Plataformas de Big Data: Spark, Storm e Flink

Technical Report · July 2015

DOI: 10.13140/RG.2.1.3147.1209

1 author:

Jean Bez
Universidade Federal do Rio Grande do Sul
35 PUBLICATIONS **42** CITATIONS

Plataformas de *Big Data*: Spark, Storm e Flink

Jean Luca Bez

5 de julho de 2015

1 Introdução

Diferentes plataformas para o tratamento de *Big Data* vêm sendo construídas, com diferentes finalidades. O objetivo deste trabalho é comparar as diferenças e funcionalidades das plataformas: Apache Spark, Apache Storm e Apache Flink. Além da comparação serão executados experimentos entre duas delas para avaliar o tempo de execução e o *speedup* obtido.

O restante do trabalho está divido da seguinte forma: a seção 2 apresenta os principais detalhes das plataformas que são foco deste trabalho, juntamente com uma comparação entre elas. A seção 3 descreve o ambiente, a metodologia e os experimentos conduzidos nos ambientes com as aplicações *Word Count* e *PI Estimation*. Os resultados e sua análise são reproduzidos na seção 4. Por fim, as conclusões deste trabalho são apresentadas na seção 5.

2 Plataformas de *Big Data*

Esta seção apresenta as ferramentas de processamento de dados, no contexto de *Big Data*, que são foco deste trabalho (Figura 1). O Apache Spark é descrito na seção 2.1, juntamente com seus componentes (2.1.1) e com seu modelo de programação (2.1.2). O Apache Storm é abordado na seção 2.2, onde são apresentados os seus componentes (2.2.2) e o seu modelo de execução (2.2.1). A seção 2.3 detalha a abordagem utilizada pelo Apache Flink, sua organização interna (2.3.1), seus componentes (2.3.2) e seu modelo de execução (2.3.3). Finalmente na seção 2.4 as três plataformas são comparadas quanto às suas funcionalidades e modelos.

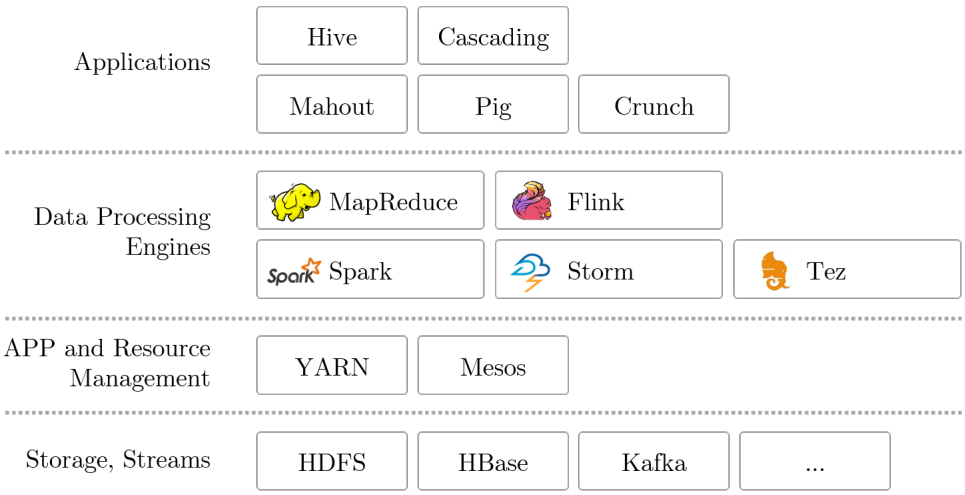


Figura 1: Panorama atual de *Big Data*

2.1 Apache Spark

Apache Spark é um sistema rápido e de uso geral para computação em *cluster*. Ele fornece APIs de alto nível em Java, Scala, Python e R, além de suportar gráficos de execução em geral [1]. Conforme

demonstrado pela Figura 2, ele também suporta um rico conjunto de ferramentas de alto-nível, incluindo Spark SQL para SQL e processamento de dados estruturados, MLlib para aprendizado de máquina, GraphX para processamento gráfico, e Spark Streaming para processamento de dados em tempo real.

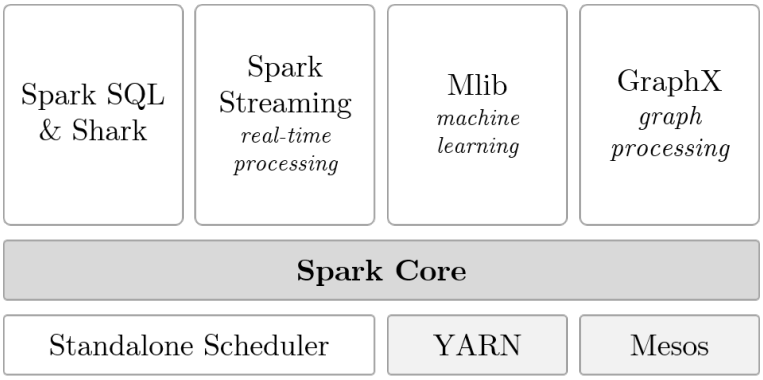


Figura 2: Spark *Stack*

Diversas empresas e organizações utilizam Spark em suas aplicações. Entre elas, podemos destacar: Amazon, Baidu, eBay Inc. (agregação de *logs* de transações e análises), Yahoo!, Grupon, NASA JPL - Deep Space Network e Yahoo! [2].

2.1.1 Componentes

Aplicações Spark executam como conjuntos de processos independentes em um *cluster*, coordenados pelo objeto *SparkContext* presente no programa principal (também chamado de programa *driver*). Para ser executado em um *cluster*, o *SparkContext* pode se conectar a vários gerenciadores (gerenciador próprio ou Mesos/YARN), que alocam recursos entre os programas. Uma vez conectado, o Spark adquire *executors* em nós do *cluster*, que são processos que realizam cálculos e armazenam dados da aplicação. Em seguida, ele envia o código da aplicação no formato JAR ou Python para estes. Finalmente, o *SparkContext* envia tarefas para os *executors*. Todo este processo está ilustrado na Figura 3.

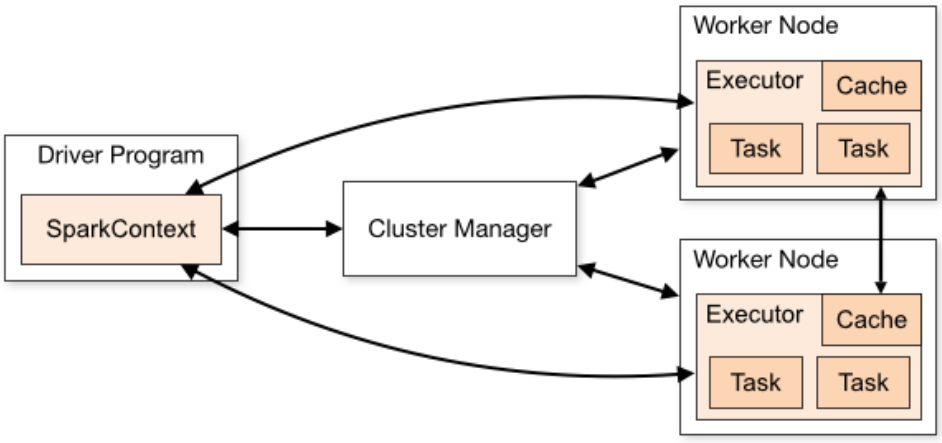


Figura 3: Componentes do Spark em um *cluster*

2.1.2 Modelo de Programação

Para utilizar o Spark, os desenvolvedores devem escrever um programa *driver* que implementa o controle de fluxo da aplicação em alto nível e inicia as operações em paralelo [3]. Duas abstrações

principais são utilizadas para descrever um programa em paralelo: *resilient distributed datasets* (RDDs) e operações paralelas sobre estes RDDs.

RDSs são coleções de objetos, com somente permissão de leitura, particionados entre todas as máquinas [4]. Estes sempre podem ser reconstruídos caso uma destas partições seja perdida. Na sua implementação, RDDs são objetos Scala que podem ser construídos a partir de arquivos de um sistema compartilhado (HDFS), da paralelização de uma coleção Scala (*array*), da transformação de outro RDD ou da alteração da persistência de um RDD já existente.

Dois tipos de operações podem ser aplicadas sobre RDDs: transformações e ações. A primeira cria novos conjuntos de dados a partir de um conjunto existente, enquanto que a segunda, após executar uma operação sobre o conjunto, retorna o valor ao programa *driver* [5]. Conforme apresentado em [6] as operações existentes são reproduzidas pela Figura 4.

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Figura 4: Tranformações e ações disponíveis nos RDDs do Spark

As principais operações paralelas que podem ser aplicadas em RDDs são: *reduce*, *collect* e *foreach* [3]. A operação *reduce* combina elementos dos conjuntos de dados utilizando uma função de associação. Já a operação *collect* envia todos os elementos de um conjunto de dados (RDD) para o programa *driver*. Por fim, a operação *foreach* aplica uma determinada função, informada pelo programador, em cada elemento de um RDD. Diferentemente de outras plataformas, o Spark não suporta a operação de redução de forma paralela, isto é, os resultados desta operação são coletados apenas pelo programa *driver*.

2.2 Apache Storm

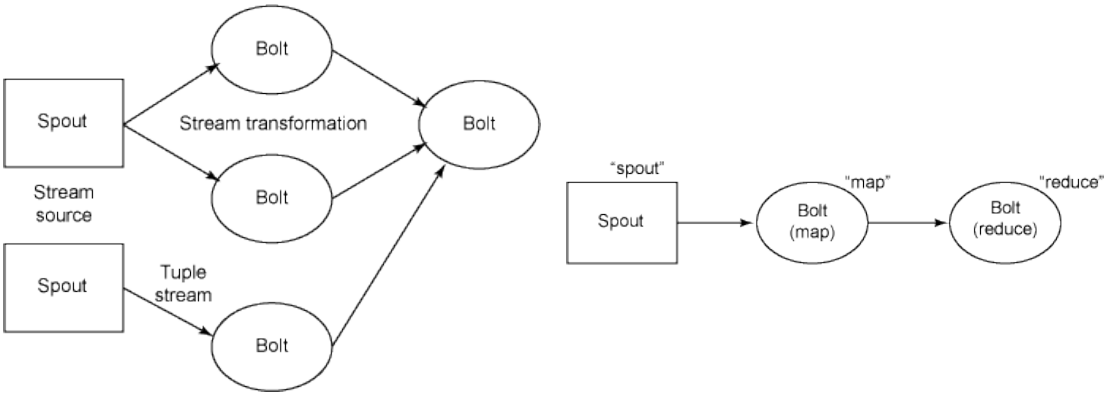
Storm é uma plataforma distribuída para o processamento de fluxo de dados em tempo real. [7]. Ele foi construído para ser escalável, resiliente, extensível, eficiente e fácil de administrar.

Diversas empresas e organizações utilizam o Apache Storm. Entre elas, podemos destacar: Twitter, Baidu, Spotify, Verisign, Yahoo! e Mercado Livre [8].

2.2.1 Componentes do Storm

Conforme apresentado em [7], a arquitetura de processamento de dados do Storm consiste em *streams* de tuplas percorrendo uma topologia. Uma topologia é um grafo direcionado (que admite ciclos) onde os vértices representam computação e as arestas representam o fluxo dos dados entre os componentes [9]. Por sua vez, os vértices podem ser divididos em duas categorias: *spouts* e *bolts*. A Figura 5(a) ilustra uma topologia no Storm e a Figura 5(b) exemplifica a topologia de um *MapReduce*.

Spouts são a origem do fluxo de dados em uma topologia. Eles são responsáveis por ler tuplas de uma fonte externa e inseri-las na topologia [10]. No aspecto tolerância a falhas, estes vértices



(a) Exemplo de topologia em Storm

(b) Topologia para MapReduce

Figura 5: Exemplo de topologias, *spouts* e *bolts* no Apache Storm

são divididos em confiáveis e não-confiáveis, onde os primeiros são capazes de reexecutar uma tupla caso o Storm tenha falhado ao processá-la.

Todo o processamento nas topologias é feito ns *bolts*. *Bolts* podem tem múltiplas funções que vão desde aplicação de filtros, funções, agregações, associações até comunicação com bancos de dados. Eles podem aplicar apenas transformações simples em streams. São necessários vários passos, e consequentemente vários *bolts* em transformações mais complexas.

2.2.2 Modelo de Execução

O Storm executa em um *cluster* distribuído. Clientes submetem topologias para execução a um nodo mestre chamado de *Nimbus*. Este é responsável por distribuir e coordenar a execução de uma topologia. O processamento em si é realizado pelos *workers*. Cada um destes pode executar um ou mais *worker processes* [7]. Por sua vez, cada *worker process* executa uma JVM com um ou mais *executors*, que são compostos por uma ou mais tarefas (*tasks*). O processamento de um *bolt* ou *spout* é realizado por estas tarefas. Estes conceitos são melhor ilustrados pela Figura 6.

O conceito de tarefas permite um paralelismo intra-*bolt*/intra-*spout* e o *executors* permitem paralelismo entre as topologias. Os processos denominados *worker processes* se comportam como *containers* nas máquinas que estão executando uma determinada topologia.

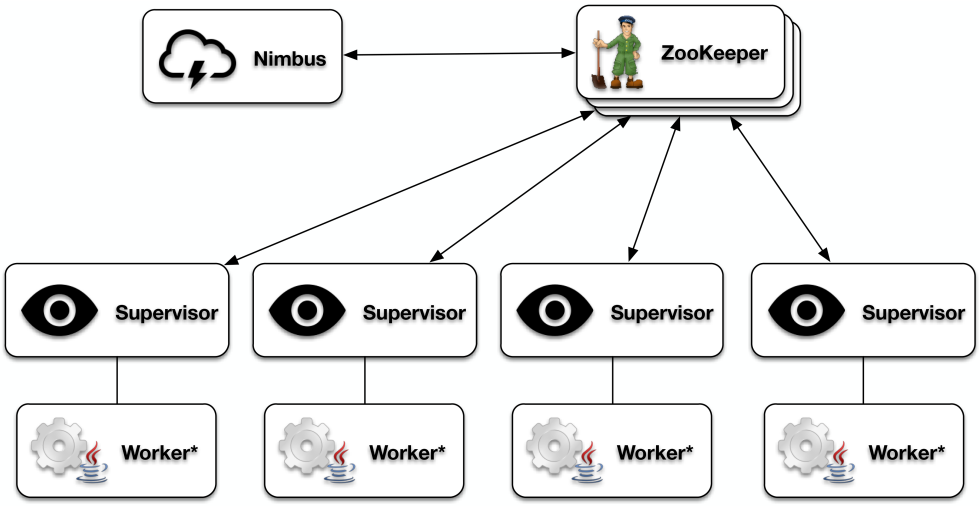


Figura 6: Modelo de execução do Apache Storm

2.3 Apache Flink

Apache Flink é uma plataforma para processamento de dados de forma eficiente, distribuída e de uso geral, que ainda está em fase de incubação pela Apache. Ele apresenta abstrações de programação em Java e Scala, um gerenciador de execução de alto desempenho e otimização automática de código. Ele tem suporte nativo para iterações, iterações incrementais e programas compostos por grandes DAGs de operações [11].

Programas de análise, em Flink, são programas normais que implementam transformações em conjuntos de dados (*data sets*). Estas transformações podem ser:

- *Map*: A partir de um elemento, gera um novo elemento (Figura 7);

```
data.map(new MapFunction<String, Integer>() {
    public Integer map(String value) {
        return Integer.parseInt(value);
    }
});
```

Figura 7: Trecho do código-fonte em Java para operação `map()` no Apache Flink

- *FlatMap*: A partir de um elemento produz zero ou mais elementos;
- *MapPartition*: Transforma uma partição paralela em uma única chamada de função. Esta função pega a partição como um objeto *Iterable* e pode produzir um número arbitrário de resultados. O número de elemento em cada partição depende do grau de paralelismo de operações anteriores;
- *Filter*: Avalia uma função booleana para cada elemento e mantém aqueles para os quais a função retorna verdadeiro;
- *Reduce*: Combina um grupo de elementos em um único elemento através da combinação de dois elementos repetidamente em um novo elemento (Figura 8);

```
data.reduce(new ReduceFunction<Integer> {
    public Integer reduce(Integer a, Integer b) {
        return a + b;
    }
});
```

Figura 8: Trecho do código-fonte em Java para operação `reduce()` no Apache Flink

- *ReduceGroup*: Combina um grupo de elementos em um ou mais elementos;
- *Aggregate*: Agrega um conjunto de valores em um único valor. As funções de agregação podem ser vistas como uma função de redução incorporada na plataforma;
- *Join*: Junta dois conjuntos de dados através da criação de todos os pares de elementos cujas chaves são iguais;
- *CoGroup*: Variante bidimensional da operação de redução;
- *Cross*: Constrói o produto cartesiano de duas entradas, criando todos os pares de elementos;
- *Union*: Produz a união de dois conjuntos de dados. Esta operação acontece implicitamente se mais de um conjunto de dados é usado como entrada de alguma função;
- *Rebalance*: Faz o balanceamento das partições paralelas de um conjunto de dados de forma uniforme;

- *Hash-Partition*: Particiona um conjunto utilizando *hash* em uma determinada chave de dados;
- *Custom Partitioning*: Permite definir manualmente um particionamento sobre os dados;
- *Sort Partition*: Ordena localmente um campo em uma determinada ordem em todas as partições de um conjunto de dados;
- *First-n*: Retorna os primeiros n elementos (arbitrários) de um conjunto de dados.

Os conjuntos de dados são inicialmente criados a partir de alguma fonte (leitura de arquivos ou coleções locais). Os resultados são retornados via *sinks* que podem escrever diretamente em um arquivo distribuído ou na saída padrão.

2.3.1 Flink *Stack*

As diferentes camadas da pilha do Flink são construídas de forma a aumentar o nível de abstração que as representações em programa aceitam (Figura 9).

Os tipos concretos e as interações com a biblioteca de execução são definidas nas camadas mais altas. A camada de API implementa múltiplas APIs que criam DAGs de operadores para os seus programas. Cada API necessita de serializadores, comparadores, etc, que descrevem a interação entre seus tipos de dados e a biblioteca de execução do Flink.

As camadas de API comum do Flink e do otimizador recebem o programa na forma de DAGs de operadores. Os operadores são específicos (*Map*, *Join*, *Filter*, *Reduce*, ...), mas os dados são variados.

A camada de execução (*runtime*) recebe um programa no formato *JobGraph*, que é uma representação paralela e genérica do fluxo de dados com tarefas arbitrárias que consomem e produzem *streams* de dados [12].

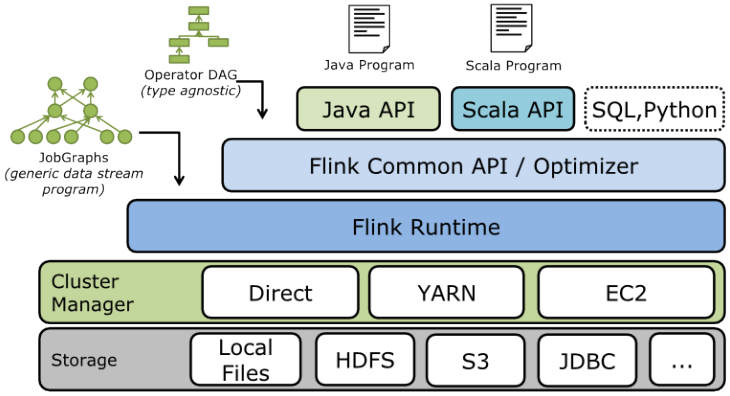


Figura 9: Flink *Stack*

2.3.2 Componentes

Logicamente o Flink segue o modelo mestre-escravo, através de dois componentes: o *JobManager* e os *TaskManagers*. O *JobManager* é o coordenador do sistema, enquanto que os *TaskManagers* são os trabalhadores que executam partes do programa paralelo (Figura 10).

Ao ser submetido para execução no ambiente, é criado um cliente que realiza um pré-processamento e transforma o programa em um fluxo de dados paralelo, que é então executado pelo *JobManager* e pelos *TaskManagers*.

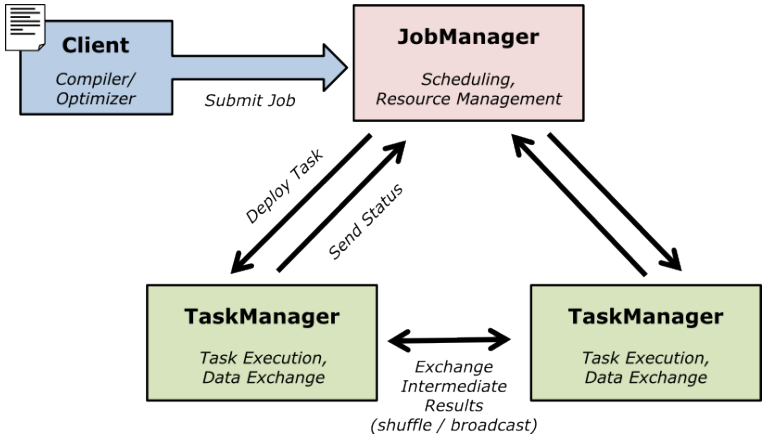


Figura 10: Interação entre os diferentes componentes do Flink

2.3.3 Modelo de Execução

No Flink, os recursos de execução são definidos através de *Task Slots*. Cada *TaskManager* possui um ou mais *slots* que podem executar tarefas em paralelo como em um *pipeline*. Como exemplo, pode-se utilizar as n instâncias paralelas de uma função de mapeamento juntamente com as n instâncias paralelas de uma função de redução. Estas tarefas podem vir a ser executadas de forma concorrente, principalmente em programas envolvendo *streaming* de dados.

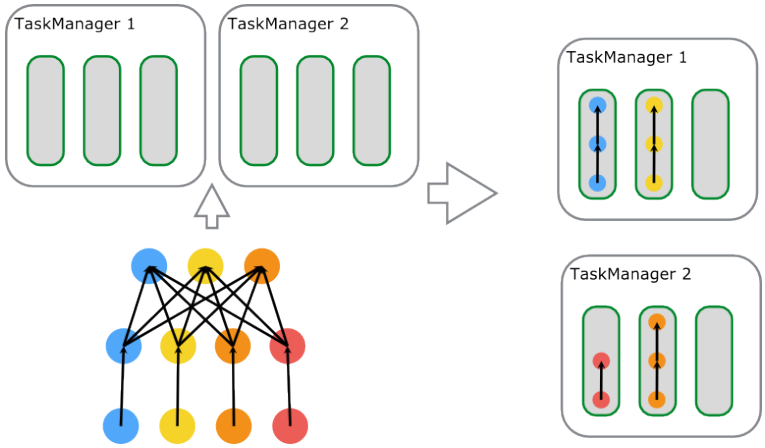


Figura 11: Exemplo de escalonamento de tarefas *map* e *reduce* no Flink

A Figura 11 demonstra o escalonamento de tarefas no Flink, considerando um *cluster* com 2 *TaskManagers*, cada um com 3 *slots*. As funções que atuam como origem dos dados e as funções de mapeamento possuem um paralelismo de 4, enquanto que as funções de redução são executadas com paralelismo de 3.

2.4 Comparações

As plataformas foco deste trabalho possuem algumas características em comum, mas também apresentam grandes diferenças, conforme é resumido pelo quadro da Figura 12.

3 Experimentos

Nesta seção estão descritos detalhes dos experimentos envolvendo as plataformas Flink e Spark. O ambiente utilizado é descrito na seção 3.1. As configurações do sistema de arquivo distribuído são

		Spark	Storm	Flink
PROJETO	Origem	UC Berkeley	BackType, Twitter	TU Berlin
	Implementação (*Core*)	Scala	Java/Clojure	Java
	Versão Atual	1.4.0 (11 Junho 2015)	0.9.5 (04 Junho 2015)	0.8.1 (19 Fevereiro 2015)
	Desenvolvimento	Projeto Completo Apache Software Foundation (ASF)	Projeto Completo Apache Software Foundation (ASF)	Projeto em Incubação Apache Software Foundation (ASF)
	Contribuidores*	565	144	97
	Utilização por Empresas	Amazon, Baidu, eBay Inc., Grupon, NASA JPL, Yahoo!	Alibaba, Baidu, Mercado Livre, Spotify, Twitter, Verisign, Yahoo!	Nenhuma
LINGUAGENS	Suporte a Linguagens	Java, Scala, Python, R	Todas (Java, Ruby, Python, JavaScript, Perl)	Java, Scala, Python
	Bibliotecas	Machine Learning (MLlib), SQL (Spark SQL), Graph and Parallel Graph (GraphX), Streaming (Spark Streaming)	io-storm (Perl), RedStorm (JRuby), Python, Ruby, storm-node (Node.js), Storm-PHP (PHP), ScalaStorm (Scala), SQLstream (SQL)	Graph API & Library (Gelly), Batch (Table API), Streaming (Table API)
	API	Spark Scala API, Spark Java API, Spark Python API, Spark R API		DataSet API (Scala, Java e Python), DataStream API (Java e Scala), Table API (SQL-*like* em Java e Scala)
EXECUÇÃO	Modo de Execução	Standalone, Hadoop, Mesos, Cloud (Amazon EC2)	Local, Cluster, e Cloud (Amazon EC2)	Standalone, Hadoop YARN, Apache Tez, Cloud (Amazon EC2, and Google Compute Engine)
	Modelo de Processamento	Micro-Batch	Event-Streaming, Micro-Batching (Storm Trident)	Batch, Streaming
	Gerenciamento	*SparkContext*	*Nimbus/Supervisors*	*JobManager/TaskManagers*
	Processamento	*Executors*	*Executors (threads)* nos *Workers*	*Workers*
MODELO	Transformações	Transformações em coleções de pares *key/value*	Transformações contínuas em tuplas	Transformações iterativas em coleções
	Paradigma	RDD	Grafo direcionado (topologias)	*Dataflow* Cíclico
	Iteracões	Programas iterativos são divididos em vários *jobs* independentes	Topologias podem conter ciclos	*Dataflows* com arestas de *feedback*, sistema está consciente das iterações, pode otimizar o *job*
DADOS	Acesso Dados	HDFS, Cassandra, HBase, Hive, Tachyon, Hadoop	HDFS, HBase, Cassandra	HDFS, Azure Storage, Tachyon, Amazon S3, XtreemFS, HBase, Google Cloud Storage, Flume, Kafka
	Streaming de Dados	HDFS, Flume, Kafka, Twitter, ZeroMQ, Custom	Kafka, Kestrel	Arquivos, Web Sockets, Message Queues (Apache Kafka, RabbitMQ, Twitter Streaming API), Custom (definido pelo usuário)
STREAMING	*Streaming*	Processamento *streaming* em cima do sistema de *batch*	*Streaming* verdadeiro	*Streaming* verdadeiro
	Streaming Latência	Alta	Baixa	Ajustável
	Streaming Throughput	Alto	Baixo	Ajustável

* Número de contribuidores no repositório oficial do projeto no GitHub

Figura 12: Comparação das Plataformas Spark, Storm e Flink

apresentados na sessão 3.1.1. Por sua vez, nas seções 3.1.2 e 3.1.3 são apresentadas as configurações do Flink e Spak respectivamente. A metodologia utilizada nos experimentos é abordada na seção 3.2, juntamente com detalhes sobre as duas aplicações em análise: *Word Count* (3.2.1) e *PI Estimation* (3.2.2).

3.1 Ambiente

Todos os testes aqui apresentados foram executados no cluster Parasilo do Grid5000 [13]. Este cluster é composto por 28 nós (2 × CPUs Intel Xeon E5-2630v3, 8 cores/CPU, 128GB RAM, 5 × 600GB HDDD) interligados por uma rede 10 Gigabit Ethernet. No contexto dos experimentos foram utilizados apenas 25 nós.

3.1.1 Configurações HDFS

O Hadoop Distributed File System (HDFS) é um sistema de arquivos distribuído. Ele possui várias semelhanças com outros sistemas existentes, no entanto, as diferenças são significativas. O HDFS é altamente tolerante a falhas e é projetado para ser implantado em hardware de baixo custo. Ele provê altas taxas de transferência de dados da aplicação e é adequado para aplicações que trabalham com grandes conjuntos de dados. O HDFS relaxa alguns requisitos POSIX para permitir acesso *streaming* aos arquivos do sistema [14].

Em todos os experimentos foi utilizado a última versão do HDFS (2.7.0) como sistema de arquivo padrão.

3.1.2 Configurações Flink

Nesta seção estão descritas as configurações do Flink para os experimentos envolvendo as aplicações *Word Count* e *PI Estimation*.

Os arquivos de configuração do Flink estão no diretório `flink/conf`. Dois arquivos foram modificados, o primeiro (`flink/conf/slaves`) contém uma descrição com todos os nós escravos. O segundo arquivo (`flink/conf/flink-conf.yaml`), está parcialmente reproduzido na Figura 13 onde estão indicados os parâmetros que foram configurados. É importante observar, além do endereço do mestre, a quantidade de memória alocada e o grau de paralelismo para aplicações que não fazem esta definição diretamente no código.

```
jobmanager.rpc.address: parasilo-2.rennes.grid5000.fr

jobmanager.rpc.port: 6123

jobmanager.heap.mb: 65536 # 64GB

taskmanager.heap.mb: 65536 # 64GB

taskmanager.numberOfTaskSlots: 16

parallelization.degree.default: 384 # 24 * 16

taskmanager.network.numberOfBuffers: 24576

fs.hdfs.hadoopconf: /root/hadoop/etc/hadoop/
```

Figura 13: Arquivo de configuração do Apache Flink

3.1.3 Configurações Spark

Nesta seção estão descritas as configurações do Spark para os experimentos envolvendo as aplicações *Word Count* e *PI Estimation*.

Os arquivos de configuração do Spark estão no diretório `spark/conf`. Dois arquivos foram modificados, o primeiro (`spark/conf/slaves`) contém uma descrição com todos os nós escravos. O segundo arquivo (`spark/conf/spark-default.conf`), está reproduzido na Figura 14. É importante observar a definição do endereço do mestre e a quantidade de memória alocada, que nos experimentos foi de $64GB$.

```
spark.master                     spark://parasilo-2.rennes.grid5000.fr:7077
spark.eventLog.enabled           true
spark.eventLog.dir               hdfs://parasilo-2.rennes.grid5000.fr:8020/directory
spark.serializer                 org.apache.spark.serializer.KryoSerializer
spark.driver.memory              64g
spark.executor.memory            64g
```

Figura 14: Arquivo de configuração do Apache Spark

3.2 Metodologia

Como o objetivo é avaliar o desempenho das diferentes plataformas, foi utilizado o *wall time* para medir o tempo de execução dos diferentes programas. Esta medida foi considerada pois, na perspectiva do usuário, é uma das mais importantes.

Neste sentido foram avaliadas duas aplicações: *Word Count* e *PI Estimation*. A primeira trabalha com grandes volumes de dados e a segunda é computacionalmente intensiva. O principal motivo para a escolha destas aplicações para avaliação de ambas plataformas foi a disponibilidade de ambas como exemplo padrão e a similaridade da implementação utilizando Flink e Spark.

Os códigos utilizados nestes experimentos foram os exemplos, inclusos na distribuição da versão atual do Spark e do Flink, desenvolvido pelas mesmas equipes que criaram cada projeto. Optou-se em utilizar estes códigos pois considera-se que os mesmos estejam otimizados para melhor executar sobre sua plataforma.

3.2.1 Word Count

É possível mapear e reduzir dados com base em uma variedade de critérios. Um exemplo comum é o WordCount. Como o nome sugere ele mapeia as palavras dado uma entrada e reduz (resume) os resultados com uma contagem do número de instâncias de cada palavra.

A implementação do *Word Count*, em Java, recebe dois parâmetros informando o caminho do arquivo de entrada e o caminho do arquivo de saída. Como utilizou-se o HDFS como sistema de arquivos padrão, os caminhos informados seguiam o padrão: *hdfs://horstname:port/folder/filename.txt*.

Os arquivos utilizados nos experimentos possuem tamanhos de $9GB$, $18GB$ e $27GB$. Com o objetivo de demonstrar e avaliar a complexidade do código, o conteúdo da função principal é reproduzido pela Figura 15.

```
// Configura o ambiente de execucao
final ExecutionEnvironment env =
    ExecutionEnvironment.getExecutionEnvironment();

// Dados de entrada
DataSet<String> text = getTextDataSet(env);

DataSet<Tuple2<String, Integer>> counts =
  // Divide as linhas em pares (tuplas) contendo: (word,1)
  text.flatMap(new Tokenizer())
  // Agrupa os resultados pelo campo "0" da tupla
  .groupBy(0)
  // Soma os resultados pelo campo "1" da tupla
  .sum(1);

// Escreve o resultado
counts.writeAsCsv(outputPath, "\n", " ");

// Executa o programa
env.execute("WordCount Example");
```

Figura 15: Trecho do código-fonte para o *Word Count* em Flink

Dependendo dos parâmetros, como tamanho dos dados e número de máquinas no *cluster*, o

otimizador interno do Flink automaticamente seleciona uma estratégia de execução para o programa. Saber exatamente como o Flink executa o programa é, muitas vezes, útil. Para fazer esta visualização, o Flink utiliza uma ferramenta própria que pode ser acessada através da interface Web. A Figura 16 ilustra as operações e a movimentação de dados do *Word Count*, utilizado neste trabalho, gerada pela ferramenta.

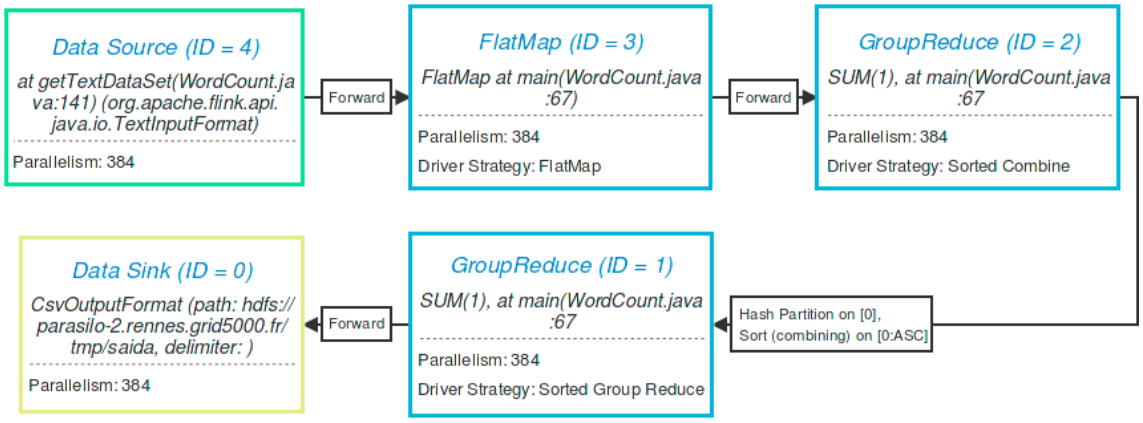


Figura 16: Plano de execução do *Word Count* gerado pelo Flink

Com o objetivo de avaliar a implementação do código em Spark, reproduzimos, na Figura 17, apenas o conteúdo da função principal.

```
String inputFile = args[0];
String outputFile = args[1];

// Cria um Java Spark Context
SparkConf conf = new SparkConf().setAppName("WordCount");
JavaSparkContext sc = new JavaSparkContext(conf);

// Carrega os dados de entrada
JavaRDD<String> input = sc.textFile(inputFile);

// Separa em palavras
JavaRDD<String> words = input.flatMap(
    new FlatMapFunction<String, String>() {
        public Iterable<String> call(String x) {
            return Arrays.asList(x.split(" "));
        }
    });

// Transforma em palavras e faz a conta
JavaPairRDD<String, Integer> counts = words.mapToPair(
    new PairFunction<String, String, Integer>(){
        public Tuple2<String, Integer> call(String x){
            return new Tuple2(x, 1);
        }
    }).reduceByKey(new Function2<Integer, Integer, Integer>() {
        public Integer call(Integer x, Integer y) {
            return x + y;
        }
    });

// Grava o resultado em um arquivo .txt
counts.saveAsTextFile(outputFile);

sc.stop();
```

Figura 17: Trecho do código-fonte para o *Word Count* em Spark

3.2.2 PI Estimation

Este código Java utiliza o método de Monte Carlo para aproximar o valor de π. Ele recebe como entrada um parâmetro que indica o número de amostras a ser utilizado no cálculo. Neste experimento foram considerados 10, 100 e 1000 amostras.

A Figura 18 reproduz o conteúdo da função principal, em Java, utilizando o Flink

```
final long numSamples = args.length > 0 ? Long.parseLong(args[0]) : 1000000;

final ExecutionEnvironment
    env = ExecutionEnvironment.getExecutionEnvironment();

// Conta-se quantos pontos aleatorios estao contidos na circunferencia
DataSet<Long> count =
    env.generateSequence(1, numSamples)
    .map(new Sampler())
    .reduce(new SumReducer());

DataSet<Double> pi = count
    .map(new MapFunction<Long, Double>() {
        public Double map(Long value) {
            return value * 4.0 / numSamples;
        }
    });

env.execute();
```

Figura 18: Trecho do código-fonte para o *PI Estimation* em Flink

A Figura 19 apresenta o conteúdo da função principal do código em Java, utilizando o Spark.

```
SparkConf sparkConf = new SparkConf().setAppName("JavaSparkPi");
JavaSparkContext jsc = new JavaSparkContext(sparkConf);

int slices = (args.length == 1) ? Integer.parseInt(args[0]) : 2;
int n = 100000 * slices;

List<Integer> l = new ArrayList<Integer>(n);

for (int i = 0; i < n; i++) {
    l.add(i);
}

JavaRDD<Integer> dataSet = jsc.parallelize(l, slices);

int count = dataSet.map(new Function<Integer, Integer>() {
    @Override
    public Integer call(Integer integer) {
        double x = Math.random() * 2 - 1;
        double y = Math.random() * 2 - 1;
        return (x * x + y * y < 1) ? 1 : 0;
    }
}).reduce(new Function2<Integer, Integer, Integer>() {
    @Override
    public Integer call(Integer integer, Integer integer2) {
        return integer + integer2;
    }
});

System.out.println("PI " + 4.0 * count / n);

jsc.stop();
```

Figura 19: Trecho do código-fonte para o *PI Estimation* em Spark

4 Resultados

Os testes com a aplicação *Word Count* foram repetidos até que se atingisse 95% de confiança com uma distribuição *t-student* - com pelo menos 90 execuções. O erro máximo aceito foi de 5%. Por sua vez, os testes com a aplicação *PI Estimation* foram repetidos até que se atingisse 90% de confiança com uma distribuição *t-student*, com pelo menos 90 execuções. O erro máximo aceito foi de 10%.

A Figura 20 apresenta um comparativo base entre Apache Flink, Apache Spark e Apache Hadoop para a aplicação *Word Count* com arquivo de entrada de $9GB$.

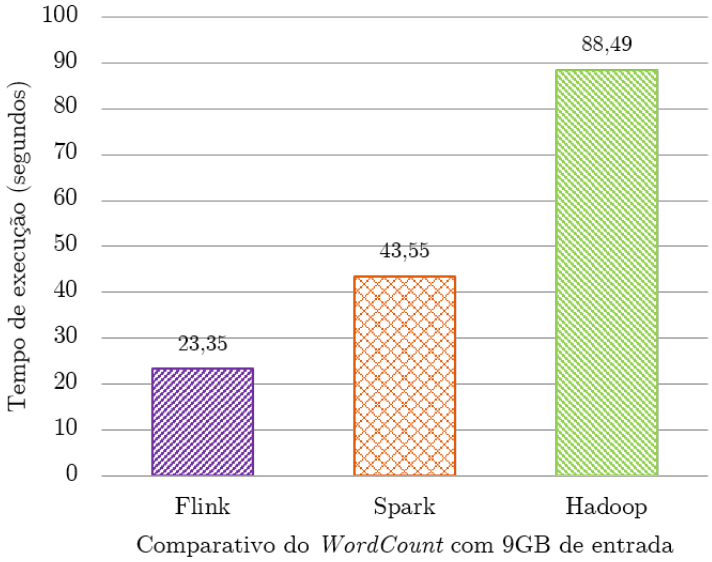


Figura 20: Comparação do tempo de execução do *Word Count* no Flink, Spark e Hadoop

Conforme pode ser observado na Figura 21, o Apache Flink apresenta desempenho melhor, nesta aplicação, do que o Apache Storm. O *speedup* foi de 1.86 no caso de teste com $9GB$, 1.57 no de $18GB$ e 1.12 no de $27GB$ de entrada.

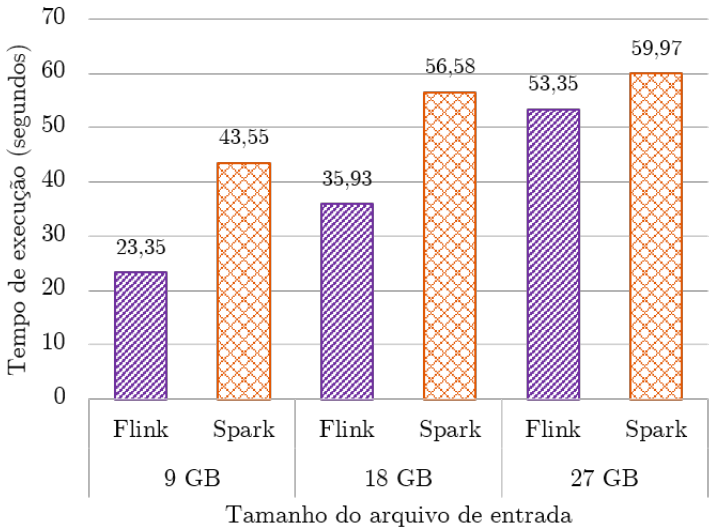


Figura 21: Comparação do tempo de execução do *Word Count* no Spark e no Flink

Já no teste com o *PI Estimation*, o Apache Flink apresentou tempo de execução com crescimento próximo ao linear quando o número de amostras aumentou, se comparado com o Apache Spark. O *speedup* foi de 3.31 no caso de teste com 10 amostras, 3.53 no de 100 e 7.30 no de 1000 de entrada.

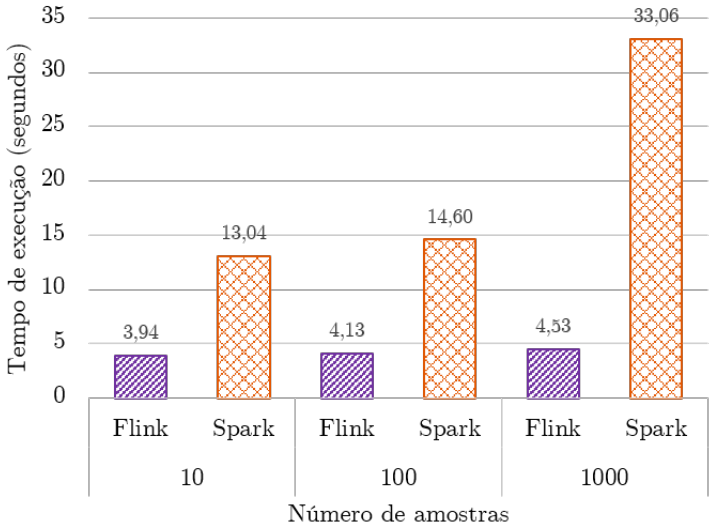


Figura 22: Comparação do tempo de execução do *PI Estimation* no Spark e no Flink

Por ser relativamente novo, o Flink ainda não coleta dados referente a comunicação e I/O. Os únicos dados disponíveis em seus arquivos de *log* de execução são o tempo de cada *task* em um *job*. A Figura 23 demonstra o tempo médio de cada *task* em um dos testes com a aplicação *Word Count*.

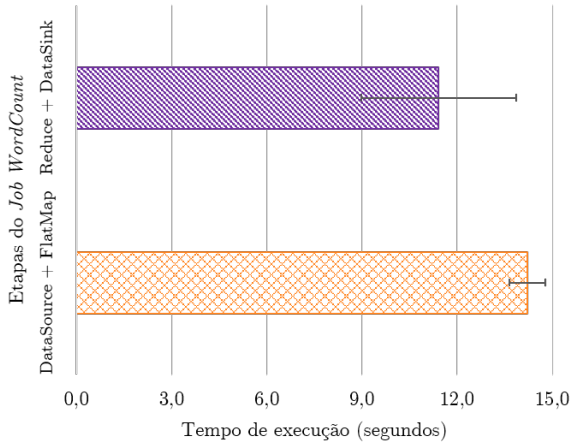


Figura 23: Tempo de execução das tarefas de um *job WordCount* com $9GB$ no Flink

Utilizando a ferramenta Trace Analysis[1], com base nos *logs* de execução do Spark foi possível gerar a Figura 24, que apresenta o comportamento das tarefas de mapeamento e redução do *Word Count*.

[1]Disponível em: https://github.com/kayousterhout/trace-analysis

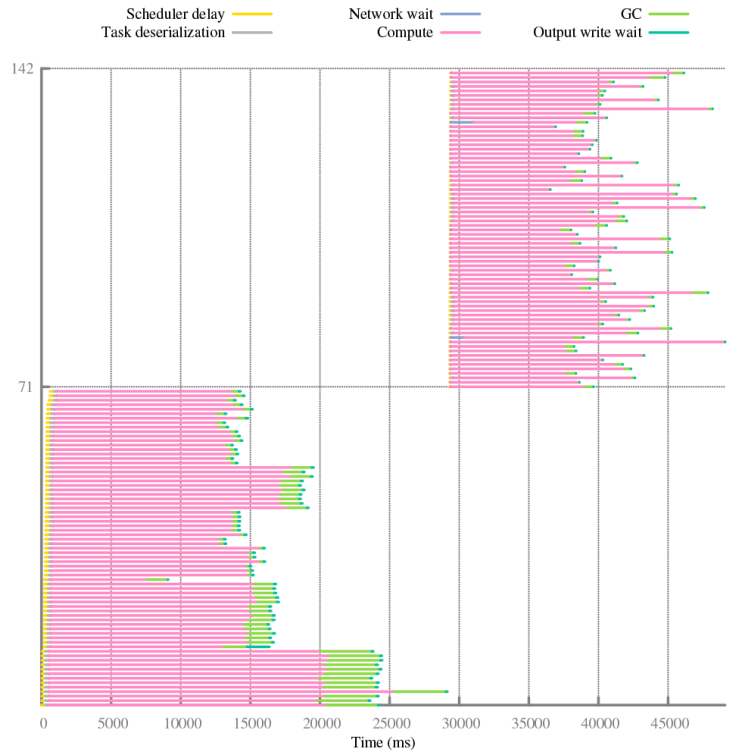


Figura 24: *Trace* de uma execução do *WordCount* com $9GB$ no Spark

5 Conclusões

Apesar de ser uma plataforma relativamente nova, o Flink apresentou excelentes resultados, quando comparado ao Apache Spark. Em uma aplicação que trabalha com grandes quantidades de I/O, como o *Word Count*, o *speedup* foi de até 1.86. Já em uma aplicação como o *PI Estimation*, o tempo de execução do Flink se manteve quase constante ao aumentar o número de amostras, diferentemente do observado utilizando o Spark. Neste caso o *speedup* máximo observado foi de 7.30.

Devido a sua natureza relativamente nova, o Flink ainda não possui suporte para capturar e gerar estatísticas de comunicação e de I/O como o Spark tem. Por esta razão estes parâmetros não foram analisados e comparados.

Referências

[1] Apache.org, "Apache Spark 1.4.0," 2015. Acessado em: 12 maio 2015.

[2] A. Konwinski, "Powered By Spark," 2015. Acessado em: 20 junho 2015.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2010.

[4] A. Spark, "Resilient Distributed Datasets RDDs," 2015. Acessado em: 23 junho 2015.

[5] A. Spark, "RDDs Operations," 2015. Acessado em: 21 junho 2015.

[6] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2012.

[7] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@Twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, (New York, NY, USA), pp. 147–156, ACM, 2014.

[8] A. Storm, "Companies Using Apache Storm," 2015. Acessado em: 20 junho 2015.

[9] A. Storm, "Storm Simple API," 2015. Acessado em: 21 junho 2015.

[10] A. Storm, "Apache Storm Concepts," 2015. Acessado em: 12 maio 2015.

[11] Apache.org, "Apache Flink 0.9," 2015. Acessado em: 28 maio 2015.

[12] Apache.org, "Apache Flink 0.9 - General Architecture and Process Model," 2015. Acessado em: 19 junho 2015.

[13] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard, "Grid'5000: A Large Scale and Highly Reconfigurable Grid Experimental Testbed," in *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, (Washington, DC, USA), pp. 99–106, IEEE Computer Society, 2005.

[14] Apache.org, "HDFS Architecture Guide," 2015. Acessado em: 23 junho 2015.