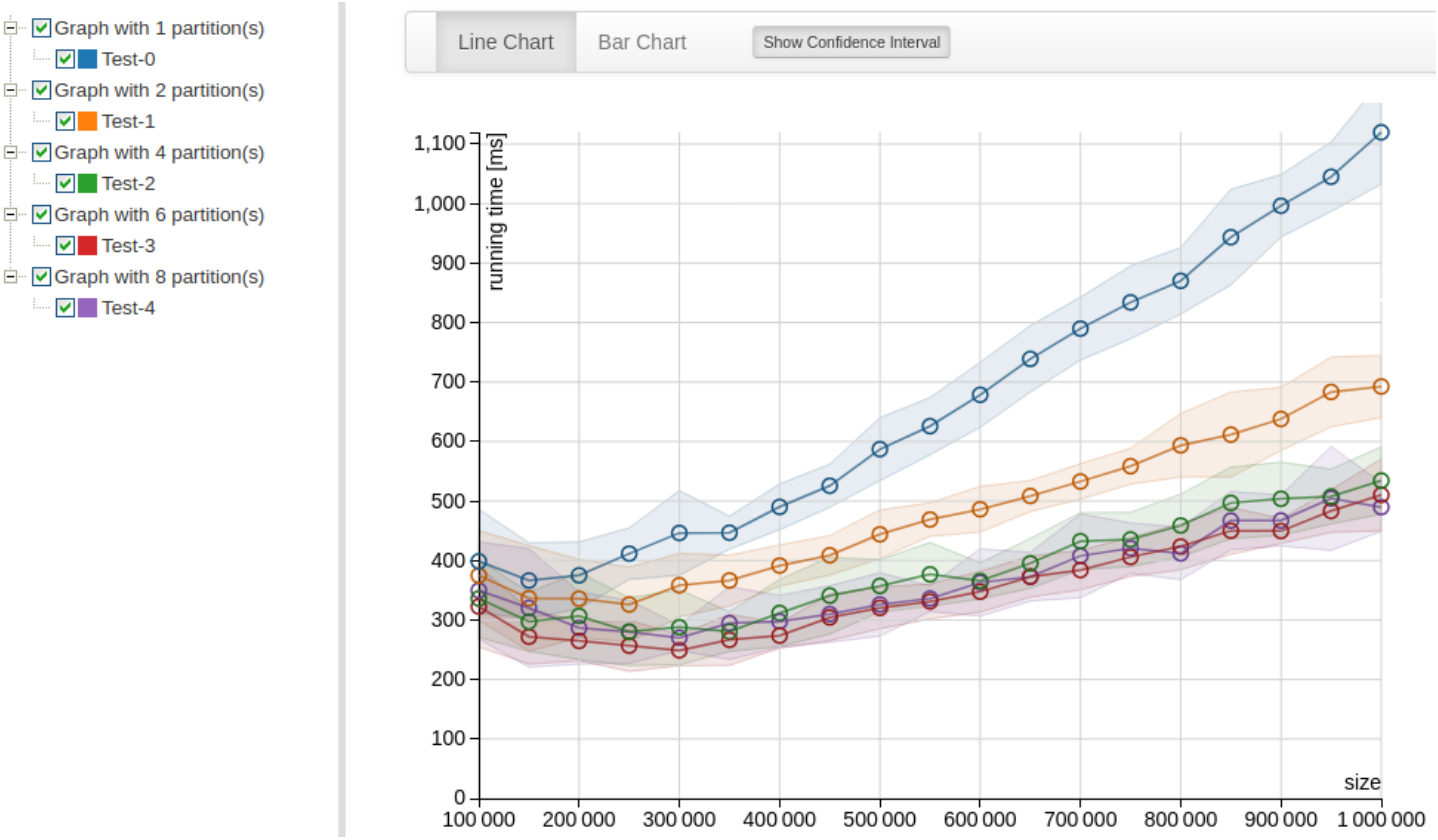


Figura 6.4: Gráfica con funciones de carga con un rango de elementos medio

- `rangePartitions` = 4

La gráfica de la figura 6.5 se ha realizado con un menor número de mediciones, debido a que la cantidad de elementos generados es muy grande (la más pequeña 1 millón de elementos) pero es un número de mediciones suficiente para sacar algunas conclusiones. La diferencia entre usar una partición o utilizar cuatro se mantiene e incrementa ligeramente (con 9 millones de elementos es un 55 % aproximadamente). En cambio, la diferencia entre usar cuatro u ocho particiones es más pequeña, aunque en cada medición la función que usa ocho particiones es cada vez más rápida en comparación con la función que usa cuatro particiones. Aun así, en ningún momento dejan de superponerse ambos intervalos de confianza.

Para concluir con esta sección de resultados, es importante matizar que todas las pruebas realizadas se han ejecutado únicamente sobre un ordenador, aunque se simulase un entorno distribuido con este. Esto quiere decir que los resultados obtenidos pueden variar en caso de que se ejecuten estas funciones de carga en un *cluster* real.

6.3. Eficiencia de programas Flink

En esta última sección del capítulo se mostrará la aplicación de los generadores y ScalaMeter para comprobar la eficiencia de dos programas realizados con Flink. El

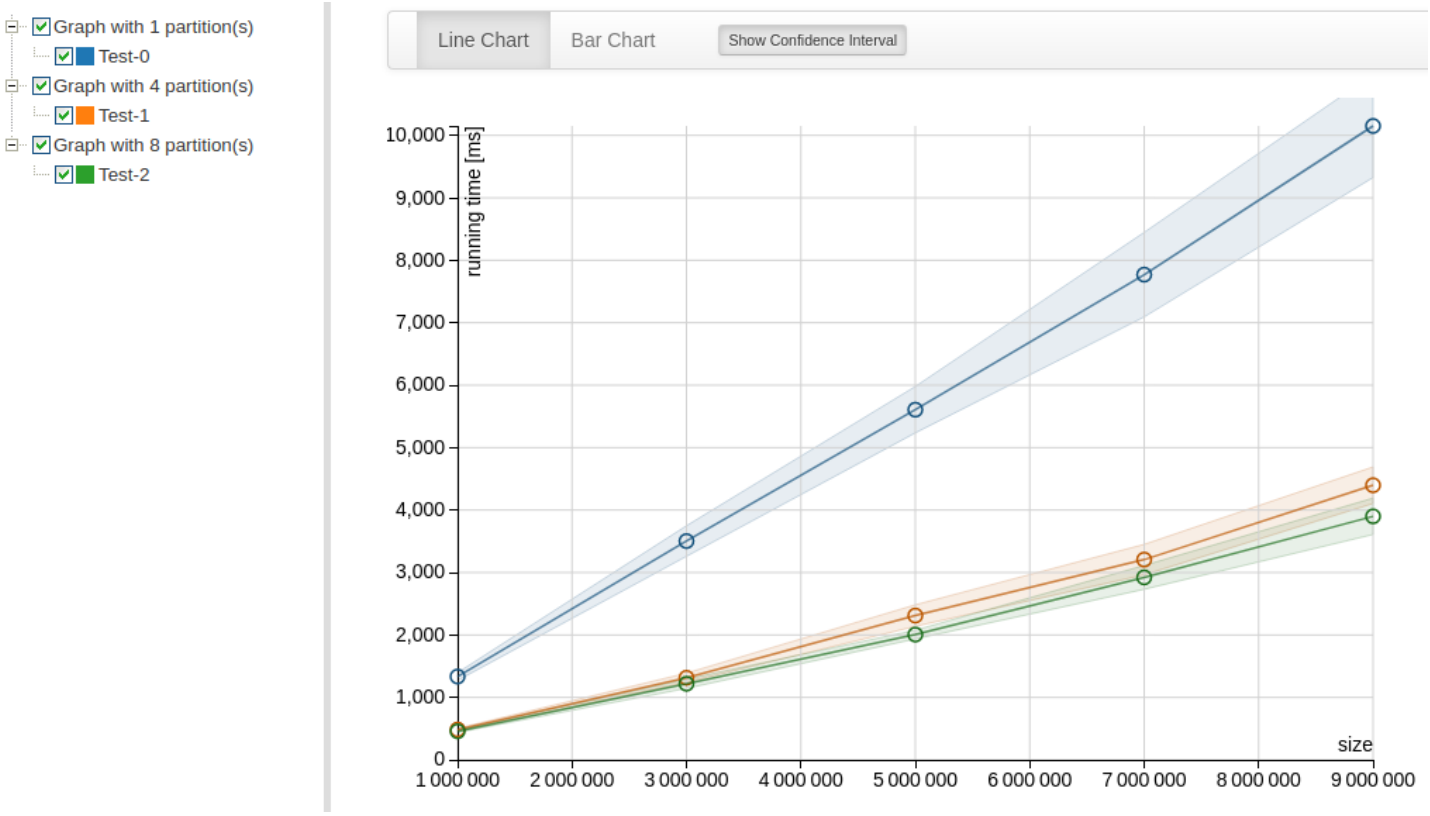


Figura 6.5: Gráfica con funciones de carga con un rango de elementos medio

primero de ellos tendrá una complejidad lineal y el segundo tendrá una complejidad de orden n^2, es decir, cuadrática. Este apartado sienta las bases de uno de los puntos tratados en trabajo futuro, que es realizar una inferencia del coste de programas Flink a partir de pruebas.

Antes de entrar en detalle, es importante remarcar que el código que se utiliza para medir los tiempos, tanto en la función lineal como en la cuadrática, no contienen ninguna funcionalidad ni propósito relevante aparte del coste computacional que generan.

6.3.1. Función lineal

Para esta función se han tomado medidas a un programa de Flink que aplica una función `filter` y luego otra función `map`. En la figura 6.6 se puede ver el código de la función lineal (lineas 1-7) las cuales aplican un filtro al campo `launchYear` de los objeto `Game` del `DataSet` y se devuelve el nombre de todos los objetos `Game` que cumplen el filtro. Los generadores se han creado con los siguientes parámetros:

- `initElements` = 1.000
- `incrementElements` = 5.000
- `iterations` = 10
- `maxPartitions` = 8

```
1  def linearFunct(gameDataset: DataSet[Game], filterYear: Int): Long = {
2    val result = gameDataset
3      .filter(game => game.launchYear >= filterYear)
4      .map(game => game.name).count()
5
6    result
7  }
8
9  def quadraticFunct(gameDataset: DataSet[Game], partitions: Int): Long = {
10     val result = gameDataset
11        .cross(gameDataset)
12        .filter(xs => xs._1.name == xs._2.name)
13        .map(xs => xs._1)
14        .count()
15     result
16   }
```

Figura 6.6: Código para crear la función lineal y la función cuadrática

- `rangePartitions` = 2

Como resultado ha dado la gráfica lineal de la figura 6.7. En la figura se puede observar que, al igual que en las dos últimas pruebas realizadas en la sección 6.2, la eficiencia en la ejecución del programa al crear el generador con una partición es muy inferior a cuando se usan más particiones (sobre todo con cuatro y ocho particiones). Teniendo en cuenta que la distribución del programa solo se ha aplicado en la creación del generador, esto significa que el tiempo en construir el generador es mayor que el tiempo necesario para ejecutar las funciones `filter` y `map`. Aparte de la comparativa entre distinto número de particiones, se puede observar que el crecimiento entre muestra y muestra es claramente lineal en todos los casos. La máquina que se ha usado es la misma que la descrita en la sección 6.2.

6.3.2. Función cuadrática

Para implementar una función cuadrática, se ha usado la función `cross` de Apache Flink sobre el `DataSet` generado consigo mismo, es decir, se ha aplicado un producto cartesiano (línea 11 de la figura 6.6). Una vez aplicado el producto cartesiano también se filtra el resultado y se cuenta el número de elementos. Debido a que la complejidad computacional es mucho más alta, se han creado `DataSet` más pequeños, pasando los siguientes parámetros:

- `initElements` = 1.000
- `incrementElements` = 1.000
- `iterations` = 10
- `maxPartitions` = 8
- `rangePartitions` = 4

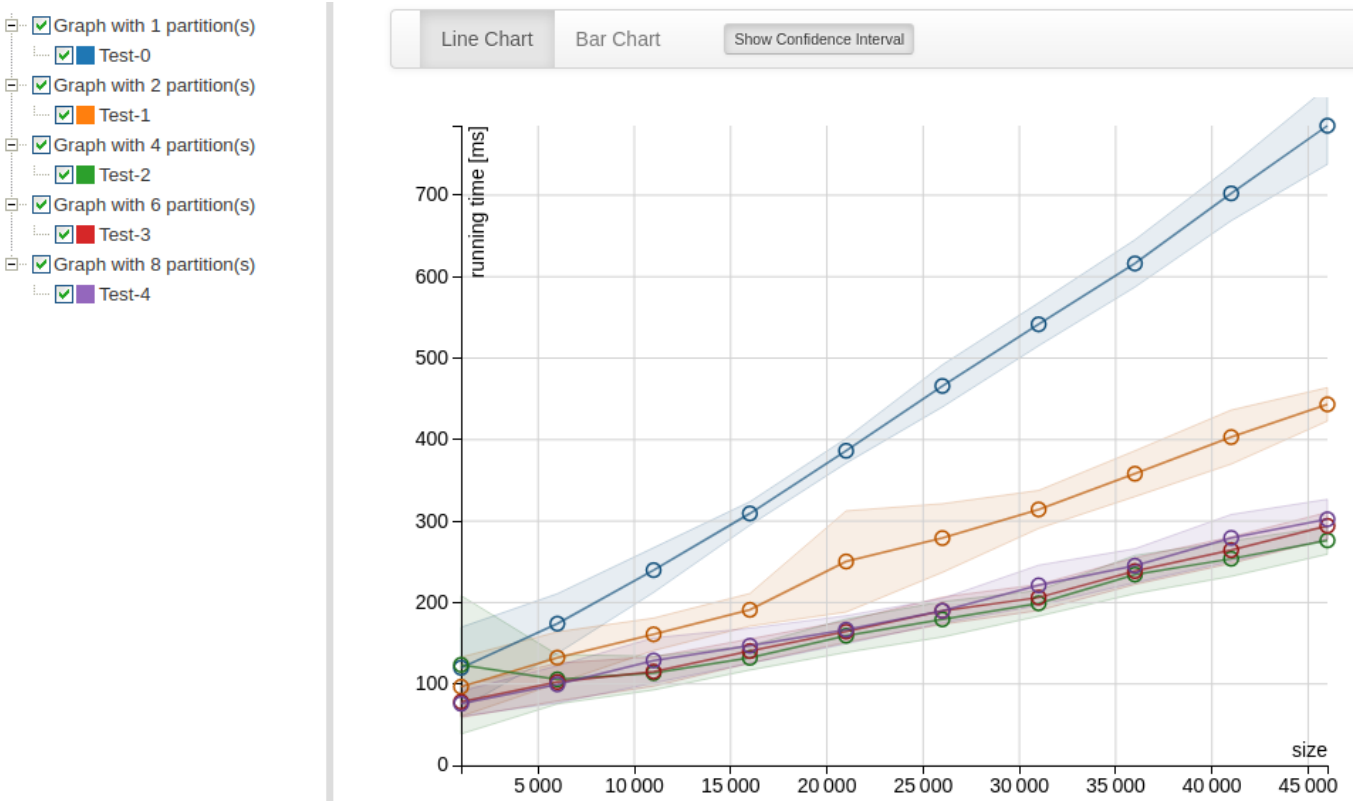


Figura 6.7: Gráfica con funciones de carga de complejidad lineal

La función resultante es la mostrada en la figura 6.8. El comportamiento cuadrático se ve reflejado en la curva que forma la función de manera evidente. Uno de los aspectos más interesantes de la gráfica es que en este caso la diferencia de tiempo entre las ejecuciones con diferentes particiones es casi inapreciable. Esto se debe a que, al contrario que en la función lineal, el cálculo necesario para crear el generador es notablemente más rápido que el necesario para ejecutar la función `cross`, significando esto que la mejora realizada para calcular el generador es muy pequeña en comparación con el tiempo total de ejecución.

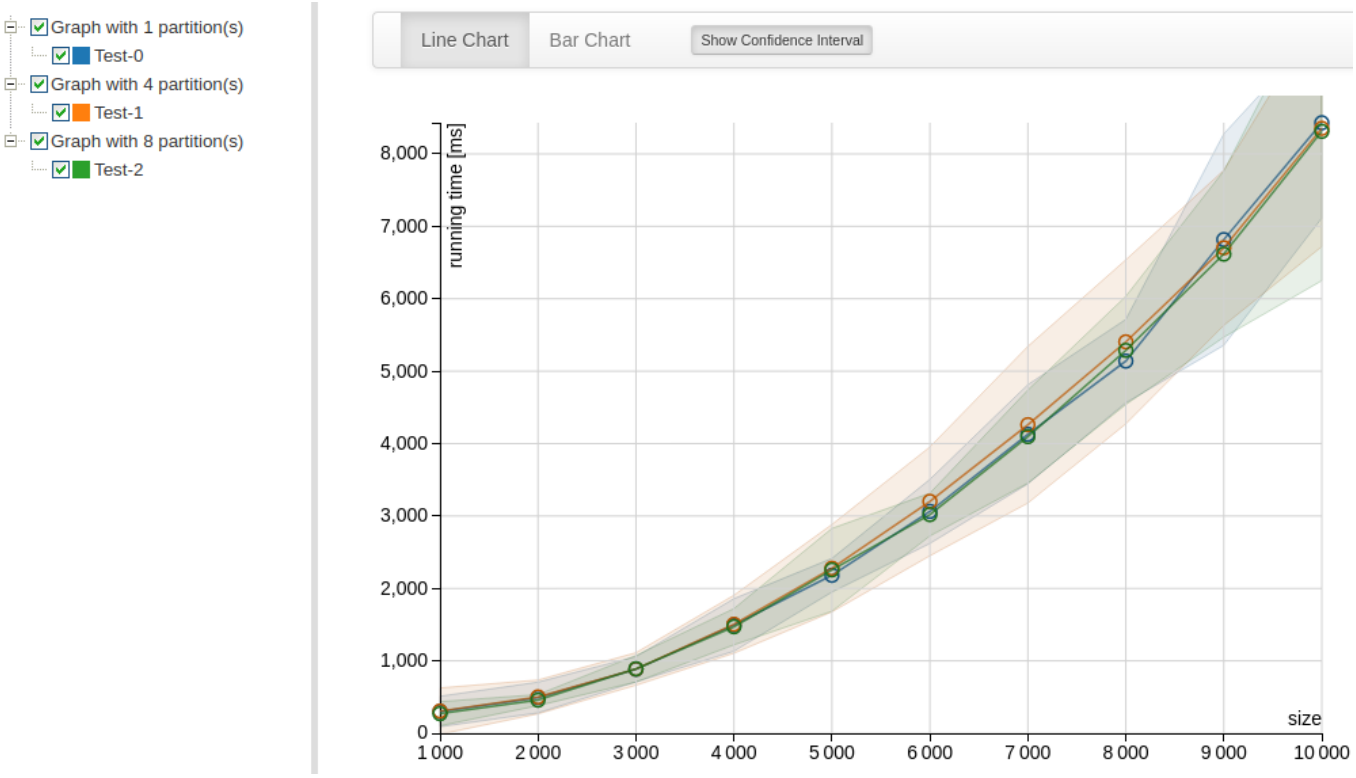


Figura 6.8: Gráfica con funciones de carga de complejidad cuadrática

Capítulo 7

Conclusiones

Llegado el final de este proyecto se puede afirmar que todos los objetivos principales y los objetivos que fueron surgiendo a lo largo del desarrollo (ver en sección 1.2) se han cumplido exitosamente. Respecto a la funcionalidad de la herramienta se pueden realizar las siguientes afirmaciones:

- El generador de `DataSet` y `Table` se ha puesto a prueba en diferentes situaciones y programas típicos de para el procesamiento de grandes cantidades de datos, probando así su versatilidad y flexibilidad en cualquier tipo de propiedad ScalaCheck para testear una gran cantidad de programas.
- Aunque no se ha contado con un *cluster* de Flink compuesto por diferentes máquinas, se ha podido probar que la creación de generadores se puede realizar de manera distribuida, repartiendo la carga de trabajo de manera equitativa entre los recursos que se dispongan.
- La herramienta es tolerante a fallos siempre y cuando el programa de Flink no falle en todos sus reintentos. En caso de que el programa falle en todos sus reintentos o exista un fallo ajeno al entorno de Flink, la herramienta fallará en la creación del generador.
- Se ha optimizado la comparación entre `DataSet` para que se aprovechen los recursos del entorno de Flink y que dicha comparación se pueda realizar de manera distribuida.

Es verdad que se han podido realizar todos estos logros de manera exitosa, pero el camino no ha sido sencillo y, además, todavía existe trabajo por delante. Ambos puntos se tratarán en las siguientes secciones.

7.1. Dificultades encontradas

Se podría decir que la dificultad de este proyecto ha sido más alta en la parte inicial que en el final del mismo, debido a que los conocimientos de las tecnologías usadas al principio del desarrollo eran prácticamente nulas. Las principales dificultades han sido las siguientes:

- Aprendizaje desde 0 del lenguaje Scala, el motor Apache Flink y el concepto de *Property-based testing*.
- Implementación de semillas para añadir la tolerancia a fallos en varias particiones. Este apartado se modificó en varias ocasiones a lo largo de todo el desarrollo del proyecto.
- Implementación de propiedades de ScalaCheck con specs2 ya que no existe demasiada documentación de ambas tecnologías.
- Aprendizaje de la herramienta ScalaMeter, debido a que la documentación de la biblioteca está desactualizada en su página web.

7.2. Trabajo futuro

Por último, la funcionalidad de la herramienta y de las diferentes pruebas realizadas se pueden mejorar. Algunas de las mejoras propuestas son las siguientes:

1. Implementar *shrinking* en la herramienta. Un problema frecuente que se puede dar en ScalaCheck es que una propiedad puede convertirse en una porción de código muy compleja. Esto es debido a que la funcionalidad que se quiere medir sea difícil de comprender, que se usen tipos no primitivos o que los valores que se generen para comprobar la propiedad sean muy grandes. Cuando se intenta validar una propiedad y dicha comprobación falla, se produce lo que se conoce como *shrinking* (a no ser que se haya desactivado). Este proceso consiste en simplificar los valores con los que ha fallado la propiedad, consiguiendo devolver otra combinación de parámetros de entrada más simple, con la que también fallaría la propiedad.

   Cuando se realiza *shrinking* por defecto usando `DataSet` o `Table`, se devuelve el identificador del objeto, lo que no ayuda a poder encontrar cual es el fallo de la propiedad. Por esta razón, puede ser muy útil implementar *shrinking* para ambos tipos.

2. Realizar las pruebas que se han hecho en este proyecto usando un *cluster* con varios servidores, para poder sacar conclusiones más reales sobre cómo funciona la herramienta de manera distribuida.

3. Realizar una inferencia de coste sobre las funciones de carga explicadas en la sección 6.3 mediante un cálculo empírico de la eficiencia, como se expone en el libro *A Guide to Experimental Algorithms* [40].

4. Mejorar el código de las funciones de carga para que actúe como un *framework* en el que el usuario pueda especificar los parámetros que quiera usar para crear las funciones. Con el desarrollo actual es necesario cambiar manualmente en el código los parámetros con los que se generan las funciones. Además, solo se puede crear una función por ejecución. Lo ideal es que el usuario pudiese llamar a una función (o usando una interfaz gráfica) en la que se pudiesen especificar los parámetros deseados.

5. Buscar una biblioteca alternativa a ScalaMeter que tenga un soporte constante y una comunidad más activa.

Chapter 8

Introduction

This chapter explains the context of the project and motivation. Next, it exposes the main objectives will, listing them by milestones. The last section is about the work plan, which explains the monitoring that has been followed on the project, showing the different objectives achieved.

8.1. Description

Nowadays, the amount of information generated manipulated is immense and its volume is growing exponentially in recent years. This is due to the constant improvement of the hardware and computer systems infrastructures. The result is visible in algorithms that large companies use to know what their customers want (from Netflix recommending their series, to Google placing advertising on commercial websites, based on a conversation of one person with another).

To be able to process a huge volume of data, the tools that have been used to store and manage data a few years ago, such as relational databases, are no longer sufficient. Now it is necessary to use data processing engines, designed to be able to carry out this task as efficiently as possible. It is important to note that these engines are designed to be used in a distributed way, that is, several servers are connected in order to carry out a task as efficiently as possible.

Distributed work implies greater complexity when designing and scheduling a task, leading to greater ease of error on the part of the programmer. Another added problem is the difficulty of finding an error in the program because it runs in a distributed manner. Due to this problem, it is essential to perform an exhaustive testing process that can reproduce an error that appears in a program.

Apache Flink is one of these engines, relatively new compared to its competitors (Apache Spark, Apache Hadoop) but it is already been used by companies known as Amazon, King (creators of Candy Crush) or Alibaba among others listed in [1]. It is capable of processing both real-time data streams and separate data sets in batches. Currently, there exist tools to test the data flows created by Apache Flink, such as the flink-check [2] library. As explained in the 2.5.3 section, flink-check is a library that implements PBT using temporal logic formulas.

The main idea of this project is to create a tool that allows creating large data sets using Apache Flink's batch data processing functionality. The data created will be used to test tasks in this engine by using a type of testing called *Property Based Testing* (PBT from now on) which will be explained in detail in this paper (see section 2.1.2).

The tool must be distributed, scalable and, for greater ease in reproducing failed tests, fault-tolerant.

8.2. Objectives

Following the abstract and the introduction of the project, the next objectives are defined:

1. Create a tool capable of generating data sets in Apache Flink which has to be:
   - **distributed**, offering the possibility of being executed in a cluster, as Apache Flink architecture recommends. This implies that generators have to be serializable due to Flink needs to send information in execution time between the workers involved in the task.
   - **scalable** so that the job is distributed evenly when the number of elements generated increases.
   - **partially fault-tolerant** using retries, so the same generator can be replicated if any of the hosts is down when the generator is being calculated.
2. Use the tool to try several Apache Flink programs using PBT paradigm.
3. Implement Apache Flink `Table` generators.
4. Implement several `Matchers` in order to compare `DataSet` objects in a flexible and efficient way.
5. Based on generators, develop a library to make load testing using Apache Flink programs and show graphs with the results.

8.3. Working plan

In order to accomplish the cited objectives, a working plan based on meetings around every fifteen days with the project mentors has been developed.

In the meetings, tasks from the previous meetings were reviewed and new tasks were assigned regarding the milestones achieved. If reviewed tasks had errors or new ideas about that tasks were suggested, then these tasks were assigned for the new iteration again. Following this process, this project simulates a kanban methodology, as accurately as possible.

The objectives are grouped by milestones, which contains several of the tasks discussed in the meetings. These milestones are:

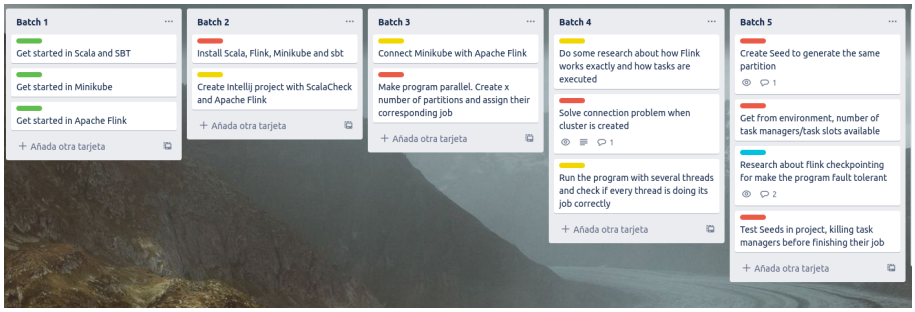


Figure 8.1: How Trello was used. Colours define the priority if every task

1. Research about the main technologies used in this project (Scala, Apache Flink, ScalaCheck).
2. Implementation of a distributed, scalable and fault-tolerant `DataSet` generator tool.
3. Use of PBT generators using ScalaCheck, in order to test its functionality and show its use.
4. Increment of the tool functionality, adding an Apache Flink `Table` generator feature.
5. Fulfillment of several load functions using ScalaMeter framework, so that the tools behavior when it creates generators with different amounts of elements and partitions can be checked.

In order to track the project progress, the following tools were used:

- Email as the main resource the mentors and the student used to communicate with each other.
- Google Meets to make video call meetings when face-to-face meetings were not possible.
- Trello as the tool to track the progress using its board to place every task, simulating kanban methodology as shown in Figure 8.1

Table 8.1 shows in a detailed way of how milestones were accomplished during this project. Each milestone shows the different tasks finished. Dates provided are estimated.

Iterations	**Tasks**
Research about main tools used (1/1/20 to 15/1/20)	▪ Learning how to use Scala and SBT. ▪ Learning how to use Apache Flink.

Creation of a work environment and second part of the research (16/1/20 to 30/1/20)	▪ Installation of Flink, Scala and SBT in Intellij. ▪ Learning how to use ScalaCheck.
Implementation of the generator's basic functionality (1/2/20 to 15/2/20)	▪ Creation of the `DataSet` generator with a given elements generator, a number of elements and a number of partitions the user wants to use. ▪ Simulation of the generators in a cluster in order to check its scalability, using Apache Flink standalone cluster.
Incorporation seeds to the generator so that it can be fault-tolerant (16/2/20 to 29/2/20)	▪ Seed integration so that the generator can always reproduce the same data set of elements, even if Flink fails in the process. ▪ Integration of specs2. ▪ Fulfillment of a test consisting of writing the generated data in temporal files, to check how the seed implementation works. ▪ First code refactorization.

Tests creation using ScalaCheck over several Apache Flink examples (1/3/20 to 15/4/20)	▪ Implementation of Apache Flink WordCount code. ▪ Tests creation using ScalaCheck to execute WordCount code using data created by the `DataSet` generator. ▪ Implementation of Apache Flink TPCH10 code. ▪ Tests creation using ScalaCheck to execute TPCH10 code using data created by the `DataSet` generator. ▪ Test implementation, simulating an ETL tool using data created by the `DataSet` generator. ▪ Implementation of Apache Flink KMeans code. ▪ Tests creation using ScalaCheck to execute KMeans code using data created by the `DataSet` generator.
Tool extension to handle `Table` generators (22/3/20 to 7/4/20)	▪ Second code refactorization. ▪ Implementation of a `Table` generator using the `DataSet` generator. ▪ Test integration to compare `DataSet` objects created using seeds. ▪ Tests extension adding a test for `Table` generator in WordCount.
Adaptation and implementation of several `Matcher` objects from flink-check library (8/4/20 to 21/4/20)	▪ Update of `Matcher` objects from the library to new Scala y Apache Flink versions. ▪ Implementation of a `Matcher` to check the equality between two `DataSet`. ▪ New `Matcher` integration which compares `DataSet` including duplicates (strict mode) or not.

Implementation of load functions using ScalaMeter (22/4/20 to 7/5/20)	▪ Learning how to use ScalaMeter library. ▪ Implementation of a function which returns a ScalaCheck property, used to execute the benchmarking. ▪ Implementation of several tests using generators with a different number of elements and partitions.
Code optimization and writing the paper (8/5/20 to 21/5/20)	▪ Third and last code refactorization. ▪ Integration of code comments in the project, following the Scaladoc standard. ▪ Writing the paper.

Table 8.1: Project work plan

8.4. Paper structure

This paper is divided by several chapters having a specific topic each of them. The chapters are:

- **Preliminaries**: explains the different tools, libraries and methodologies applied in this project.
- **`DataSet` and `Table` generators**: explains the basic functionality of the tool which are functions to create `DataSet` and `Table` generators.
- ***Property-based testing* using generators**: explains how the generators are applied in PBT properties using several Apache Flink programs.
- **Implementation and usage of `Matcher` objects from flink-check**: explains how these objects has been implemented to compare efficiently `DataSet` objects.
- **Elaboration of load functions using ScalaMeter**: shows functions that expose the behavior and efficiency of the generator, not only individually but also in Flink programs, regarding the number of elements generated.
- **Conclusions**: shows the conclusions, the main difficulties found in the project and possible future work to expand the tool.

The whole code is available in the next GitHub repository: `https://github.com/AntonioBarral/TFM-flink-dataset-generators`.

Chapter 9

Conclusions

At the end of this project, it is a fact that every of the main objectives and the ones appeared in the project development, has been accomplished successfully. The next assertions can be done regarding the tool functionality:

- The `DataSet` and `Table` generator has been tested in different situations and using some typical big-data-related programs. Thanks to this, the generators has probed its versatility and flexibility to be implemented in any ScalaCheck property to test a large amount of programs.
- Even without having an Apache Flink cluster consisting of several machines, generators has been tested to run in a distributed manner, being load balanced with the available resources.
- The tool is fault tolerant as long as Flink's program does not fail every of its restarts. In this case or if an external error appears, the tool will fail when it creates the generator.
- `DataSet` comparison has been optimized so that the Flink environment resources will be harnessed. This comparison is also supported in a distributed system.

Milestones has been achieved, however the process has not been easy and there is still work to do to improve the tool. Both topics will be exposed in the next sections.

9.1. Difficulties

The most difficult part of this project was the beginning, due to there was no knowledge about the technologies used in the project. The main difficulties were the following ones:

- Learning from the beginning Scala language, Apache Flink engine and *Property-based testing* concept.

- Implementing seeds in order to add fault tolerance to every partition. This process was modified more than once along the project development.
- Implementing ScalaCheck properties using specs2 due to the lack of documentation for both technologies.
- Learning ScalaMeter tool. In this case, the documentation located in its website is obsolete.

9.2. Future work

To conclude, both the tool functionality and the testing implemented, can be improved. Some of these improvements are:

1. Implement *shrinking* to the tool. A frequent problem in ScalaCheck properties is that the code write on them can become in a really complex piece of code. This can happen because the functionality to measure is very difficult to understand, the use of non primitive types or the values generated are too large. When this kind of property fails when it is being validated, *shrinking* is executed (unless it is disabled). This process consist of simplify the values which produces the property failure, in order to return a simpler combination of values which make the property fail too.

   When *shrinking* is executed by default using `DataSet` or `Table`, what it is returned is an identifier from the object. This does not help to find the error in the property. For this reason it can be helpful to develop a custom *shrinking* for both types.

2. Make the same tests done in this project using a cluster consisting of several hosts, so that the tool can work in a distributed way and the conclusion will be more realistic.

3. Develop a cost inference about the load functions from section 6.3 performing an empiric calculation in the efficiency of the functions, as stated in *A Guide to Experimental Algorithms* book [40].

4. Improve load functions code so that this code can be used as a framework where the user can specify the desired parameters. Currently, it is necessary to change the parameters manually in the code and, what is more, only one load function can be created at a time. Ideally, the user should be able to call a function and pass the parameters by argument (or using a user interface) so that the user can specify the desired parameters.

5. Search for an alternative for ScalaMeter which has a constant support and a more active community.

Bibliografía

[1] Flink Use Cases, 2016. Disponible en `https://wints.github.io/flink-web//usecases.html` (último acceso: mayo 2020).

[2] Cristina Valentina Espinosa, Enrique Martin-Martin, Adrián Riesco, and Juan Rodríguez-Hortalá. FlinkCheck: Property-Based Testing for Apache Flink. *IEEE Access*, 7:150369–150382, 2019.

[3] Dan Radigan. Kanban, 2013. Disponible en `https://www.atlassian.com/agile/kanban` (último acceso: mayo, 2020).

[4] Jiantao Pan. Software Testing. *Dependable Embedded Systems*, 5:2006, 1999.

[5] Rickard Nilsson. *ScalaCheck: The Definitive Guide*. Artima, 2013.

[6] JUnit 5, 2020. Disponible en `https://junit.org/junit5/` (último acceso: mayo, 2020).

[7] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, 46, 01 2000.

[8] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, 5th edition, 2011.

[9] Mina Ayoub. Difference between Distributed and Cluster?, 2016. Disponible en `https://medium.com/@mena.meseha/difference-between-distributed-and-cluster-aca9d50c2c44` (último acceso: mayo, 2020).

[10] Tejas Ambekar. What are the differences between a cluster computer and a distributed system?, 2016. Disponible en `https://www.quora.com/What-are-the-differences-between-a-cluster-computer-and-a-distributed-system` (último acceso: mayo, 2020).

[11] Apache Flink Architecture, 2019. Disponible en `https://flink.apache.org/flink-architecture.html` (último acceso: mayo, 2020).

[12] Fabian Hueske and Vasiliki Kalavri. *Stream Processing with Apache Flink*. O'Reilly, 2019.

[13] Diego García-Gil, Sergio Ramírez-Gallego, Salvador García, and Francisco Herrera. A comparison on scalability for batch big data processing on Apache Spark and Apache Flink. *Big Data Analytics*, 03 2017.

[14] Ivan Mushketyk. Apache Flink vs Apache Spark, 2017. Disponible en `https://dzone.com/articles/apache-flink-vs-apache-spark-brewing-codes` (último acceso: mayo, 2020).

[15] The Scala Programming Language, 2020. Disponible en `https://www.scala-lang.org/` (último acceso: mayo, 2020).

[16] Cay S. Horstmann. *Scala for the Impatient*. Addison-Wesley Professional, 2012.

[17] Josh Suereth and Matthew Farwell. *SBT in Action: The Simple Scala Build Tool*. Manning Publications Co., 2015.

[18] Sbt reference manual, 2020. Disponible en `https://www.scala-sbt.org/1.x/docs/index.html` (último acceso: mayo, 2020).

[19] Rickard Nilsson. ScalaCheck: Property-based testing for Scala, 2018. Disponible en `https://www.scalacheck.org/` (último acceso: mayo, 2020).

[20] specs2: Software specifications for Scala, 2020. Disponible en `https://etorreborre.github.io/specs2/` (último acceso: mayo, 2020).

[21] ScalaMeter, 2019. Disponible en `https://scalameter.github.io/` (último acceso: mayo, 2020).

[22] Apache Hadoop, 2020. Disponible en `https://hadoop.apache.org/` (último acceso: mayo, 2020).

[23] Apache MRUnit, 2016. Disponible en `https://mrunit.apache.org/index.html` (último acceso: mayo, 2020).

[24] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[25] Holden Karau. spark-testing-base, 2020. Disponible en `https://github.com/holdenk/spark-testing-base` (último acceso: mayo, 2020).

[26] Adrián Riesco and Juan Rodríguez Hortalá. sscheck, 2018. Disponible en `https://github.com/juanrh/sscheck` (último acceso: mayo, 2020).

[27] Adrián Riesco and Juan Rodríguez-Hortalá. Property-Based Testing for Spark Streaming. *Theory Pract. Log. Program.*, 19(4):574–602, 2019.

[28] Kartik Khare. A Guide for Unit Testing in Apache Flink, 2020. Disponible en `https://flink.apache.org/news/2020/02/07/a-guide-for-unit-testing-in-apache-flink.html` (último acceso: mayo, 2020).

[29] Data Types & Serialization, 2019. Disponible en https://ci.apache.org/projects/flink/flink-docs-stable/dev/types_serialization.html (último acceso: mayo, 2020).

[30] Parallel Execution, 2019. Disponible en https://ci.apache.org/projects/flink/flink-docs-stable/dev/parallel.html#parallel-execution (último acceso: mayo, 2020).

[31] Dataset Transformations, 2019. Disponible en https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/batch/dataset_transformations.html#hash-partition (último acceso: mayo, 2020).

[32] Table API & SQL, 2019. Disponible en https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/table/ (último acceso: junio, 2020).

[33] Zipping Elements in a Dataset, 2019. Disponible en https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/batch/zip_elements_guide.html (último acceso: mayo, 2020).

[34] Flink examples GitHub, 2020. Disponible en https://github.com/apache/flink/tree/master/flink-examples (último acceso: mayo, 2020).

[35] TPC-H, 2020. Disponible en http://www.tpc.org/tpch/ (último acceso: mayo, 2020).

[36] Douglas Steinley and Michael J. Brusco. Initializing K-means Batch Clustering: A Critical Evaluation of Several Techniques. *Journal of Classification*, 24, 2007.

[37] Pulkit Sharma. The Most Comprehensive Guide to K-Means Clustering You'll Ever Need, 2019. Disponible en https://www.analyticsvidhya.com/blog/2019/08/comprehensive-guide-k-means-clustering/ (último acceso: mayo, 2020).

[38] Krista Miller. What is ETL (Extract, Transform, Load)?, 2019. Disponible en https://www.talend.com/resources/what-is-etl/ (último acceso: junio, 2020).

[39] Repositorio GitHub de ScalaMeter, 2020. Disponible en https://github.com/scalameter/scalameter (último acceso: junio, 2020).

[40] Catherine C. McGeoch. *A Guide to Experimental Algorithmics*. Cambridge University Press, 2012.

Apéndice A

Manual de usuario

En este apéndice se explicará como usar algunas de las funcionalidades desarrolladas en este proyecto. Lo primero de todo es descargar el proyecto del repositorio `https://github.com/AntonioBarral/TFM-flink-dataset-generators`.

Para preparar el entorno y poder así ejecutar los test realizados y/o usar la función de generación de `DataSet` y `Table` es necesario seguir la siguiente instalación.

A.1. Instalación y uso de Intellij + SBT

Es la manera recomendada para usar la herramienta, ya que Intellij proporciona muchas facilidades para añadir bibliotecas con SBT y para desarrollar nuevo código si se desea.

Una vez descargado el código del repositorio, los pasos a seguir son los siguientes:

1. Abrir Intellij y seleccionar *Create new project*.
2. Seleccionar Scala → SBT.
3. En la siguiente ventana seleccionar la ruta donde se ha clonado el proyecto y añadir las versiones del JDK, SBT y Scala de la figura A.1 (son las versiones con las que se ha ejecutado el código del proyecto) y clicar en *Finish*.
4. Si aparece alguna ventana en la que pregunta si sobreescribir algún fichero del proyecto relacionado con Intellij (`.idea` o `.iml`) se clica en la opción de sobreescribir.
5. En este punto ya aparecerá el proyecto con una estructura similar a la de la figura A.2.
6. En caso de querer ejecutar los test, abrir una terminal SBT y ejecutar uno de los test contenidos en la carpeta test → scala (ejemplo en la figura A.3).
7. Si se desea crear código para poder probar el generador de `DataSet` y `Table` es necesario añadir la línea `import generator.Generator` en el fichero donde se quiera usar.

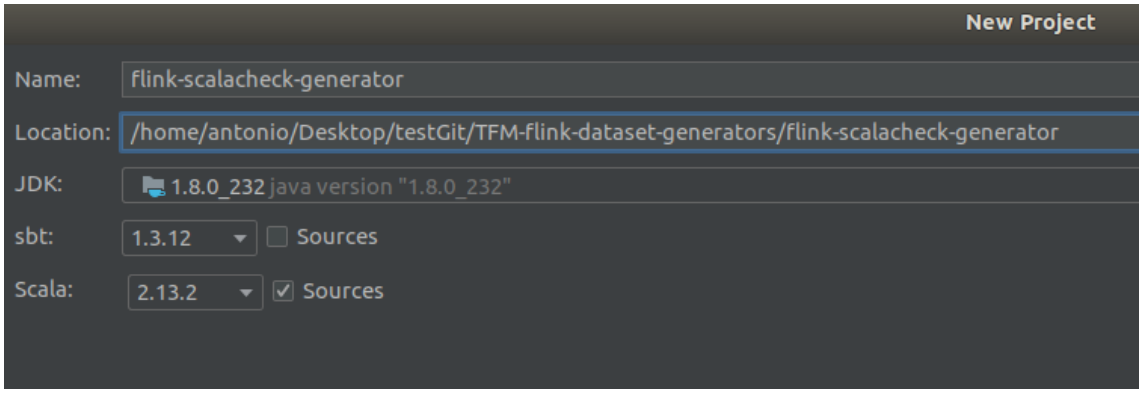


Figura A.1: Versiones de Scala JDK y SBT usadas

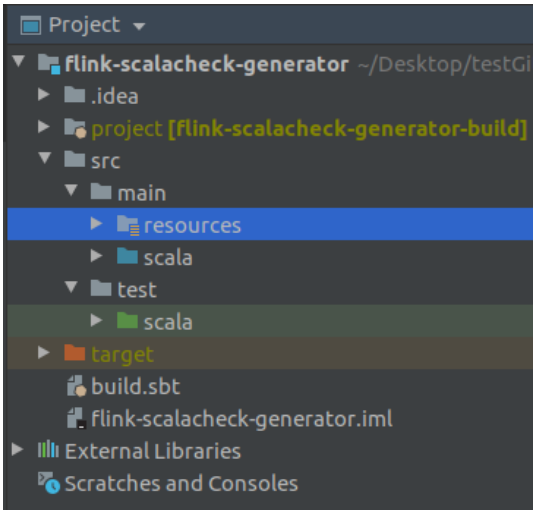


Figura A.2: Estructura del proyecto en Intellij

```
[IJ]sbt:flink-scalacheck-generator> testOnly WordCountTestSpecs
[info] Compiling 7 Scala sources to /home/antonio/Desktop/testGit/TFM-flink-dataset-generators/flink-scalacheck-generator/target/scala-2.12/classes ...
[info] Done compiling.
[info] Compiling 7 Scala sources to /home/antonio/Desktop/testGit/TFM-flink-dataset-generators/flink-scalacheck-generator/target/scala-2.12/test-classes ...
[warn] there was one deprecation warning (since 2.11.0); re-run with -deprecation for details
[warn] one warning found
[info] Done compiling.
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
[info] WordCountTestSpecs
[info] + Count total elements joining 3 datasets and check it is the same compared to the flink wordcount program
[info] + Total count - elements must be different than total count
[info] + Number of words is equal to its length
[info] + Dataset generated is never empty
[info] + The Table val with a greater number of elements will be the one returned by Word Count program
[info] Total for specification WordCountTestSpecs
[info] Finished in 50 seconds, 432 ms
[info] 5 examples, 300 expectations, 0 failure, 0 error
[info] Passed: Total 5, Failed 0, Errors 0, Passed 5
[success] Total time: 58 s, completed Jun 16, 2020 7:28:19 PM
```

Figura A.3: Ejemplo de ejecución de `WordCountTestSpecs`

8. Si se desea ejecutar una función de carga de ScalaMeter, es necesario modificar el `object GeneratorBenchmark` en el fichero con el mismo nombre, dejando descomentada la función que se quiera probar.