PROCESSAMENTO DE BIG DATA: UMA REVISÃO

BIG DATA PROCESSING: A REVIEW

Adriano L. L. da Silva[*]
Dayse S. Almeida[**]

RESUMO

Um dos principais desafios no tratamento de dados em larga escala atualmente produzidos, chamados big data, está relacionado ao processamento. O grande volume desses conjuntos de dados inviabiliza o uso das tecnologias tradicionais de processamento e análise, especialmente quando essas tecnologias são projetadas para ambientes centralizados. Neste artigo, apresentamos uma revisão de modelos de programação distribuídos adequados para processamento de big data. Entre esses modelos está o modelo de programação MapReduce, que é apresentado como uma alternativa simples e poderosa para o processamento de grandes quantidades de dados.

Palavras-chave: MapReduce. Processamento distribuído. BigData.

ABSTRACT

One of the major challenges in the treatment of large-scale data currently produced, called big data, is related to processing. The large volume of these data sets makes impracticable the use of traditional processing and analysis technologies, especially when these technologies are designed for centralized environments. In this paper we present a review of distributed programming models suitable for big data processing. Among these models is the MapReduce programming model, which is presented as a simple and powerful alternative for processing large amounts of data.

Keywords: MapReduce. Distributed Processing. BigData.

Introdução

Com o avanço do poder computacional, dos sistemas de informação e o aumento da quantidade de serviços disponibilizados ao usuário, cresceu também o volume de dados gerados e que precisam ser processados pelos sistemas computacionais, iniciando assim a tendência conhecida como *Big Data*. Dentro desse contexto, novas tecnologias vêm sendo projetadas e desenvolvidas com o objetivo de tratar essa quantidade massiva de dados.

[*] Departamento de Ciências da Computação – Universidade Federal de Goiás (UFG). adriano.lopes10@discente.ufg.br
[**] Departamento de Ciências da Computação – Universidade Federal de Goiás (UFG). daysesa@gmail.com

Introduzido nesse conjunto de tecnologias que surgiram se encontra o MapReduce, um método paralelo e distribuído que utiliza duas funções advindas das linguagens funcionais, *Map* e *Reduce*, para processamento de dados. A função *Map* tem como objetivo aplicar uma ação descrita pelo programador sobre cada dado de um agrupamento definido como entrada. Já a função *Reduce* tem como objetivo aglomerar os dados semelhantes identificados anteriormente pelo mapeamento.

Existem várias maneiras de executar esse método MapReduce paralelamente e de maneira distribuída. O Hadoop MapReduce e o Spark são as *frameworks* mais conhecidas e possuem uma maneira de trabalho parecida. Porém existem outras maneiras de executar o método de programação, como proposto no Phoenix (RANGER *et al.*, 2007; YOO; ROMANO; KOZYRAKIS, 2009), que utiliza de programação paralela em um mesmo processador, com a distribuição das tarefas entre as *threads*, e tem como benefício a utilização de apenas um recurso de processamento, porém pode afetar a capacidade máxima de trabalho.

No artigo (ALMEIDA; DUTRA, 2019) foi abordado o armazenamento de big data e neste são discutidos o conceito de MapReduce e as *frameworks* que se utilizam desse conceito, bem como outras formas de processamento distribuído. O objetivo é promover uma visão geral sobre o tema bem como uma comparação entre as *frameworks* que utilizam o método MapReduce com outros modelos de processamento mais tradicionais, concluindo-se sobre quais os modelos de processamento adequados para o *Big Data* e os desafios encontrados na área. Para isso, o artigo está organizado da seguinte maneira. Na Seção 2 é descrito o modelo MapReduce. Na Seção 3 são abordadas as principais tecnologias que utilizam o modelo de MapReduce para processamento de *Big Data*. Na Seção 4 são discutidos outros modelos de processamento distribuído. Na Seção 5 é apresentada a conclusão do trabalho com uma ênfase na importância de se utilizar o MapReduce para os atuais conjuntos de dados a fim de se melhorar o processamento e obter informações úteis no tempo exigido pelas atuais aplicações.

1 Modelo de processamento MAPREDUCE

O MapReduce é um modelo de programação distribuída eficiente para processamento e análise de grandes conjuntos de dados. Sua estrutura permite a execução de aplicações em paralelo, alocação dos dados em bancos de dados distribuídos e a possibilidade de comunicação em rede com tolerância a falhas de forma eficaz, através

da utilização de replicação dos dados entre os nós de um *cluster* (DEAN; GHEMAWAT, 2008).

Esse modelo é descrito através de uma biblioteca composta por duas funções, que são responsáveis por todo o processamento dos dados nesse paradigma. A primeira função é denominada *Map* e tem como entrada um conjunto de pares chave/valor. Ao aplicar uma ação descrita pelo programador sobre cada dado de entrada, ela retorna para a parte intermediária do processo outro conjunto de pares chave/valor. Na etapa intermediária, os dados são organizados por um *iterador* para facilitar a leitura e o encaminhamento para a função de *Reduce* que, assim como a fase de mapeamento e a fase intermediária, é executada paralelamente em um *cluster*. Nessa fase de *Reduce*, a entrada dos dados também é feita no formato de pares chave/valor na qual a função, após a leitura, executa os procedimentos para o qual foi programada, equivalente a redução dos dados que foram mapeados anteriormente.

Na Figura 1 é mostrado um exemplo no qual o método MapReduce é utilizado para a contagem de acesso aos *sites* do Facebook, de busca da Google, da Globo, do Youtube, da Magazine Luiza, do AliExpress e da Wikipédia. Primeiramente, o MapReduce divide em blocos a entrada dos dados, encaminhando cada um desses blocos à um dos nós do *cluster*. Esses blocos são encaminhados no formato de pares chave/valor, sendo que as chaves estão representadas pelas cores dos blocos e os valores são os dados contidos neles. Posteriormente, a função *Map* descrita pelo programador realiza o mapeamento dos valores de entrada, gerando então a sua saída no formato de pares chave/valor, sendo agora as chaves cada um dos valores contidos nos blocos e, os valores, uma atribuição unitária de contagem. Logo após esse processo, os dados são encaminhados para uma fase chamada de *Iterador*, na qual os pares chave/valor serão agrupados por chaves em blocos separados. Isso é feito com objetivo de aumentar o desempenho da fase de redução. No processamento da função de *Reduce*, também descrita pelo programador, ocorre apenas uma junção dos valores, os quais as chaves foram mapeadas e agrupadas anteriormente. A leitura dessa fase também é realizada usando o formato de pares chave/valor, bem como a saída, produzindo como resultado uma contagem de acessos à cada *site* mostrado na figura.

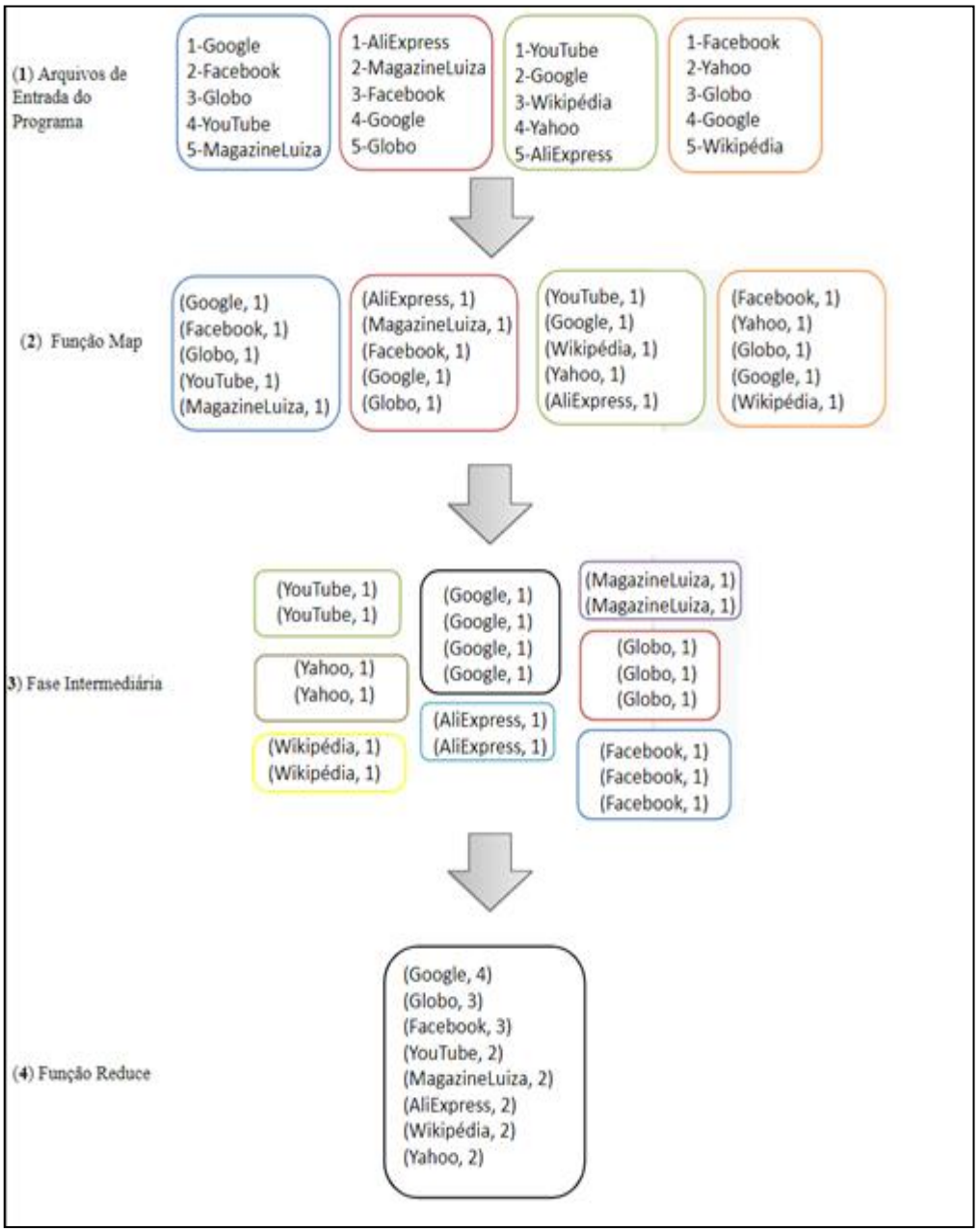

Figura 1. Execução do MapReduce para contagem de acessos aos *sites*

A implementação mais comum do MapReduce é de forma distribuída usando *clusters* de computadores como recurso para processar a grande quantidade de dados da entrada. Porém existem outras formas de implementar esse paradigma, como por exemplo, na implementação feita para sistemas *multi-core* e multiprocessadores, que utiliza *threads* de memória compartilhada para construir o paralelismo. Essas implementações são discutidas na Seção 3.

2 Frameworks que utilizam o modelo MapReduce

2.1 Hadoop MapReduce

A estrutura de execução e programação de código aberto mais conhecida do modelo de programação MapReduce, proposto pela Google é o Hadoop MapReduce da Apache (DEAN; GHEMAWAT, 2008).

O Hadoop é um ecossistema utilizado para processamento de dados de forma distribuída, ou seja, utiliza várias unidades de processamento para obter o poder computacional necessário para resolver um problema, sendo o conjunto dessas unidades chamado de *cluster*.

Aprofundando no funcionamento do Hadoop Mapreduce, tem-se um nó no *cluster* denominado como *master*, sendo esse o responsável por monitorar os demais nós do *cluster* controlando os estados e fazendo o tratamento de falhas. Os nós que são passíveis de gerenciamento pelo *master* são chamados de *workers*. Sua tarefa é executar as funções *Map* e *Reduce* de forma paralela com uma determinada parte dos dados. Devido ao fato de a quantidade de dados recebidos pelo *framework* serem grandes, o mesmo faz uma divisão dos dados em partes menores, guardando assim somente a localização dessas partes no nó *master*. Os dados são processados pelos *workers* da tarefa *Map* após receber essa informação de localização a partir do nó *master*. Os *workers Reduce* leem os dados encaminhados pelo *iterador* de forma direta. Já a saída dessa fase é armazenada em um arquivo de saída.

O *Hadoop Distributed File System* (HDFS) e a programação MapReduce fornecem à trabalhos analíticos intensivos de *Big Data* uma arquitetura escalável e capacidade de processar dados numa forma paralela em *cluster*. O Hadoop permite o processamento paralelo de uma vasta quantidade de dados, distribuição de dados, tolerância a falhas e balanceamento de carga, o que resulta em uma computação confiável e escalável.

O desenvolvimento das formas de processamento distribuídas de dados se deve ao fato de as ferramentas tradicionais terem se tornado ineficientes para processar grandes quantidades de dados. O *Hadoop* obteve bons resultados ao ser aplicado no problema de manipular e processar terabytes e petabytes de dados, que são produzidos todos os dias por grandes empresas de tecnologia, como Google, Facebook, Yahoo, entre outras (DITTRICH; QUIANÉ-RUIZ, 2012).

2.2 Spark

Devido às características do *Big Data* quanto à variedade dos formatos dos dados e ao volume, foram criados diversos sistemas para processar, tratar e consultar essas massivas quantidades de dados. O Spark (ZAHARIA *et al.*, 2010) se encaixa nesse contexto, trazendo uma unificação do processamento distribuído de dados, de forma que este seja realizado em uma única plataforma. Essa característica torna o Spark mais adequado para o contexto de sistemas interativos que as demais implementações do MapReduce.

Na utilização do Spark, os desenvolvedores geram um programa de *driver* que monta o fluxo de controle de alto nível de sua aplicação e inicia várias operações em paralelo. São duas as abstrações para programação paralela que o Spark oferece: (i) conjuntos de dados distribuídos resilientes ou RDD, que são coleções particionadas de objetos somente de leitura, e; (ii) operações paralelas nesses conjuntos de dados, sendo essas operações voltadas para cálculos sobre o conjunto de dados. Os objetos pertencentes aos RDDs não precisam ser armazenados fisicamente pois o identificador do RDD contém informações para que ele possa ser gerado a partir de dados armazenados persistentemente. Por essa razão, os RDDs também podem ser recriados caso uma das partições seja perdida.

Os RDDs no *Spark* são construídos como objetos da linguagem funcional *Scala* e podem ser criados de quatro formas:

- A partir de um arquivo disponível em um sistema de arquivos compartilhado, como o HDFS.
- Paralelizando uma coleção Scala no programa driver, de forma que ela seja particionada e enviada a vários nós.
- Utilizando a aplicação de uma função sobre um conjunto de dados para gerar um novo conjunto de dados, por meio da operação flatMap, advinda da própria linguagem. Essa operação permite que dada uma função descrita pelo usuário e um conjunto de entrada sejam utilizadas para gerar os dados do conjunto resultante. Esse conjunto é resultado da aplicação da função sobre a coleção de origem. Uma função chamada filter também é utilizada determinar quais dados devem pertencer ao conjunto resultante.
- Por padrão, as partições são materializadas sob demanda de acordo com a sua necessidade em operações paralelas. A persistência dos RDDs pode então, ser

> modicada das seguintes maneiras: (i) colocando o conjunto de dados em cache, sugerindo que sejam mantidos na memória após a primeira execução (onde são calculados), para que os dados possam ser reutilizados posteriormente, ou; (ii) gravando o RDD em um sistema de arquivos distribuído diretamente.

Comparado com os outros frameworks de MapReduce, o Spark se mostra mais flexível por fornecer diversos níveis de persistência ao usuário de acordo com as necessidades da sua aplicação.

2.3 Phoenix

O Phoenix (RANGER *et al.,* 2007) é uma implementação MapReduce para multiprocessadores de chip único e multiprocessadores de memória compartilhada. Tendo em vista a alta complexidade de programação nesses sistemas baseados em paralelismo em *threads*, uma alternativa para o processamento é confiar no sistema operacional para gerenciar a simultaneidade. Em sua primeira versão o Phoenix obteve bom desempenho em sistemas pequenos com baixa latência de acesso uniforme. Porém, após o Phoenix ser utilizado, foi constatada a necessidade de processar sistemas de larga escala e característica de acesso não uniforme (NUMA).

Uma versão posterior do Phoenix (YOO; ROMANO; KOZYRAKIS, 2009) teve seu tempo de execução otimizado em um sistema com processador *quadcore* de 32 núcleos e 256 *threads* com memória compartilhada. Para uma execução eficiente, também é necessária uma abordagem de otimização em camadas, na qual o desenvolvedor deve otimizar seus algoritmos em torno do desafio do NUMA.

Na Figura 2 é mostrado o funcionamento do processamento no Phoenix. Nas etapas 1 e 2 o usuário entra com as funções de *Map* e *Reduce*, respectivamente. Os dados armazenados em um Banco de Dados ou Sistemas de Arquivos, por exemplo, são divididos em blocos para facilitar a sua manipulação, tendo em vista que se trata de grandes volumes. As etapas de armazenamento e particionamento dos dados são mostradas em 3 e 4. Na etapa 5, os blocos de dados são divididos em *threads* do processador para serem processados de acordo com a função *Map* descrita pelo usuário. Na etapa 6, os dados resultantes da aplicação da função *Map*, são separados novamente na forma de blocos. Por fim, na etapa 7, os dados são processados de acordo com a função *Reduce* também descrita pelo usuário, gerando a saída. Essa saída pode ser devolvida ao

sistema de armazenamento utilizado, encaminhada para outro processamento ou utilizada de outra forma por um sistema de análise, por exemplo.

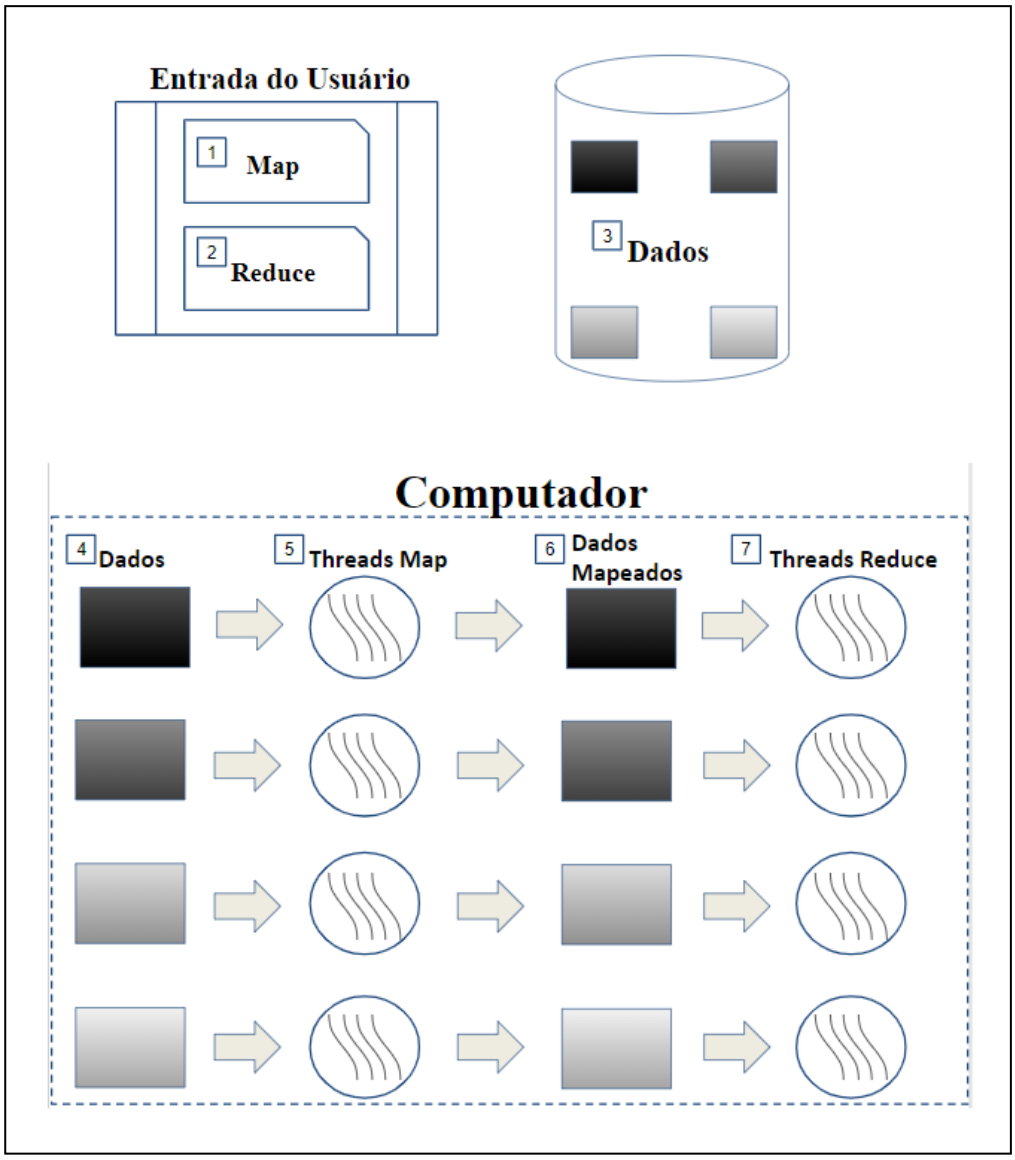

Figura 2. Modelo de execução do Phoenix MapReduce

2.3 Pregel

Sistemas de processamento de grafo são projetados, normalmente, para realizar computações *off-line* em grafos muito grandes, usando um ambiente distribuído. Esses sistemas são focados na computação sobre o grafo e não em consultas sobre o grafo.

O Pregel (MALEWICZ *et al*., 2010) é um *framework* de programação distribuída, desenvolvido pela Google, conceitualmente similar ao MapReduce. No entanto, o Pregel foi desenvolvido para facilitar o processamento de grafos de grande porte, como por

exemplo, análise em grafos de redes sociais, nas quais o número de vértices chega a bilhões e o número de arestas a trilhões.

No Pregel, uma tarefa computacional é expressa por um grafo direcionado. Cada vértice é relacionado a um valor definido e atualizável pelo usuário, e cada aresta direcionada é relacionada a um vértice de origem, o identificador de um vértice de destino e um valor definido pelo usuário. Depois que o grafo de uma determinada tarefa computacional é construído e inicializado, o programa realiza sequências de iterações chamadas *supersteps*. Essas sequências são separadas por pontos de sincronização globais que revelam qual o *superstep* está sendo executado (primeiro, segundo, terceiro, ...). Durante a execução do *superstep*, o Pregel invoca a função definida pelo usuário, para todos os vértices, em paralelo. Assim, todo vértice executa a mesma função, que expressa a lógica de um determinado algoritmo.

As funções especificam o comportamento dos vértices. Cada vértice pode receber mensagens enviadas no *superstep* anterior, enviar mensagens para outros vértices que as receberão no *superstep* posterior, modificar seu estado e suas arestas de saída, e até mesmo modificar a estrutura topológica de todo o grafo. As mensagens são enviadas normalmente pelas arestas de saída, mas podem ser enviadas para qualquer vértice utilizando o seu identificador. As arestas não possuem computações associadas.

A execução do programa termina quando nenhum vértice possui mais tarefas a realizar, mensagens a enviar e, todos já votaram para parar a sua computação em *supersteps* anteriores, tendo entrado, portanto, em um status inativo. A saída do programa Pregel é um conjunto que consiste nos valores de saída de todos os vértices. Assim, a entrada e a saída do programa Pregel são, normalmente, grafos direcionados isomorfos. Mas deve ser considerado que vértices e arestas podem ser adicionadas ou removidas durante a execução do programa.

O Pregel usa *clusters* que consistem em milhares de PCs em *commodity*. O cluster consiste em uma máquina mestre e várias máquinas *workers*. Assim, o grafo é dividido em partições, cada uma consistindo de um conjunto de vértices e todas as arestas de saída desses vértices, e cada partição é atribuída a uma máquina *worker*. A atribuição de um vértice a uma partição é decidida por meio de uma função *hash* ou de funções de atribuição implementadas pelos usuários. O mestre é responsável por coordenar as atividades dos *workers*, atribuir partições e entradas do usuário aos *workers*, instruir cada *worker* a executar um *superstep* e, após o término da computação, instruir os *workers* a

salvar sua parte do grafo. Cada *worker* é responsável por manter o estado de sua parte do grafo, executar as funções do usuário em seus vértices e gerenciar as mensagens.

No Pregel+ (YAN *et al*., 2014) cada *worker* é um processo MPI (*Message Passing Interface*). Ele introduz duas técnicas para reduzir o número de mensagens: espelhamento de vértices e um método de requisição-resposta. O espelhamento é utilizado para resolver o problema de balanceamento de carga de trabalho naqueles vértices que possuem um alto grau de arestas de saída. Para isso, são construídos espelhos de vértices com grande número de arestas de saída em outras máquinas, de modo que as mensagens desses vértices para seus vértices adjacentes sejam transmitidas pelos espelhos localmente na mesma máquina. A API de requisição-resposta permite que um vértice de origem requisite um valor a um vértice de destino. Esse valor é disponibilizado para o vértice de origem na próxima iteração. Todas as requisições de uma máquina para um mesmo vértice de destino são agrupadas em uma única requisição de modo a reduzir o número de mensagens transmitidas entre duas máquinas.

2.4 Giraph

O Giraph (GIRAPH, 2020) é um projeto de código aberto da Apache baseado no modelo Pregel, com características adicionais. Essas características incluem: computações pelo *master*, agregadores de fragmentação e entradas orientadas às arestas.

Uma computação no Giraph executa na fase *Map* da tarefa do Hadoop e os *workers* usam o Zookeper para selecionar o *master* que coordenará a computação. Depois que o grafo é carregado e particionado entre os *workers*, o *master* determina quais *workers* devem iniciar a computação dos *supersteps* consecutivos. Quando a computação termina, os *workers* salvam a saída. As aplicações podem reiniciar automaticamente em caso de falha de *workers*, graças aos pontos de checagem iniciados em intervalos definidos pelo usuário.

O Giraph oferece mecanismos que permitem a implementação de algoritmos em grafos além daqueles predefinidos em biblioteca. Outras contribuições do Giraph incluem: a obtenção dos dados de entrada (vértices e arestas) de qualquer fonte, tais como arquivos de texto de banco de dados NoSQL; a possibilidade das aplicações calcular um valor global a partir dos valores fornecidos por cada vértice, graças aos agregadores, reduzindo-se o tráfego na rede, e; a possibilidade de se armazenar os valores e mensagens no disco, como no cluster Hadoop, para melhorar a escalabilidade do sistema.

2.5 Dryad

O Dryad (ISARD *et al.*, 2010), assim como o Pregel e o Giraph, processam programas de forma paralela utilizando uma estrutura de grafo. Os grafos nesse modelo são direcionados e acíclicos, e as arestas representam os canais de dados e os vértices são os programas. Dessa forma, os programas representados pelos vértices, são executados em *clusters*, e os dados, são transmitidos por meio das arestas. Esses canais de dados podem fazer uso de conexões TCP ou de memória compartilhada.

Os vértices disponibilizados pelo programador da aplicação são gravados, muitas vezes, como programas sequenciais. Sendo assim, o processamento simultâneo, seja em vários nós do *cluster* ou em vários núcleos do processador, fica como responsabilidade dos vértices de agendamento do Dryad.

O Dryad possui um coordenador central, chamado de gerenciador de tarefas. Esse é constituído por códigos usados na construção de um grafo de comunicação de tarefas e, por códigos de biblioteca usados para organizar os recursos disponíveis. Os grafos podem ser alterados em tempo de execução devido à identificação do tamanho e posicionamento dos dados na procura do uso eficiente dos recursos. O gerenciador de tarefas não interfere nas transmissões de dados. Essas são realizadas diretamente entre os vértices.

Os desenvolvedores de aplicação têm a possibilidade de escolher qualquer grafo acíclico direcionado para representar os modos de comunicação de sua aplicação e os mecanismos de transmissão dos dados. Além disso, os vértices suportam qualquer quantidade de dados na entrada e saída, o que pode ser uma vantagem em comparação com o MapReduce, tendo em vista que esse último suporta apenas um conjunto de dados de entrada e um conjunto de dados saída.

O Dryad fornece a linguagem DryadLINQ (FETTERLY *et al.*, 2009) usada para integrar seu ambiente de execução com linguagens SQL-Like.

2.6 All-pairs

O sistema All-pairs (MORETTI *et al.*, 2008) foi projetado para aplicações que utilizam biometria, bioinformática e mineração de dados. Ele se assemelha às implementações de MapReduce e ao Spark po aplicar uma função em um conjunto de dados de entrada e produzir como resultado, um conjunto de dados de saída. Especificamente, ele utiliza uma função para comparar elementos em dois conjuntos de

dados. Assim, o All-pairs pode ser expresso como uma 3-tupla da seguinte maneira: (Conjunto A, Conjunto B, Função F), na qual F é usada para comparar os elementos dos conjuntos A e B e produz como saída uma matriz M que é o produto cartesiano dos dois conjuntos de entrada.

O All-pairs é implementado em quatro etapas. São elas:

1.Modelagem do sistema. Essa etapa visa a construção de um modelo de aproximação a fim de se verificar a quantidade de recurso de CPU necessária e como o particionamento da tarefa pode ser feito.

2.Distribuição dos dados de entrada. Essa etapa visa a construção de uma árvore geradora para transmitir os dados para os nós, o que faz com que cada partição receba os dados de entrada de forma eficaz.

3.Gerenciamento de tarefas em lote. Após o particionamento dos dados entre os nós, as tarefas nas partições são submetidas a um sistema de processamento em lote e um nó é configurado para adquirir os dados.

4.Coleta de resultados. Após o processamento das tarefas, o mecanismo de extração coleta e combina os resultados em um arquivo. Assim, o conteúdo do arquivo é uma lista ordenada dos resultados.

Considerações finais

Neste artigo foram revisados os principais modelos e *frameworks* para realização de processamento de grandes volumes de dados de maneira distribuída. Com o objetivo de mostrar algumas opções que existem atualmene para processamento de Big Data.

Primeiramente foi mostrado o modelo de processamento distribuído proposto pela Google, o MapReduce. Abordando três implementações do mesmo, uma com melhor abordagem para sistemas em lote o Hadoop MapReduce da Apache. Outra abordagem da Apache com foco em sistemas interativos, possuindo uma framework mais completa, sendo este o Spark. E outra implementação focando na utilização melhor dos recursos de uma só máquina, realizando o processamento paralelo em *threads*, chamado Phoenix.

Posteriormente foram abordados outros métodos de processamento distribuído, com o intuito de mostrar as diferenças em relação ao MapReduce. Se situando basicamente na forma que o modelo foi elaborado e seu objetivo, que no caso do processamento de grafos, se encaixa melhor para as redes sociais. Já o All-pairs, foi elaborado para comparações simples entre grandes conjuntos de dados.

Referências

ALMEIDA, D. S.; DUTRA, D. B. Armazenamento de Big Data: uma abordagem didática. **Perspectivas em Ciências Tecnológicas**, v. 8, n. 8, p. 168-194, jun. 2019.

DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. **Communications of the ACM,** v. 51, n. 1, p. 107-113, 2008.

DITTRICH, J.; QUIANÉ-RUIZ, J. A. Efficient big data processing in Hadoop MapReduce. **Proceedings of the VLDB Endowment**, v. 5, n. 12, p. 2014-2015, 2012.

FETTERLY, Y. Y. M. I. D.; BUDIU, M.; ERLINGSSON, Ú.; CURREY, P. K. G. J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. **Proc. LSDS-IR**, v. 8, 2009.

GIRAPH. The Apache Software Foundation, Apache Giraph, http://giraph.apache.org. Acesso em: 22 abr. 2020.

ISARD, M.; BUDIU, M.; YU, Y.; BIRRELL, A.; FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. *In:* ACM SIGOPS/EUROSYS EUROPEAN CONFERENCE ON COMPUTER SYSTEMS, 2., **Proceedings[…]**, 2010. p. 59-72.

MALEWICZ, G.; AUSTERN, M. H.; BIK, A. J.; DEHNERT, J. C.; HORN, I.; LEISER, N.; CZAJKOWSKI, G. (2010, June). Pregel: a system for large-scale graph processing. *In:* ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, **Proceedings[…]**, 2010. p. 135-146.

MORETTI, C.; BULOSAN, J.; THAIN, D.; FLYNN, P. J. All-pairs: An abstraction for data-intensive cloud computing. *In:* INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING, IEEE, 2008. p. 1-11.

RANGER, C.; RAGHURAMAN, R.; PENMETSA, A.; BRADSKI, G.; KOZYRAKIS, C. Evaluating mapreduce for multi-core and multiprocessor systems. *In:* INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE, 13., IEEE, 2007. p. 13-24

ZAHARIA, M.; CHOWDHURY, M.; FRANKLIN, M. J.; SHENKER, S.; STOICA, I. Spark: Cluster computing with working sets. **HotCloud,** v. 10, n. 10-10, p. 95, 2010.

YAN, D.; CHENG, J.; XING, K.; LU, Y.; NG, W.; BU, Y. Pregel algorithms for graph connectivity problems with performance guarantees. **Proceedings of the VLDB Endowment**, v. 7, v. 14, p. 1821-1832, 2014.

YOO, R. M.; ROMANO, A.; KOZYRAKIS, C. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. *In:* INTERNATIONAL SYMPOSIUM ON WORKLOAD CHARACTERIZATION (IISWC). IEEE, 2009. p. 198-207.