Generación y Ejecución de Escenarios de Prueba para Aplicaciones MapReduce

Jesús Morán, Claudio de la Riva, Javier Tuya

Departamento de Informática, Universidad de Oviedo, Gijón, España
moranjesus@lsi.uniovi.es, {claudio, tuya}@uniovi.es

Resumen. Los programas que procesan grandes cantidades de datos se suelen ejecutar sobre una infraestructura distribuida, como es el caso de las aplicaciones implementadas bajo el modelo de procesamiento MapReduce. Estos modelos permiten al desarrollador centrarse en la funcionalidad de la aplicación y abstraer aspectos relacionados con la infraestructura en la que se ejecutará. Sin embargo, su configuración y estado pueden causar que ciertos defectos sean difíciles de detectar debido a que las pruebas se suelen ejecutar en un entorno controlado, con bajo nivel de paralelismo y con pocos datos de entrada. En este artículo se elabora una técnica de prueba que partiendo de unos datos de entrada, genera y reproduce diferentes configuraciones de la infraestructura con el objetivo de detectar aquellas que revelen defectos funcionales en la aplicación. Esta técnica se automatiza en un motor de ejecución de pruebas y se aplica a un caso de estudio que actualmente se encuentra en producción. Como resultado, partiendo de un caso de prueba de tamaño reducido, se han identificado automáticamente varias configuraciones de la infraestructura que ocasionarían fallos de la aplicación.

Palabras clave: Pruebas del software, MapReduce, Big Data Engineering

1 Introducción

Ante las nuevas necesidades de procesamiento masivo de datos en paralelo han surgido un conjunto de tecnologías de datos y modelos de procesamiento que conforman lo que se denomina *Big Data Engineering* [1]. Entre estos destaca *MapReduce* [2] que permite analizar grandes cantidades de datos basándose en el principio de "divide y vencerás". Estos programas se ejecutan en dos fases sobre una infraestructura distribuida: la fase Mapper divide el problema en varios subproblemas, y a continuación la fase Reducer resuelve cada uno de ellos. Es habitual que estos programas se ejecuten en varios computadores con diferentes recursos y características. Para facilitar la gestión de esta compleja infraestructura se emplean frameworks, destacando *Hadoop* [3] por su implantación en las organizaciones [4].

Desde el punto de vista del desarrollador, los programas *MapReduce* se pueden implementar de forma independiente a la infraestructura mediante el desarrollo de las funcionalidades Mapper y Reducer. Para ello, el framework que gestiona la infraestructura se encarga automáticamente de ejecutar el programa sobre varios computado-

res y controlar el procesamiento de los datos desde la entrada hasta la salida. Entre otros, se divide la entrada en varios subconjuntos de datos, se procesan en paralelo y se re-ejecutan partes del programa cuando sea necesario.

A pesar de que el desarrollador puede crear el programa abstrayéndose de la infraestructura, tiene que considerar cómo ésta puede afectarle a la funcionalidad. En un trabajo anterior [5] se han detectado y clasificado varios defectos funcionales que dependen de cómo la configuración de la infraestructura afecta a la ejecución del programa e influye en su resultado. Estos defectos suelen enmascararse durante la ejecución de las pruebas ya que éstas se ejecutan sobre una configuración que no tiene en cuenta las diferentes situaciones que pueden producirse en un entorno de producción, como por ejemplo el nivel de paralelismo o posibles los fallos en la infraestructura [6]. Por otra parte, aunque las pruebas se ejecuten en el entorno similar al de producción, algunos defectos podrían no ser detectables, ya que es habitual que las entradas de prueba tengan pocos datos, lo que conlleva que no sea necesario paralelizar la ejecución. Si bien existen herramientas que contemplan la simulación de algunas de éstas situaciones (por ejemplo, fallos en computadores y red) [7, 8, 9], es difícil diseñar, generar y ejecutar las pruebas de forma determinista ya que hay simular en detalle gran cantidad de los elementos que componen la infraestructura, incluyendo el propio framework.

En este trabajo se presenta una técnica que permite generar automáticamente las diferentes configuraciones de ejecución para una aplicación *MapReduce* en un entorno de desarrollo/pruebas. A partir de los datos de entrada de las pruebas, se obtienen las configuraciones basándose en las diferentes ejecuciones que pueden ocurrir en producción. Cada una de estas configuraciones se ejecuta posteriormente en un entorno de pruebas con el objetivo de reproducir los defectos funcionales del programa que podrían ocurrir en producción. Las contribuciones de este trabajo son:

1. Técnica combinatoria que, a partir de unos datos de prueba, genera las diferentes configuraciones de infraestructura que podrían producirse en un entorno de producción teniendo en cuenta las características de procesamiento de *MapReduce*.
2. Soporte automático para lo anterior con una extensión de MRUnit [10], de forma que permite generar las configuraciones de infraestructura, su ejecución y evaluar si se ha producido un fallo.

El resto del artículo es como sigue. En la sección 2 se resumen los fundamentos del paradigma *MapReduce*. La generación de las diferentes configuraciones, así como la ejecución y automatización de las pruebas se define en la sección 3. En la sección 4 se aplica a un caso de estudio. En la sección 5 se discute el trabajo relacionado sobre las pruebas en los programas *MapReduce*, y finalmente las conclusiones y el trabajo futuro en la sección 6.

2 Paradigma MapReduce

Los programas *MapReduce* procesan grandes cantidades de datos en infraestructuras distribuidas. Para ello, el desarrollador crea dos funcionalidades: Mapper que divide

el problema en varios subproblemas, y Reducer que resuelve estos subproblemas. El resultado final se obtiene a partir del despliegue y ejecución controlada de varias instancias Mapper y Reducer denominadas tarea. Esta labor se realiza automáticamente por *Hadoop* u otro framework. En primer lugar se ejecutan en paralelo varias tareas Mapper que analizan un subconjunto de los datos de entrada y determinan qué subproblemas necesitan esa información. Cuando todas las Mapper finalizan, se ejecutan también en paralelo varias tareas Reducer para resolver los subproblemas. Internamente *MapReduce* maneja pares <clave, valor>, donde la clave es el identificador del subproblema y el valor contiene la información que se necesita para resolverlo.

Suponer a modo de ejemplo que se tiene una gran cantidad de datos históricos sobre temperaturas y que se realiza un programa *MapReduce* para calcular la temperatura media por año. Este programa resuelve un subproblema por cada año, por lo que el identificador del subproblema o clave es el año. La tarea Mapper recibe un subconjunto de datos de las temperaturas y emite pares <año, temperatura de ese año>. A continuación, *Hadoop* agrupa todos los valores por su clave. Por tanto, a la tarea Reducer le llegan subproblemas del tipo <año, [todas las temperaturas de ese año]>, es decir, por cada año se tienen agrupadas todas las temperaturas y finalmente calcula la media. Por ejemplo, en la figura 1 se detalla una ejecución del programa considerando como entrada: año 2000 con 3º, 2002 con 4º, 2000 con 1º, y 2001 con 5º. Las dos primeras entradas se analizan en una tarea Mapper y el resto en otra. A continuación, por cada año se agrupan todas sus temperaturas y se envían a una tarea Reducer. La primera Reducer recibe todas las temperaturas de los años 2000 y 2002, y la otra las del año 2001. Finalmente cada una emite la temperatura media de los subproblemas que analiza: 2º en el 2000, 4º en el 2002, y 5º en el 2001. Este programa con la misma entrada podría ser ejecutado por el framework de distinta forma, por ejemplo con tres Mapper y tres Reducer. Independientemente de cómo se ejecute debería generar la salida esperada.

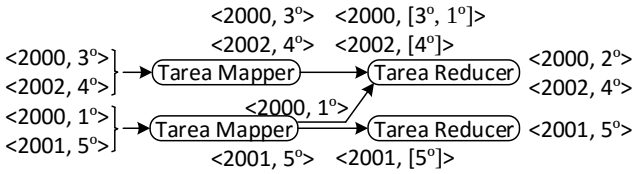

Fig. 1. Ejecución de programa que calcula la temperatura media por año

Adicionalmente, para optimizar el programa se puede implementar la funcionalidad Combiner que se ejecuta después de la tarea Mapper y elimina aquellos pares <clave, valor> que son irrelevantes para resolver el subproblema. También existen otras funcionalidades, como por ejemplo Partitioner que decide a qué tarea Reducer se envía cada par <clave, valor>, Sort que ordena los pares <clave, valor> o Group que determina cómo se agrupan los valores por cada clave antes de llegar a Reducer.

La incorrecta implementación de estas funcionalidades podría causar fallos en alguna de las diferentes formas en que *Hadoop* ejecuta el programa. Estos defectos son

difíciles de detectar en las pruebas ya que debido a que se tienen pocos datos se suele ejecutar sólo una Mapper, luego una Combiner y finalmente una Reducer.

3 Generación y Ejecución de Pruebas

En esta sección se define la generación de configuraciones de prueba (sección 3.1), y un marco general sobre el que ejecutarlas (sección 3.2).

3.1 Generación de escenarios de prueba

Con el objetivo de mostrar cómo las configuraciones de la infraestructura pueden afectar al resultado de los programas, se considera el ejemplo de la sección 2 pero al que se le añade erróneamente una tarea Combiner con el objetivo de reducir los datos que se envían por la red y así mejorar el rendimiento. La tarea Combiner recibe varias temperaturas y las sustituye por un único dato que contiene su media. Al añadirle la tarea Combiner se introduce un defecto funcional ya que Reducer necesita calcular la temperatura media de un año y no la puede obtener con las temperaturas medias parciales que le llegan de Combiner. En la figura 2 se muestran tres ejecuciones que puede seguir el programa en producción considerando diferentes configuraciones de la infraestructura ante un misma entrada.

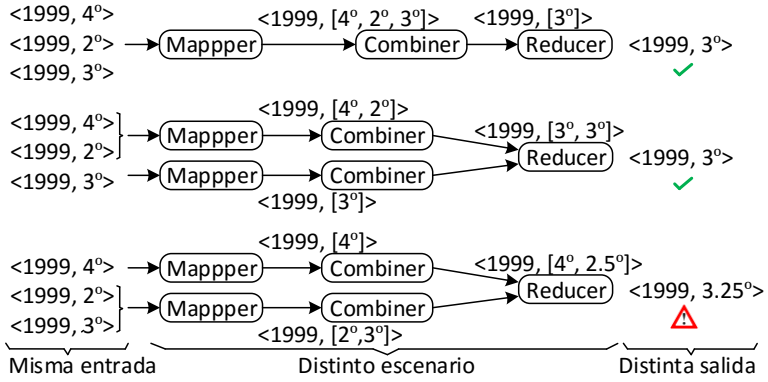

Fig. 2. Ejecución de pruebas para un programa que calcula la temperatura media por año

La primera configuración consiste en una Mapper, una Combiner y una Reducer que producen la salida esperada. La segunda configuración también genera la salida esperada ejecutando una Mapper que procesa las temperaturas 4º y 2º, otra Mapper para 3º, dos Combiner, y finalmente una Reducer. La tercera configuración también ejecuta dos Mapper, dos Combiner y una Reducer, pero produce una salida no esperada debido a que la primera Mapper procesa 4º y la segunda las temperaturas 2º y 3º. Luego una de las tareas Combiner hace la media de 4º, y la otra de las temperaturas 2º y 3º, de forma que a Reducer le llegan 4º y 2.5º. Finalmente la tercera configuración

obtiene que la temperatura media es de 3.25º en lugar de 3º, revelando el defecto funcional. Siempre que se ejecute esta configuración de la infraestructura se detecta el defecto, independientemente que ocurran fallos de computadores, de red u otros. En cambio, se enmascara al ejecutarse otra configuración. El fallo anterior es difícil de detectar ya que se tiene que conocer qué configuración de la infraestructura lo revela y ejecutarla de forma totalmente controlada.

El objetivo es que a partir de unos datos de prueba se generen las diferentes configuraciones de infraestructura, también denominadas *escenarios*. Para ello se tiene en cuenta cómo se podrían ejecutar esos datos de prueba en producción. Los programas *MapReduce* primero ejecutan las Mapper, sobre su salida se ejecutan las Combiner y finalmente las Reducer. Cada ejecución puede realizarse sobre distinto número de computadores y por tanto las tareas Mapper/Combiner/Reducer procesan diferentes conjuntos de datos en cada ejecución. Para generar cada uno de los *escenarios* se propone una técnica que combina [11] los valores de los diferentes parámetros que pueden modificar la ejecución del programa *MapReduce* y producir fallos de acuerdo a la clasificación de defectos MRTree [5]. Estos parámetros son:

- Parámetros Mapper, para los datos de entrada:
  1. Número de tareas Mapper.
  2. Las entradas que procesa cada Mapper.
  3. Orden de procesamiento de los datos de entrada, es decir, qué datos se procesan antes y cuáles después.
- Parámetros Combiner, para la salida de cada tarea Mapper:
  4. Número de tareas Combiner.
  5. Las entradas que procesa cada Combiner.
- Parámetros Reducer, para los datos que le llegan de Mapper y Combiner:
  6. Número de tareas Reducer.
  7. Las entradas que procesa cada Reducer.

Los diferentes *escenarios* se obtienen combinando todos los valores que pueden tomar los parámetros y aplicando las restricciones que impone la ejecución secuencial de tareas *MapReduce*:

1. Los valores/combinaciones de los parámetros de Mapper dependen de los datos de entrada ya que no pueden existir más tareas que datos.
2. Los de Combiner dependen del resultado de las tareas Mapper
3. Los de Reducer dependen del resultado de las tareas Combiner.

Para ilustrar la combinatoria de parámetros y sus restricciones se utiliza como ejemplo el programa de la figura 2. Este tiene una entrada con tres datos, por lo que estos datos restringen los valores que pueden tomar los parámetros de Mapper ya que como máximo podría tener 3 Mapper en paralelo (cada una analizando un dato). El primer *escenario* se genera con una Mapper, una Combiner y una Reducer. Para el segundo *escenario* se modifica el parámetro “Número de tareas Mapper” a 2 donde la primera analiza dos pares <clave, valor> y la segunda uno. El tercer escenario mantiene el parámetro “Número de tareas Mapper” en 2, pero modifica el parámetro “Las entra-

das que procesa cada Mapper" de forma que la primer Mapper analiza un par <clave, valor> y la segunda dos. Modificando de esta forma los valores que pueden tomar los parámetros se van generando los diferentes *escenarios* teniendo en cuenta las restricciones.

3.2 Ejecución de escenarios de prueba

En la sección anterior se ha propuesto una técnica para generar *escenarios* que representan diferentes configuraciones de la infraestructura teniendo en cuenta características del procesamiento *MapReduce*, A continuación, se describe en la figura 3 un marco para ejecutar sistemáticamente las pruebas bajo los diferentes *escenarios* generados con la técnica de la sección anterior.

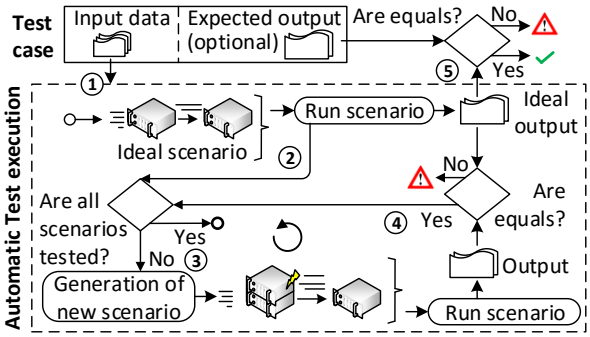

Fig. 3. Marco de ejecución de pruebas

Se parte de un caso de prueba que tiene datos de entrada y opcionalmente la salida esperada (1). Esos datos de entrada se pueden obtener previamente con diferentes técnicas de pruebas genéricas o diseñadas específicamente para *MapReduce* como por ejemplo MRFlow [12]. Se comienza ejecutando los datos de entrada en el *escenario ideal*, que es aquel formado por una configuración con una Mapper, una Combiner y una Reducer, tal y como habitualmente se ejecutan las pruebas (2). A continuación y de forma iterativa se generan y ejecutan nuevos *escenarios* mediante la técnica de la sección anterior (3). Se comprueba que la salida de cada *escenario* sea equivalente a la salida del *escenario ideal*, en caso contrario se ha revelado un defecto a pesar de desconocer cuál es la salida esperada (4). Finalmente, si en el caso de prueba se especifica la salida esperada, se comprueba que también sea equivalente a la del *escenario ideal* (5), sino se detecta un defecto.

Partiendo del caso de prueba, se generan los *escenarios* según la técnica de la sección anterior, y luego se ejecutan y evalúan iterativamente tal y como se describe en el siguiente pseudocódigo:

```
Entrada: caso de prueba formado por:
  (1) datos de entrada
  (2) salida esperada (opcional)
```

```
Salida: escenario que revela el defecto
0  /* Generación de escenarios (sección 3.1) */
1  Escenarios ← Generar escenarios (datos de entrada)
2  /* Ejecución de escenarios */
3  salida escenario ideal ← Ejecución escenario ideal
4  ∀ escenario ∈ Escenarios:
5     salida escenario ← ejecución de escenario
6     SI salida escenario <> salida escenario ideal:
7       EMITIR escenario con fallo
8  SI salida escenario ideal <> salida esperada:
9    EMITIR escenario ideal
10 SINO no se ha revelado ningún defecto
```

Por ejemplo, en la figura 2 se muestra la generación y ejecución de 3 *escenarios* para un caso de prueba. Primero se ejecuta el *escenario ideal* con una Mapper, una Combiner y una Reducer que produce 3º como salida. Luego se ejecuta el segundo *escenario* que también produce 3º. Finalmente se ejecuta un tercer *escenario* que produce como salida 3.25º que no es equivalente a los 3º de la salida del *escenario ideal*. Por tanto se revela un defecto funcional sin conocer la salida esperada del caso de prueba.

El anterior enfoque se automatiza en un motor de ejecución de pruebas a partir de una extensión de la herramienta Apache MRUnit [10]. Ésta sólo ejecuta el *escenario ideal*, por lo que se ha modificado para generar el resto de *escenarios* y ejecutar varias tareas Mapper, Combiner y Reducer.

4 Caso de Estudio

Con el objetivo de evaluar la técnica de prueba, ésta se aplica a Open Ankus [13], una herramienta de minería de datos y aprendizaje automático implementada en *MapReduce*. Consiste en un sistema de recomendación, el cual predice y recomienda varios elementos (libros, películas, etc.) a cada usuario basándose en los gustos almacenados en su perfil. Una funcionalidad del programa se encarga de comprobar el grado de acierto de las recomendaciones, mediante la comparación de la puntuación que el sistema predijo y la que proporcionó el usuario. Esta funcionalidad tiene un diseño MultipleInputs que consiste en dos implementaciones distintas de tareas Mapper: una Mapper recibe archivos con las predicciones del sistema y otra con las puntuaciones del usuario, pero ambas emiten datos a una única implementación de Reducer. Las tareas Mapper reciben las predicciones y puntuaciones de los usuarios sobre los diferentes elementos y las agrupan para cada par usuario-elemento. Las tareas Reducer reciben para cada par usuario-elemento todas las predicciones del sistema y las puntuaciones que fueron proporcionadas por el usuario, por lo que finalmente calcula lo acertadas que fueron tales predicciones.

Sobre el programa anterior, se obtiene un caso de prueba utilizando un análisis de flujo de datos específico para *MapReduce* [12]. El caso de prueba tiene como datos de entrada las siguientes dos predicciones junto con sus puntuaciones: (1) el sistema predice que Laura puntuará El Quijote con 0 puntos, (2) Laura puntúa El Quijote con

0 puntos, (3) posteriormente el sistema detecta que los gustos de Laura han cambiado y predice que puntuará El Quijote con 10 puntos, y (4) Laura puntúa con 10 puntos El Quijote. Estos datos de entrada se proporcionan en dos ficheros, uno de puntuaciones y otro de predicciones. La salida esperada es que el sistema acertó al 100% sus predicciones.

Partiendo del caso de prueba anterior, se ha aplicado el procedimiento descrito en la sección 3. Se han generado y ejecutado 49 *escenarios* de pruebas, detectando un defecto que se produce en 23 *escenarios* (47%). Este defecto sólo se revela cuando unas entradas se procesan antes que otras como ocurre en producción con grandes cantidades de datos, de forma que el programa no es capaz de asignar adecuadamente cada predicción con su puntuación real. Los *escenarios* que revelan el defecto asocian la predicción 10 con la puntuación 0, y la predicción 0 con la puntuación 10, tal y como se representa en la parte inferior de la figura 4. Este *escenario* parte de una Mapper que procesa las predicciones y otras dos Mapper que procesan respectivamente las puntuaciones 0 y 10. Además, este *escenario* fuerza a que se procese primero la puntuación 10 que la puntuación 0, llegando a la tarea Reducer que el par Laura-El Quijote tiene asignadas dos predicciones de 0 y 10 puntos, y que hay dos puntuaciones de 10 y 0 puntos. La tarea Reducer asocia la primera predicción con la primera puntuación y así sucesivamente, por lo que obtiene incorrectamente como salida que el sistema no acierta en las predicciones.

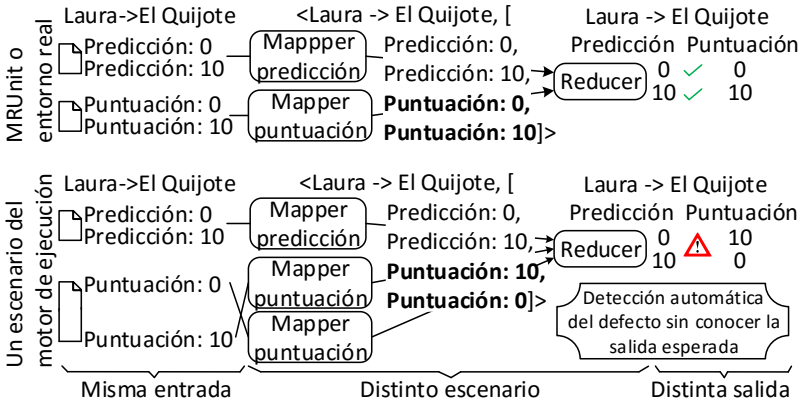

Fig. 4. Ejecución del caso de prueba en distintos escenarios

En cambio, el caso de prueba no revela el defecto si lo ejecutamos en los siguientes entornos: (a) cluster *Hadoop* de producción con 4 computadores, (b) *Hadoop* en modo local (versión más simple de *Hadoop* con un único computador), y (c) librería de pruebas unitarias MRUnit. En la tabla 1 se resumen sus resultados acompañados del tiempo de ejecución del caso de prueba. Estos entornos, a diferencia del motor de ejecución propuesto en este artículo, enmascaran el defecto al ejecutar el caso de prueba en un único *escenario*. Este se corresponde con el *escenario ideal* representa-

do en la parte superior de la figura 4, donde sólo hay dos Mapper, una para las predicciones y otra para las puntuaciones, y una Reducer.

Tabla 1. Resultados del caso de estudio

Evaluación\entorno	Cluster producción	Hadoop local	MRUnit	Motor propuesto
Nº defectos	0	0	0	1
Tiempo ejecución	2 min. 47 s. 67 ms.	4s. 19 ms.	437 ms.	468 ms.

El motor de ejecución de pruebas propuesto en este artículo ejecuta los casos de prueba en los diferentes *escenarios* que pueden ocurrir en producción. Además, a diferencia del resto de entornos analizados, no necesita la salida esperada para detectar defectos funcionales. Por ejemplo, este caso de estudio revela el defecto al comprobar que la salida del *escenario ideal* (el sistema acierta con las predicciones) no es equivalente a la de otro *escenario* (el sistema no acierta con las predicciones).

5 Trabajo Relacionado

A pesar de los desafíos de las pruebas sobre aplicaciones en Big Data [14, 15] y de los avances de técnicas relacionadas con éstas [16], se han realizado pocos esfuerzos orientados a la prueba de aplicaciones *MapReduce* [17] pese a ser uno de los principales paradigmas de procesamiento utilizados Big Data [18]. En un estudio de Kavulya et al. [19] donde se analizan varios programas *MapReduce*, el 3% no terminan de ejecutarse, mientras que otro estudio de Ren et al. [20] lo sitúa entre el 1.38% y el 33.11%.

La mayoría de investigaciones existentes de pruebas para aplicaciones *MapReduce* se centran en el rendimiento y en menor medida en la funcionalidad. En un enfoque de pruebas en programas Big Data propuesto por Gudipati et al. [21] se especifica un proceso específico para la validación en *MapReduce*. Dentro de este, Camargo et al. [22] y Morán et al. [5] identifican y clasifican varios defectos funcionales. Algunos de estos defectos son específicos del paradigma *MapReduce* y no son fáciles de detectar ya que dependen de la ejecución sobre la infraestructura. Un tipo de defecto muy común es el que se produce cuando se espera que los datos lleguen a Reducer en un determinado orden, pero a causa de la ejecución paralela llegan desordenados. Este defecto ha sido analizado por Csallner et at. [23] que proponen detectarlo con un enfoque basado en ejecución simbólica y Chen et al. [24] basándose en model checking. A diferencia de los anteriores, el enfoque presentado en este trabajo no solamente está dirigido a detectar un tipo de defecto, sino que permite detectar otros específicos de *MapReduce*. Para ello, se ejecutan los datos de las pruebas con distintas configuraciones de infraestructura.

Varias investigaciones sugieren realizar pruebas con fallos en la infraestructura [25, 26], existiendo varias herramientas que soportan la inyección de los fallos [7, 8, 9]. Por ejemplo, las investigaciones de Marynowski et al. [27] permiten crear casos de prueba especificando qué computadores fallan y en qué momento. Uno de los posibles

problemas es que pueden existir defectos específicos de *MapReduce* que no se detectan con fallos de computadores, sino que requieren un control total de *Hadoop* y de la infraestructura. En este trabajo se generan automáticamente las distintas formas en las que *Hadoop* podría ejecutar el programa desde el punto de vista funcional, independientemente de que se produzcan por fallos de la infraestructura u optimizaciones de *Hadoop*.

Por otra parte, existen enfoques orientados a obtener las entradas de prueba para aplicaciones *MapReduce*, como [12] donde se utiliza análisis de flujo de datos y [28] basadas en algoritmos bacteriológicos. En este trabajo se generan las configuraciones sobre las que ejecutar unos datos de prueba dados. Estos datos de prueba podrían obtenerse de las anteriores técnicas de prueba.

6 Conclusiones y Trabajo Futuro

En este artículo se ha elaborado una técnica de pruebas y se ha automatizado en un motor de ejecución de pruebas para programas implementados según el modelo *MapReduce*. Éste motor reproduce para un caso de prueba dado las ejecuciones causadas por las diferentes configuraciones de la infraestructura. Automáticamente, y sin necesidad de conocer la salida esperada, puede detectar defectos funcionales específicos del paradigma *MapReduce* que en muchos casos son difíciles de detectar en entornos de prueba y producción. Se ha aplicado a un programa que actualmente está en producción, y para un caso de prueba con pocos datos ha generado y ejecutado automáticamente 49 configuraciones de infraestructura distintas, de las cuales 23 (47%) revelan un defecto.

Como trabajo futuro se planifica extender la técnica para seleccionar eficientemente aquellas configuraciones con más probabilidad de hacer que el programa falle. El enfoque actual es *off-line* ya que las pruebas no se realizan cuando el programa está en producción. Se plantea extender el actual enfoque para realizar pruebas *on-line* monitorizando la funcionalidad con los datos reales de producción y detectando automáticamente los defectos.

Agradecimientos

Este trabajo ha sido realizado bajo el proyecto de investigación TIN2013-46928-C3-1-R, financiado por el Ministerio de Economía y Competitividad, y fondos FEDER. También ha sido realizado bajo el proyecto GRUPIN14-007, financiado por el Principado de Asturias y fondos FEDER.

Referencias

1. ISO/IEC JTC 1 – big data, preliminary report 2014 (2015)

2. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Proc. of the OSDI - Symp. on Operating Systems Design and Implementation, USENIX (2004) 137–149
3. Apache Hadoop: open-source software for reliable, scalable, distributed computing. https://hadoop.apache.org Accessed: 2016-04-16.
4. Instituciones que utilizan Hadoop con fines educativos o de producción. http://-wiki.apache.org/hadoop/PoweredBy Accessed: 2016-04-16.
5. Morán, J., de la Riva, C., Tuya, J.: MRTree: Functional Testing Based on MapReduce's Execution Behaviour. In: Future Internet of Things and Cloud (FiCloud), 2014 International Conference on. (2014) 379–384
6. Vishwanath, K.V., Nagappan, N.: Characterizing cloud computing hardware reliability. In: Proceedings of the 1st ACM symposium on Cloud computing, ACM (2010) 193–204
7. Hadoop Injection Framework. https://hadoop.apache.org Accessed: 2016-04-16.
8. Chaos Monkey. https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey Accessed: 2016-04-16.
9. AnarchyApe: Fault injection tool for Hadoop cluster from Yahoo anarchyape. https://-github.com/david78k/anarchyape Accessed: 2016-04-16.
10. Apache MRUnit: Java library that helps developers unit test Apache Hadoop map reduce jobs. http://mrunit.apache.org Accessed: 2016-04-16.
11. Grindal, M., Offutt, J., Andler, S.F.: Combination testing strategies: a survey. Software Testing, Verification and Reliability **15**(3) (2005) 167–199
12. Morán, J., de la Riva, C., Tuya, J.: Testing Data Transformations in MapReduce Programs. In: Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation. A-TEST 2015, New York, NY, USA, ACM (2015) 20–25
13. Open Ankus: Data mining and machine learning based on MapReduce. http://-www.openankus.org/ Accessed: 2016-04-16.
14. Nachiyappan, S., Justus, S.: Getting ready for bigdata testing: A practitioner's perception. In: Computing, Communications and Networking Technologies (ICCCNT), 2013 Fourth International Conference on, IEEE (2013) 1–5
15. [15] Mittal, A.: Trustworthiness of big data. International Journal of Computer Applications **80**(9) (2013)
16. Bertolino, A.: Software testing research: Achievements, challenges, dreams. In: Future of Software Engineering, 2007. FOSE '07. (2007) 85–103
17. Camargo, L.C., Vergilio, S.R.: Mapreduce program testing: a systematic mapping study. In: Chilean Computer Science Society (SCCC), 32nd International Conference of the Computation. (2013)
18. Sharma, M., Hasteer, N., Tuli, A., Bansal, A.: Investigating the inclinations of research and practices in hadoop: A systematic review Confluence The Next Generation Information Technology Summit (Confluence), 2014 5th International Conference -.
19. Kavulya, S., Tan, J., Gandhi, R., Narasimhan, P.: An analysis of traces from a production mapreduce cluster. In: Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on, IEEE (2010) 94–103
20. Ren, K., Kwon, Y., Balazinska, M., Howe, B.: Hadoop's adolescence: an analysis of hadoop usage in scientific workloads. Proceedings of the VLDB Endowment **6**(10) (2013) 853–864
21. Gudipati, M., Rao, S., Mohan, N.D., Gajja, N.K.: Big data: Testing approach to overcome quality challenges. Big Data: Challenges and Opportunities (2013) 65–72

22. Camargo, L.C., Vergilio, S.R.: Cassicação de defeitos para programas mapreduce: resultados de um estudo empírico. In: SAST - 7th Brazilian Workshop on Systematic and Automated Software Testing. (2013)
23. Csallner, C., Fegaras, L., Li, C.: New ideas track: testing mapreduce-style programs. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ACM (2011) 504–507
24. Chen, Y.F., Hong, C.D., Sinha, N., Wang, B.Y.: Commutativity of reducers. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer (2015) 131–146
25. Faghri, F., Bazarbayev, S., Overholt, M., Farivar, R., Campbell, R.H., Sanders, W.H.: Failure scenario as a service (fsaas) for hadoop clusters. In: Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management, ACM (2012) 5
26. Joshi, P., Gunawi, H.S., Sen, K.: Prefail: A programmable tool for multiple-failure injection. In: ACM SIGPLAN Notices. Volume 46., ACM (2011) 171–188
27. Marynowski, J.E., Santin, A.O., Pimentel, A.R.: Method for testing the fault tolerance of mapreduce frameworks. Computer Networks **86** (2015) 1–13
28. Mattos, A.J.: Test data generation for testing mapreduce systems. In: Master's degree dissertation. (2011)