Información Tecnológica
Vol. 29(2), 133-140 (2018)
http://dx.doi.org/10.4067/S0718-07642018000200133

En la Búsqueda de Soluciones MapReduce Modulares para el Trabajo con BigData: Hadoop Orientado a Aspectos

Cristian L. Vidal-Silva[(1)]*, Miguel A. Bustamante[(2,3)], María del C. Lapo[(3)], María de los Á. Núñez[(4)]
(1) Escuela de Ingeniería en Informática, Facultad de Ingeniería, Ciencia y Tecnología, Universidad Bernardo O'Higgins, Avenida Viel 1497, Ruta 5 Sur, Santiago-Chile (e-mail: cristianvidal@docente.ubo.cl)
(2) Facultad de Economía y Negocios, Universidad de Talca, Avenida Lircay S/N, Talca-Chile (e-mail: mabu@utalca.cl)
(3) Facultad de Ciencias Económicas y Administrativas y del Sistema de Posgrado de la Universidad Católica de Santiago de Guayaquil, Av. Carlos Julio Arosemena Km. 1½ vía Daule, Guayaquil, Ecuador. (e-mail: maria.lapo@cu.ucsg.edu.ec)
(4) Petroecuador, Refinería Esmeraldas, Quito-Ecuador. (e-mail: Angelesnl20@gmail.com)

* Autor a quien debe ser dirigida la correspondencia

Recibido Jul. 27, 2017; Aceptado Oct. 10, 2017; Versión final Nov. 16, 2017, Publicado Abr. 2018

Resumen

En la búsqueda de soluciones MapReduce modulares, la principal meta de este trabajo es aplicar Hadoop y AspectJ para la definición de funciones Aspect-Combine. MapReduce es un enfoque de computación para trabajar con grandes volúmenes de datos (BigData) en un entorno distribuido, con altos niveles de abstracción y con el uso ordenado de funciones Map y Reduce, la primera de ellas para el mapeo o identificación de datos relevantes y la segunda para resumir datos y resultados finales. Hadoop es una aplicación libre de MapReduce que permite la definición de funciones Combine para la agrupación local de datos en la fase de Mapeo y así minimizar el tráfico de información entre Mapper y Reducer. Sin embargo, la ejecución de Combine no es garantizada en Hadoop, asunto que motivó este trabajo. Como resultado, se alcanza un mayor grado de modularización desde un punto de vista teórico, y desde el punto de vista práctico también existen mejoras en rendimiento.

Palabras clave: procesamiento de información; aplicaciones bigdata; arquitectura de software modular; orientación a aspectos

Looking for Modular MapReduce Solutions for Working with BigData: Aspect-Oriented Hadoop

Abstract

Justly, in the search for modular MapReduce solutions, the main goal of this work is to apply Hadoop and AspectJ for the definition of Aspect-Combine functions. MapReduce is a computing approach to work with large volumes of data (BigData) in a distributed environment, with high levels of abstraction and the ordered use of Map and Reduce functions, the first one for mapping or identifying relevant data and the second for resuming data and final results. In a MapReduce system, Mapper and Reducer nodes implement the Map and Reduce functions respectively. Hadoop is a free application of MapReduce that allows the definition of Combine functions. However, the execution of Combine is not guaranteed in Hadoop. This problem motivated this work. As a result, a greater degree of modularization is reached from a theoretical point of view. From the practical point of view there are also improvements in performance.

Keywords: information processing; bigdata applications; modular software architecture; aspects-oriented

INTRODUCCIÓN

Tal y como señala (Guller, 2015), hoy se vive en la era del Big Data, esto es, estar en una sociedad que requiere trabajar con grandes volúmenes de información y, para su procesamiento, realizar alta computación; claramente, dos temas críticos en la actualidad. Justamente, como una solución a estos problemas, aparece MapReduce (Dean y Ghemawat, 2004), un marco de computación distribuida para enfrentar y resolver dichos asuntos. Sin embargo, MapReduce es un enfoque propio de Google (no es open-source) (Dean y Ghemawat, 2010; Dean y Ghemawat, 2004) En este contexto aparece Hadoop (Apache Hadoop, 2016), una herramienta libre y open-source que implementa el enfoque de computación distribuida MapReduce de Google. De acuerdo con (IEEE, 1990), modularidad se refiere al grado en el cual una solución informática se compone de componentes discretos, tal como los cambios a un componente tienen un mínimo impacto en los otros componentes. Por otra parte, tal y como señala (Sullivan et al., 2001), el uso de modularización y ocultamiento de información (Parnas, 1972), tienen un alto impacto en las fases del ciclo de desarrollo de software. Así, estas ideas han contribuido al desarrollo de software desde tipos de datos abstractos, diseño y programación orientada a objetos, y arquitecturas de software. Sin embargo, tal y como menciona (Lesiecki, 2002), algunas soluciones a menudo exhiben comportamiento y datos asociados que no son propios de su naturaleza (comportamiento y datos transversales o incumbencias cruzadas – crosscutting concerns en inglés).

Complementariamente, como indican Laddad, (2009) y Miles, (2004), la seguridad y la sincronización de procesos son ejemplos clásicos de incumbencias cruzadas. La º 1 muestra un pseudo-código de un método con incumbencias cruzadas donde se requiere acceso a un objeto s que representa un semáforo, y explícitamente llamar o invocar los métodos down() y up() de dicho objeto para la sincronización. Claramente, la principal finalidad de este método es el acceso a un recurso compartido y, para garantizar su acceso exclusivo, es necesaria la invocación de los métodos down() y up() de objeto s. De esta manera, tanto el objeto s como la acción de invocar sus métodos son claras incumbencias cruzadas. Así, para el aislamiento y modularización de las incumbencias cruzadas, nace en 1997 la Programación Orientada a Aspectos (POA) (Kiczales et al., 1997).

Desde un punto de vista de Ingeniería de Software, para remarcar los beneficios de la POA, Wampler (2007), indica y presenta ejemplos que demuestran el soporte y afinamiento de POA a principios de Diseño Orientado a Objetos (DOO), tales como el Principio de Responsabilidad Individual Principio (PRI) y el Principio Abierto-Cerrado (PAC). Aun cuando MapReduce y Hadoop permiten altos niveles de abstracción y así no poner atención en asuntos propios de soluciones concurrentes y distribuidas, esta metodología de programación también exige enfrentar un problema con un enfoque de solución definido, mediante el uso de funciones Map para identificar datos y patrones definidos, y Reduce para la agrupación y sumarización de datos. Entonces, estas funciones tienen un rol bien definido, claros ejemplos de una solución modular, y la inclusión de código no propio con su naturaleza es un claro ejemplo de incumbencia cruzada. Justamente, el principal objetivo de este trabajo es proponer y aplicar soluciones MapReduce modulares mediante la simbiosis de soluciones Hadoop en Java y AspectJ, y así dar a conocer un ejemplo con los resultados teóricos y prácticos, además de resaltar ventajas y desventajas de esta propuesta.

```
void jugar(){
   s.down();
  //acceso a recurso compartido
  s.up();
}
```

Fig. 1: Pseudo-código de ejemplo de método con incumbencias cruzadas

MAPREDUCE Y HADOOP

MapReduce es una metodología de programación dada a conocer por Google (Guller, 2015; Dean y Ghemawat, 2004) para la computación distribuida sobre grandes cantidades de datos o Big Data. Desde entonces, MapReduce ha tenido una masiva adopción por medio de Hadoop, una implementación libre de código abierto (Apache Hadoop, 2016; White, 2015; Lin y Dyer, 2013).

La idea básica de MapReduce es dividir un problema complejo en sub-problemas más pequeños que pueden ser tratados de manera independiente, esto es, en paralelo por múltiples hilos trabajadores en un procesador, por múltiples procesadores trabajadores en una máquina multiprocesador, o múltiples máquinas trabajadoras en un clúster o red de máquinas (White, 2015), donde su salida final es el resultado de la combinación de resultados intermedios de cada una de los trabajadores. De manera individual, los trabajadores o workers son

mapeadores y reducidores (mappers o reducers). Desde un punto de vista algorítmico, la técnica *divide y vencerás* simboliza la base de MapReduce, esto es, dividir el Big Data en porciones más pequeñas para su procesamiento en paralelo (White, 2015). Además, MapReduce provee un medio para distribuir la computación sin los costos propios de la programación distribuida tales como el garantizar el envío y recepción de mensajes.

La figura 2 ilustra la arquitectura de funcionamiento de Hadoop. Como se aprecia en esta figura, tal y como describe Galindo et al., (2016), un programa de solución o trabajo en Hadoop ejecuta 4 pasos principales: i) División de datos donde múltiples fracciones de datos son entregadas a cada uno de los mapeadores, ii) Map donde se ejecutan las funciones Map para procesar los datos y procesador con la identificación de elementos relevantes y enviarlos a la etapa de organización, iii) Organización para organizar los datos y agrupar resultados intermedias, y entonces distribuirlos hacia la etapa de reducción, y iv) Reduce donde se ejecutan las funciones Reduce para compactar y resumir los resultados a ser escritos en disco. En la práctica, objetos mapeadores y reducidores ejecutan funciones *map* y *reduce*, respectivamente. Así, objetos mapeadores son responsables de procesar pares clave-valor para producir un conjunto de pares clave-valor intermedios; y luego, objetos reducidores procesar y resumir el conjunto de valores intermedios junto con su clave compartida.

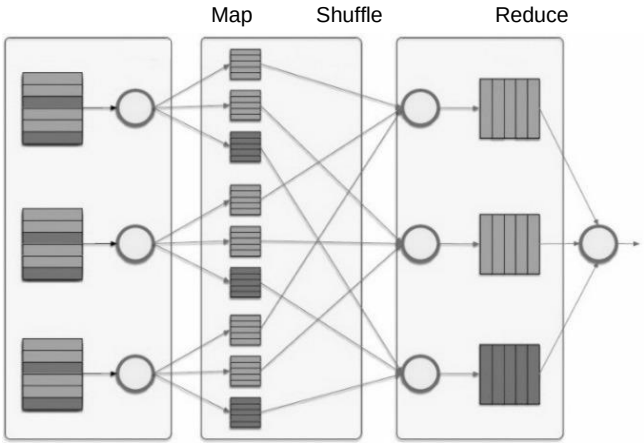

Fig. 2: Arquitectura de Funcionamiento de MapReduce.

PRINCIPIOS DE MODULARIDAD Y PROGRAMACIÓN ORIENTADA A ASPECTOS (POA)

Según Mialnes, (2008), el bajo acoplamiento y la alta cohesión son principios fundamentales y necesarios para lograr un diseño de software modular. Justamente, gracias a la separación de incumbencias, la POA permite un alto nivel de cohesión y bajo acoplamiento entre módulos aconsejados y aspectos. Además Wampler, (2007), indica que la POA soporta principios de un buen diseño orientado a objetos tales como principio de única responsabilidad y el principio de abierto-cerrado. La POA, en la búsqueda de un comportamiento modular de clases, define clases ingenuas de ser aconsejadas y modulariza incumbencias cruzadas como aspectos que representan métodos ortogonales que no pertenecen a las clases aconsejadas (Kiczales et al., 1997); así, soluciones POA son aptas para respetar el principio de única responsabilidad de diseño orientado a objetos DOO (Wampler, 2997). Además, POA permite modularizar incumbencias cruzadas dinámicas avanzadas, y no solo la intercepción de llamadas y ejecución de métodos. Por ejemplo, AspectJ (Miles, 2004; Griswold, et al., 2001; Gradecki y Lesiecki, 2003; Kiczales et al., 1997), como lenguaje de POA, permite el uso de cflow, if, y execution para la definición de puntos de corte. La figura 3 ilustra los componentes y la estructura de funcionamiento de soluciones AspectJ (Bodden et al., 2014).

Tal como Miles (2004), Gradecki y Lesiecki (2003), Griswold et al. (2001) y Kiczales et al. (1997) remarcan, la POA permite modularizar con relativa facilidad las denominadas incumbencias cruzadas homogéneas, pero realizar una modularización de clases mediante el uso de aspectos resulta en el software complejo para su evolución ya que los aspectos asociados no reflejan la estructura y cohesión de las características y módulos refinados. Además, tal y como resaltan Bodden et al. (2014), las soluciones de programación POA estilo AspectJ introducen dependencias implícitas entre aspectos y clases. Además, los aspectos de dichas soluciones, usualmente, no respetan el principio de ocultamiento de la información.

Dado lo anterior, como una posible consecuencia, instancias de clases ingenuas aconsejadas pueden experimentar un comportamiento no esperado así como cambios en sus propiedades, además de aspectos no efectivos o falsos como resultado de posibles cambios por la modificación del comportamiento a ser aconsejado de clases ingenuas. Así, tal y como señala Bodden et al. (2014), en lenguajes de POA similares

a AspectJ, los aspectos requieren conocer claramente acerca de las clases a ser aconsejadas antes de aconsejar dichas clases, lo cual simboliza un gran asunto para el desarrollo de soluciones POA de forma independiente.

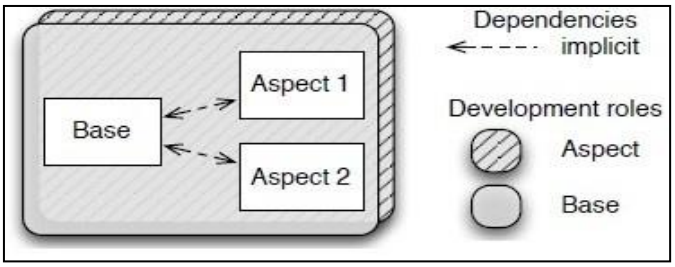

Fig. 3: Asociaciones entre módulo base y aspectos en POA tradicional.

FUNCIONES ASPECT-COMBINE

Las funciones *Aspect-Combine* representan aspectos de la POA sobre funciones de mapeo de Hadoop para así respetar principios de DOO (Wampler, 2007), y así mismo, implementar ideas de funciones combinadoras en-mapeadores (Lin y Dyer, 2013) de manera de mantener y preservar la simplicidad de las funciones de mapeo. Particularmente, la meta de una función Aspect-Combiner es obtener una agregación local en los mapeadores, de manera de disminuir el tráfico de red entre mapeadores y reducidores, y también las operaciones de entrada / salida de disco asociadas. En consecuencia, como se puede observar, la figura 4, muestra una aplicación que corresponde a una función in-Mapper Combining de una solución al problema de contar las palabras de un texto (WordCount). Como se aprecia, además de considerar el principal objetivo de la función Map de analizar los datos recibidos para reconocer elementos a ser transferidos a la función de reducción; este ejemplo combina o totaliza resultados obtenidos según las palabras identificadas. Claramente, esta solución no respeta el principio de única responsabilidad de la función Map.

```
private Text word = new Text();
HashMap<String, Integer> palabras =
        new HashMap<String, Integer>();

public void map(LongWritable key, Text value,
        OutputCollector<Text, IntWritable> output,
        Reporter reporter) throws IOException{

  palabras.clear();

  String line = value.toString();
  StringTokenizer tokenizer = new StringTokenizer(line);
  while (tokenizer.hasMoreTokens()){
     String w = tokenizer.nextToken();

     if (palabras.containsKey(w))
         palabras.put(w, palabras.get(w )+ 1);

     else
         palabras.put(w, 1);
  }
```

```
  for (Entry<String, Integer> entry :
                        palabras.entrySet()){

      String k = entry.getKey();
      Integer v = (Integer) entry.getValue();

      word.set(k);
      output.collect(word, new IntWritable(v));
  }
}
```

Fig. 4: Ejemplo de función In-Mapper Combining Hadoop para un contador de palabras de un texto.

Dados los asuntos de modularidad de In-Mapper Combining ya mencionados, la tabla 1 presenta la hipótesis nula y alternativa de este trabajo, así como detalle de los modelos usados como entrada, los invariantes y las contantes para la realización de los experimentos. En la búsqueda de solución Map modular, claramente la clase mapper requiere al menos de un atributo adicional para mantener el conjunto de valores identificados. Justamente, en el ejemplo de WordCount, además de las palabras identificadas, por cada una de ellas, es necesario conocer el número de ocurrencias o apariciones en el texto ya analizado. Así, en Aspect-Combine, mediante declaración de datos entre-tipos, se implementan dichos atributos adicionales en la clase junto con los métodos para operar sobre dichos campos.

Además, hay tres eventos que deben ser inyectados sobre las funciones de mapeo aconsejadas: uno para inicializar los atributos, uno para considerar una instancia de un elemento existente o crear uno nuevo en caso de la no existencia de este, y uno para enviar información a la siguiente fase de una solución MapReduce en Hadoop. Claramente, las definición de la posible ocurrencia de estos eventos se corresponden con reglas de pointcut o punto de corte en POA y AspectJ (Laddad, 2009; Miles, 2004; Gradecki y Lesiecki, 2003). Respecto al tiempo asociado a la inyección del nuevo comportamiento se corresponde con los consejos o advices de

POA, los cuales son de 3 tipos, tal y como se mencionó antes: before, around y after. Justamente, las figuras 5, 6 y 7 presentan una solución para la definición de puntos de corte, declaración entre tipos y consejos.

Tabla 1: Metodología de la Investigación para la realización de los experimentos.

Hipótesis del Experimento	
Hipótesis Nula (H_0)	Soluciones Aspect-Combine no son más rápidas ni más modulares que Combiner e In-Mapper Combining para el caso de estudio de contador de palabras.
Hipótesis Alternativa (H_1)	Existen casos en los cuales soluciones Aspect-Combine trabajan más rápido que Combiner e In-Mapper Combining sin la presencia de incumbencies cruzadas para el caso de studio de Contador de palabras.
Modelos de entrada	Archivos aleatoriamente generados a partir de la selección de palabras de un conjunto definido. El tamaño de estos archivos varía de 10KB a 1MB.
Invariantes	Para cada experimento, se crea un conjunto de archivos que en conjunto son de tamaño mayor que para el experimento previo.
Constantes	
Hadoop 2.4.1 en Ubuntu Linux 14.02	Soluciones para Contador de palabras implementas durante 2016.

```
pointcut init(WordCountMapper mapper):
    execution(* map(..)) && target(mapper);

pointcut send(WordCountMapper mapper, Text word,
    IntWritable val):
    call(* collect(..)) && args(word, val) &&
    this(mapper)
    && !within(AspectMapper);

pointcut end(WordCountMapper mapper, LongWritable key,
    Text value, OutputCollector<Text, IntWritable> output,
    Reporter reporter): execution(* map(..)) &&
    args(key, value, output, reporter) &&
    target(mapper);
```

Fig. 5: Definición de Puntos de Corte para una Solución Aspect-Combine del Ejemplo de un Contador de Palabras.

```
private ArrayList<Palabra> WordCountMapper.palabras =
        new ArrayList<Palabra>();

public void WordCountMapper.initPalabras(){
    palabras.clear();
}
public void WordCountMapper.incPalabra(String palabra){
    if (palabras.contains(palabra)){
        Integer index = palabras.indexOf(palabra);
        Palabra p = palabras.get(index);
        p.incCuantas();
        palabras.set(index, p);
    }
    else
        palabras.add(new Palabra(palabra));
}

public ArrayList<Palabra> WordCountMapper.getPalabras(){
    return palabras;
}
```

Fig. 6: Declaración Intertipo para Aspect-Combine en el ejemplo de un Contador de Palabras.

RESULTADOS

Claramente, desde un punto de vista de modularidad, Aspect-Combine permite que la clase Map respete principios esenciales de diseño orientado a objetos tales como el principio de única responsabilidad. Desde un punto de vista práctico, la tabla 1 presenta los resultados de tiempo de ejecución de soluciones a WordCount con el uso de Combinadores tradicionales, Combinadores como parte de funciones Map, y la propuesta de solución de Aspect-Combine. Como se aprecia en estos resultados, salvo en el caso de archivo pequeño, las soluciones de Combinador en Funciones Map y Aspect-Combiner presentan un resultado superior a solución tradicional. Además, en ejemplos de ejecución de mayor tamaño Aspect-Combiner permite obtener mejores resultados de computación.

Según los resultados de Tabla 1, en la cual se presenta el tiempo en milisegundos de WordCount tradicional, y de las soluciones In-Mapper Combing y Aspect-Combine, junto con el % relativo de mejora (+) o tiempo extra (-) relativa a la solución tradicional, en los experimentos 3 y 4, Archivo Words + 3 ebooks (168 B + 4.8 MB) y Archivo Words + 30 ebooks (168 B + 48 MB), claramente las soluciones de Aspect-Combine son muy superiores en rendimiento. Ante esta situación, dado que el tejedor de AspectJ es quien finalmente mezcla los códigos de los aspectos y los módulos base, los productos tejidos finales resultan en soluciones de mejor rendimiento. Todo esto, con la mayor modularidad que permite el uso de soluciones orientadas a aspectos.

Respecto al equipo computacional para realizar estos experimentos, se utilizó un Intel Core I3 con 16GB de RAM y 500GB de Disco Duro.

```
before(WordCountMapper mapper): init(mapper){
      mapper.initPalabras();
}


void around(WordCountMapper mapper, Text word,
     IntWritable val): send(mapper, word, val){

    mapper.incPalabra(word.toString());
}

after(WordCountMapper mapper, LongWritable key, Text value,
    OutputCollector<Text, IntWritable> output,
    Reporter reporter) throws IOException:
    end(mapper, key, value, output, reporter){

    ArrayList<Palabra> palabras = mapper.getPalabras();

    for(Palabra p: palabras){
         Text w =  new Text();
         IntWritable c = new IntWritable();

         w.set(p.getPalabra());
         c.set(p.getCuantas());

         output.collect(w, c);
    }
}
```

Fig. 7: Consejos para Aspect-Combine en el Ejemplo de Contador de Palabras.

Por último, tal y como señala Guller (2015), se vive en una era de Big Data lo que motiva el desarrollo y aparición de nuevas tecnologías para el soporte de Big Data. Es así como, se prevé que, en un trabajo futuro, se analice la aplicabilidad teórico y práctica de la POA, en particular de AspectJ, sobre otros tecnologías de Big Data tales como Giraph (Martella et al., 2015) y Spark (Shi et al., 2016), para finalmente probar el uso de otros enfoques de POA tales como JPI (Bodden et al., 2014) y Ptolemy (Rajan et al., 2011) sobre tecnologías Big Data; y así, evaluar su efectividad y rendimiento práctico.

Tabla 2: Resultados solución Hadoop tradicional, in-Mapper Combining, y Aspect-Combine con ejemplo WordCount.

Entrada	*Resultados Versiones WordCount en Milisegundos (ms)*					
	Tradicional - %		In-Mapper Combining		Aspect-Combine	
Archive Words (168 B)	2768,8	0%	2797,6	- 1%	2830,7	- 2%
Archivo Words + 1 ebook (168 B + 1.6 MB)	6474,8	0%	4750,8	+ 26,6%	5675,3	+ 12,3%
Archivo Words + 3 ebooks (168 B + 4.8 MB)	7799,1	0%	7770,8	+ 0,3%	7646,4	+ 2%
Archivo Words + 30 ebooks (168 B + 48 MB)	36835,5	0%	36011,4	+ 2%	33913,1	+ 8%

CONCLUSIONES

Con los resultados de este trabajo y la discusión de ellos, se puede concluir lo siguiente:

(i) Las funciones Aspect-Combine son un claro ejemplo de simbiosis entre soluciones MapReduce y POA, en particular un ejemplo de simbiosis entre Hadoop y AspectJ; *(ii)* Tal y como se describió en este trabajo, dado el principal foco de las funciones Map y Reduce, *MapReduce* permite la definición de funciones Combine las cuales no son siempre efectivas. Dado que la principal misión de Combine es sumarizar o totalizar resultados de funciones Map, antes de ser enviados a la siguiente etapa, para reducir el número de operaciones de envío y recepción de mensajes, para garantizar la efectividad de funciones Combine; *(iii)* En atención a que en este trabajo se detalla y analiza, desde un punto de vista de modularidad el uso de in-mapper Combine, entonces, este trabajo propone funciones Aspect-Combine para garantizar soluciones MapReduce modulares y efectivas, además de eficientes respecto a experimentos realizados.

La gran ventaja de Aspect-Combine es la generalidad de la misma ya esta sería aplicable a todo problema MapReduce que permita una agregación local. Justamente, los autores están actualmente realizando otros experimentos con el uso de Aspect-Combine para validar su utilidad. Además, aun cuando MapReduce funciona en entornos distribuidos usualmente, dado que Aspect-Combine, al igual que In-Mapper Combining, funciona en el nodo Mapper, esta agrupación local tiende a causar mejores rendimientos cuando dicha agregación existe. Es decir, el uso de Aspect-Combine no afectaría el rendimiento de una solución MapReduce en un entorno completamente distribuido.

REFERENCIAS

Apache Hadoop. Welcome to Apache Hadoop (en línea), http://hadoop.apache.org/, Access Date: October 28th (2016)

Bodden E., E. Tanter y M. Inostroza, Join Point Interfaces for Safe and Flexible Decoupling of Aspects, in ACM Trans. Softw. Eng. Methodol. 23(1), Article 7, pp. 1-41 (2014)

Dean J. y S. Ghemawat, MapReduce: a flexible data processing tool, Communications of the ACM, 53(1), pp.72-77 (2010)

Dean J. y S. Ghemawat, MapReduce: Simplified data processing on large clusters, OSDI 2004, San Francisco, CA, USA, pp.137-150 (2004)

Galindo J., M. Acher, J. M. Tirado, C. Vidal C., B. Baudry y D. Benavides, Exploiting the enumeration of All Feature Model Configurations - A New Perspective with Distributed Computing, in proceedings of, SPLC 2016, Beijing, China, September 2016.

Gradecki J. y N. Lesiecki, Mastering AspectJ: Aspect-Oriented Programming in Java, John Wiley & Sons, Inc., New York, NY, USA, (2003)

Griswold B., E. Hilsdale, J. Hugunin, W. Isberg y G. Kiczales, Aspect-Oriented Programming with AspectJ™, AspectJ.org, Xerox PARC, (2001) Tutorial slides (en línea), https://goo.gl/6oV4mW. Acceso: 28 de Octubre de (2016)

Guller, M. Big Data Analytics with Spark., ISBN-13 (pbk): 978-1-4842-0965-3, Apress – Spring, New York, NY, USA, (2015)

IEEE, IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990 (1990)

Kiczales G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier y J. Irwin J., Aspect oriented programming, in Proceeding of the European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag LNCS 124, Finland (1997)

Laddad, R. AspectJ in Action: Enterprise AOP with Spring Applications, Manning Publications Co., 2nd Ed., Greenwich, CT, USA (2009)

Lesiecki, N. Improve modularity with aspect-oriented programming AspectJ brings AOP to the Java language, IBM, Developer Work, 01 January, (2002), (en línea), https://goo.gl/sNYstn. Acceso: 28 de Octubre de (2016)

Lin J. y C. Dyer, Data-Intensive Text Processing with MapReduce, Morgan and Claypool Publishers, USA (2013)

Martella C., D. Logothetis y R. Shaposhnik, Practical Graph Analytics with Apache Giraph, Apress, USA (2015)

Mialnes, A. Language Support for the Heterogeneous Migration of Computations, Tesis de Doctorado en Ciencias de la Computación, Pontificia Universidad Católica de Rio de Janerio, Rio de Janeiro, Brazil, pp. 37–38 (2008)

Miles, R. AspectJ Cookbook, O'Reilly Media, Inc, USA, (2004)

Parnas, D. On the criteria to be used in decomposing system into modules, Communications of the ACM, 15 (12), December, pp. 1053–1058 (1972)

Rajan H., Leavens G. T., Dyer R. y Bagherzadeh M. Modularizing crosscutting concerns with Ptolemy. In Proceedings of the tenth international conference on Aspect-oriented software development companion (AOSD '11), ACM, New York, NY, USA, 155-179 (2011)

Shi J., Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald y F. Öscan, "Clash of the Titan: MapReduce vs. Spark for Large Scale Data Analytics," in Proceedings of the Very Large Data Bases (VLDB), September 5th-09th, New Delhi, India (2016)

Sullivan K., W. G. Griswold, Y. Cai y B. Hallen, The Structure and Value of Modularity in Software Design, in proceedings of the 8th European Software Engineering Conference ESEC/FSE-9, Viena, Austria, pp. 99-108 (2001)

Wampler, D. Aspect-Oriented Design Principles: Lessons from Object-Oriented Design, Sixth International Conference on Aspect-Oriented Software Development (AOSD'07), Vancouver, British Columbia, Canada, March, pp. 12-16 (2007)

White, T. Hadoop: The Definitive Guide, O'Reilly, 4th Ed., Sebastopol, CA, USA (2015)