FUNDAMENTOS DE BIG DATA

Maycon Viana Bordin

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS

Apache YARN: gerenciador de recursos no Hadoop MapReduce

OBJETIVOS DE APRENDIZAGEM

- Determinar os desafios de gerenciamento de recurso em *big data*.
- Explicar o que é o Apache Yarn.
- Apontar o Apache Yarn como provisor de recursos para o MapReduce.

Introdução

O Hadoop surgiu como uma implementação *open source* do MapReduce e, ao longo dos anos, se tornou a principal plataforma de *big data* do mercado. Com seu crescimento, também começaram a surgir limitações, à medida que os *clusters* assumiam proporções cada vez maiores.

Neste capítulo, você vai estudar os principais desafios encontrados no gerenciamento de recursos de *big data*. Você também vai compreender como esses desafios culminaram na criação do Apache YARN e vai verificar como funciona esse recurso e como ele opera com o MapReduce.

Desafios do gerenciamento de recursos em *big data*

Com o surgimento do **Hadoop**, criado para processar os volumes de dados cada vez maiores devido à escala da internet, também começaram a surgir as dificuldades em gerenciar grandes quantidades de recursos computacionais. No Yahoo!, antes mesmo de o Hadoop surgir, já existiam *clusters* com mais de 800 nós utilizados para o processamento de grafos da *web*.

Os requisitos de escalabilidade começaram a se destacar devido ao crescimento da internet e aos volumes de dados cada vez maiores. Com o tempo, a comunidade do Apache Hadoop foi capaz de escalar a plataforma para *jobs* cada vez maiores, trazendo consigo a necessidade de permitir múltiplos usuários (*multi-tenancy*) executando aplicações no mesmo *cluster* (VAVILAPALLI *et al.*, 2013).

Antes de o Hadoop suportar *multi-tenancy*, muitos usuários subiam um *cluster* do Hadoop, carregavam seus dados no Hadoop Distributed File System (HDFS), realizavam seus processamentos e depois derrubavam todo o sistema. Com o tempo e com a evolução da plataforma, dentro do Yahoo! desenvolveu-se o **Hadoop On Demand** (HoD), que permitia a alocação de *clusters* do Hadoop em um *pool* compartilhado de *hardware*, por meio do uso de uma fila que recebia *jobs* dos usuários com uma descrição das características do *cluster*. Assim que recursos suficientes eram liberados, o *job* submetido pelo usuário era executado com a alocação dos recursos e a criação do *cluster* Hadoop. Como cada execução leva à criação de um *cluster*, isso permitiu que usuários executassem versões diferentes do Hadoop (VAVILAPALLI *et al.*, 2013).

Essa característica do HoD garantia a flexibilidade necessária para que aplicações fossem atualizadas na mesma cadência com que o Hadoop era atualizado. Entretanto, o HoD possuía algumas limitações, que levaram o Yahoo! a aposentá-lo em favor de *clusters* compartilhados do Hadoop. Dentre essas limitações estava a falta de conhecimento sobre a localidade dos dados na hora de alocar recursos para um *job*, o que causava problemas de *performance* quando tarefas precisavam fazer chamadas remotas para obter dados para seu processamento.

Outra limitação do HoD ocorria principalmente com ferramentas de alto nível do Hadoop, como Pig e Hive, que geram um *workflow* de aplicações MapReduce. Nesse cenário, a alocação de recursos era estática, enquanto os *jobs* do *workflow* são dos mais variados. Isso causava subutilização de recursos e, em muitos casos, bloqueio de recursos que não estão sendo utilizados, mas estão alocados para um *job*. Esse fato evidenciou a limitação do HoD em

se adaptar ao uso de recursos de cada *job*. Por fim, a latência dos *jobs* que executavam no HoD era dominada pelo tempo de alocação de recursos, sem contar com o fato de que a estimativa de quantidade de recursos necessários para cada *job* era muito precária, levando à subutilização do *cluster*.

Entretanto, se o HoD possuía suas limitações, a utilização de *clusters* compartilhados do Hadoop também tinha suas limitações e problemas. Por exemplo, a falha do *JobTracker* poderia causar a perda de todos os *jobs* que estavam executando no *cluster* compartilhado, sendo necessário reiniciar e recuperar manualmente os *jobs*. Além disso, o grande número de *jobs* sendo submetidos para um *cluster* compartilhado colocava grande pressão no *JobTracker*, que, além de tudo, era um ponto único de falha no sistema, colocando em risco a confiabilidade e a disponibilidade do Hadoop.

Dentre todas as limitações impostas pelo Hadoop, talvez a maior delas era o fato de ele suportar apenas um modelo de programação, o MapReduce. Essa limitação levou muitos usuários a escreverem aplicações MapReduce que por baixo estavam disparando outros modelos de programação, causando, dessa forma, perda de desempenho e instabilidade no *cluster* (PERWEJ *et al*., 2017).

Finalmente, a utilização de *clusters* compartilhados impôs uma grande limitação com relação a atualizações do Hadoop. Isso porque era necessário parar o *cluster*, realizar as atualizações no *cluster*, realizar as atualizações nas aplicações dos usuários, validá-las e só então retornar o *cluster* ao serviço. Em alguns casos, as atualizações poderiam quebrar aplicações, pois as premissas dos desenvolvedores com relação ao *framework* haviam mudado.

Todas essas dificuldades são inerentes ao gerenciamento de recursos de *big data*, bem como à execução de aplicações que necessitam de grandes quantidades de recursos computacionais. Com a popularização do Hadoop, houve uma concentração de esforços para mitigar esses problemas e encontrar soluções, tanto na academia como na indústria, já que diversas empresas estavam utilizando o Hadoop em larga escala. Esses esforços culminaram na criação do YARN (*Yet Another Resource Negotiator*).

Fique atento

A primeira versão do Hadoop possuía suporte apenas para o modelo de programação conhecido como MapReduce. Na versão 2 do Hadoop, com a criação do YARN, as responsabilidades de gerenciamento de recursos e de aplicação foram divididas, possibilitando o uso de outros modelos de programação como Spark, Tez, Storm, Samza, Impala, Giraph, Accumulo, Flink, Solr e OpenMPI.

Apache YARN

O **Apache YARN** é o sistema de gerenciamento de recursos de *clusters* do Hadoop, introduzido na versão 2 do Hadoop para melhorar a implementação do MapReduce, bem como tornar a plataforma genérica o suficiente para dar suporte a outros modelos de programação distribuída (WHITE, 2015). O YARN fornece interfaces de programação de aplicações (APIs, do inglês *application programming interface*) que permitem a requisição e a utilização de recursos de *cluster*. Essas APIs não são expostas diretamente para aplicações de usuários, mas para os *frameworks* de computação distribuída que utilizam essas APIs para conversar com o YARN, escondendo todos os detalhes de gerenciamento de recursos do usuário final.

A Figura 1 mostra um exemplo de *frameworks* de processamento distribuído que executam em cima do YARN, como o MapReduce, o Spark e o Tez. Além das camadas apresentadas na Figura 1, existem ainda as camadas de aplicações construídas sob os *frameworks* de processamento distribuído e que não conversam diretamente com o YARN, como é o caso do Hive, do Pig e do Crunch.

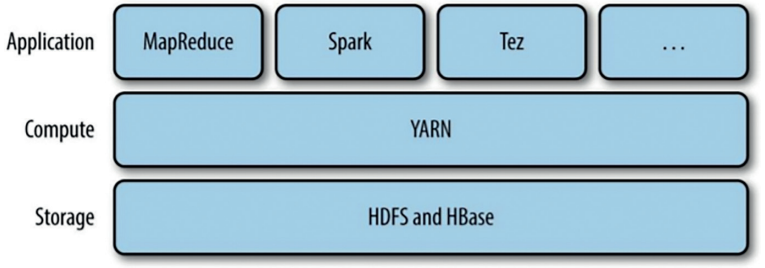

Figura 1. Aplicações no YARN.
Fonte: WHITE (2015).

A arquitetura do YARN é composta por um *Resource Manager* (um por *cluster*) responsável pelo gerenciamento dos recursos de todo o *cluster* e pelo monitoramento dos nós que fazem parte do *cluster*, os *Node Managers*. Estes executam em cada um dos nós do *cluster* e são responsáveis pela criação e pelo monitoramento de *containers* de execução (KULKARNI; KHANDEWAL, 2014).

O *Resource Manager* é o alocador central de recursos e utiliza uma descrição abstrata das necessidades de seus inquilinos, que se mantêm, dessa forma, ignorantes da semântica de cada alocação de recurso. Essa responsabi-

lidade agora é delegada ao *Application Master*, que coordena o plano lógico de um único *job*, requisitando recursos ao *Resource Manager*, gerando um plano físico de execução por meio dos recursos que ele recebeu e coordenando a execução do plano criado (VAVILAPALLI *et al.*, 2013; LIN *et al.*, 2016).

A Figura 2 mostra a arquitetura do YARN, com seus componentes em cor azul, enquanto os componentes das aplicações estão nas cores amarela e rosa. Como componente principal do YARN, o *Resource Manager* executa como um *daemon* em uma máquina dedicada e age como a autoridade central que decide a alocação de recursos para diversas aplicações rivais dentro de um *cluster*. Esse componente tem uma visão global dos recursos, o que lhe permite tomar decisões com relação à justiça, à capacidade e à localidade dos recursos (com relação ao local de armazenamento dos dados no HDFS, por exemplo) (ELSHATER *et al.*, 2015).

Dependendo das demandas da aplicação, das prioridades de escalonamento e da disponibilidade de recursos, o *Resource Manager* dinamicamente aloca *containers* para que a aplicação execute em nós específicos. O *container* é um conjunto lógico de recursos (CPU e memória RAM) atrelados a um nó específico do *cluster*. Para garantir o cumprimento dessas alocações, bem como monitorá-las, o *Resource Manager* interage com o *Node Manager*. O *Node Manager* é responsável pelo monitoramento dos recursos disponíveis no nó, por relatar falhas e gerenciar o ciclo de vida dos *containers* (inicializar e destruir). A visão global do *cluster* é construída por *snapshots* das visões dos *Node Managers* enviadas para o *Resource Manager* por meio da comunicação via *heartbeats* (VAVILAPALLI *et al.*, 2013).

A submissão de aplicações para o YARN é feita por meio de um protocolo público e passa por uma etapa de controle de admissão, que valida credenciais de segurança, além de várias outras validações administrativas. As aplicações aceitas são enviadas para o escalonador para execução. Assim que o escalonador conseguir recursos suficientes para a aplicação, esta passa do estado aceito para o estado em execução. Esse processo envolve a alocação de um *container* para o *Application Master* e a execução dele em um nó do *cluster*.

O *Application Master* é a cabeça de uma aplicação, ele gerencia todo o ciclo de vida da aplicação, desde o aumento ou a redução dos recursos utilizados, o gerenciamento do fluxo de execução, lidando com falhas e atrasos computacionais, além de realizar outras otimizações locais. Além disso, fica a cargo do *Application Master* gerenciar os recursos disponíveis para concluir um *job*, sendo possível a ele requisitar mais recursos ao *Resource Manager*. O YARN ainda fornece protocolos específicos que permitem a comunicação diretamente entre o *Application Master* e os *containers*.

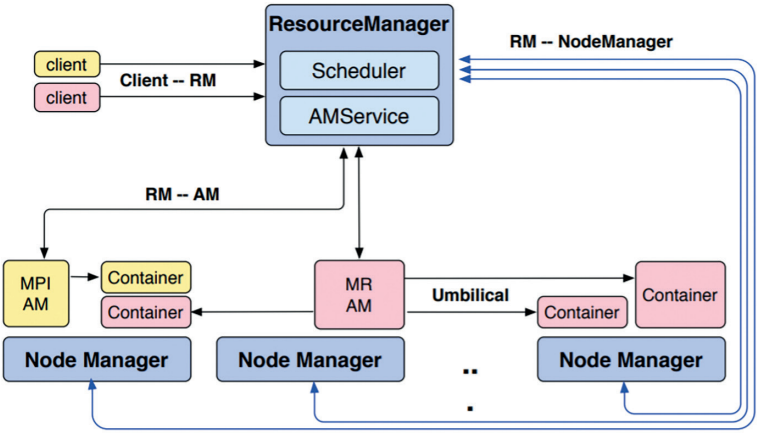

Figura 2. Arquitetura do YARN.
Fonte: VAVILAPALLI *et al.* (2013)

O modelo de requisição de recursos definido pelo YARN é bem flexível, permitindo a requisição de um conjunto de *containers* a qualquer momento, sendo cada *container* com suas quantidades de recursos necessários, bem como restrições de localidade com base na localização de blocos do HDFS. Cada tipo de aplicação pode ter uma estratégia diferente para a alocação de recursos. O Spark, por exemplo, aloca todos os recursos necessários para sua execução no início, enquanto o MapReduce inicia alocando apenas os recursos da fase de `map`, e os recursos da fase de `reduce` são alocados posteriormente (WHITE, 2015).

No YARN, aplicações podem ser classificadas de acordo com o número de aplicações por *jobs* dos usuários. No caso do MapReduce, cada *job* do usuário é uma aplicação. Já no segundo modelo, uma aplicação é utilizada para executar um *workflow* completo de *jobs*, como é o caso do Spark, em que vários *jobs* reutilizam os mesmos recursos, permitindo otimizações como o cache de dados em memória e a reutilização de *containers*. O terceiro modelo é o de uma aplicação de longa duração que é compartilhada por diferentes usuários, fazendo da aplicação uma espécie de coordenador, como é o caso do Apache Slider e do Cloudera Impala (WHITE, 2015).

A introdução do Apache YARN permitiu o desenvolvimento de diversos *frameworks* de processamento distribuído além do MapReduce. Para aplicações que se assemelham a DAGs (*directly acyclic graphs*), existem o Spark

e o Tez; para processamento em tempo real, há o Spark, o Samza e o Storm. Existem ainda *frameworks* que facilitam a construção de aplicações em cima do YARN, como o Apache Slider e o Apache Twill.

Saiba mais

Os problemas identificados ao longo dos anos no Hadoop que culminaram na criação do Apache YARN também levaram outras empresas a criarem ferramentas para melhorar o gerenciamento de recursos de *clusters*. Dentre elas, as mais conhecidas são: Apache Mesos, Omega e Corona.

O Apache YARN e o MapReduce

Na versão 1 do Hadoop, o MapReduce era o único paradigma de programação suportado. Além disso, o processo de execução dos *jobs* era controlado por dois componentes: o *JobTracker* e o *TaskTracker*. O *JobTracker* era responsável por coordenar a execução de todos os *jobs* do *cluster* e por escalonar tarefas dos *jobs* para executar nos *TaskTrackers*. Os *TaskTrackers* eram os responsáveis por executar tarefas e enviar relatórios de progresso para o *JobTracker*, e este mantinha um registro do progresso geral de cada *job*. Se uma tarefa falhasse, o *JobTracker* poderia escalonar ela para reexecutar em outro *TaskTracker* (WHITE, 2015).

No MapReduce 1, o *JobTracker* era responsável tanto pelo escalonamento de tarefas nos nós quanto pelo monitoramento do progresso das tarefas (controle das tarefas em execução, reexecução de tarefas que falharam ou que estão demorando demais). O *JobTracker* era ainda responsável por manter um histórico de todos os *jobs* executados. Já no YARN, essas responsabilidades são gerenciadas separadamente pelo *Resource Manager* e pelo *Application Master* (um para cada *job* do MapReduce). Inclusive, o histórico de *jobs* executados é um componente separado no YARN, sendo gerenciado pelo *TimelineServer*.

No YARN, o *Resource Manager* é genérico e agnóstico a qualquer tipo de modelo de programação, enquanto o *Application Master* é específico de cada *framework* de processamento distribuído. No caso do MapReduce, ele possui uma implementação específica do *Application Master*, com toda a lógica necessária para gerenciar uma aplicação de MapReduce. O Quadro 1 compara os componentes do Map Reduce versão 1 e do YARN.

Quadro 1. Comparação entre os componentes do MapReduce 1 e do YARN

MapReduce 1	YARN
JobTracker	*Resource Manager*, *Application Master*, *Timeline Server*
TaskTracker	*NodeManager*
Slot	*Container*

Fonte: WHITE (2015)

A introdução do YARN trouxe diversos benefícios a todo o ecossistema Hadoop e mais especificamente para aplicações MapReduce. Antes do YARN, aplicações MapReduce começavam a sofrer de gargalos de escalabilidade quando chegavam próximo a 4 mil nós e 40 mil tarefas, devido ao fato de o *JobTracker* precisar gerenciar tanto os *jobs* como as tarefas dos *jobs*. Já com o YARN, esse problema é superado, devido à sua arquitetura que separa o gerenciamento de recursos (*Resource Manager*) do gerenciamento de tarefas (*Application Master*), permitindo a escalabilidade para mais de 10 mil nós e 100 mil tarefas (WHITE, 2015). Essa escalabilidade é possível porque agora cada *job* possui um *master* dedicado para gerenciar as tarefas do MapReduce, ao contrário do *JobTracker*, que ficava sobrecarregado devido à necessidade de gerenciar tarefas de todos os *jobs* do *cluster*.

Outro aspecto que melhorou com a introdução do YARN foi a alta disponibilidade do *cluster*, que antes era muito frágil. Isso porque era muito difícil replicar o estado do *JobTracker*, dado que este estava em constante mudança (com o estado de milhares de tarefas sendo alterado a cada segundo). A dificuldade em replicar o estado do *JobTracker* tornava difícil a recuperação de falhas, que consiste em subir um novo *JobTracker* com o estado replicado. Com a separação das responsabilidades do *JobTracker* no YARN, garantir a alta disponibilidade se tornou uma tarefa viável. No YARN, o *Resource Manager* é altamente disponível, bem como o *Application Master* de cada uma das aplicações executando no YARN.

No MapReduce 1, cada *TaskTracker* é configurado com uma alocação estática de *slots*, estes divididos em *slots* de `map` e *slots* de `reduce` em tempo de configuração. Os *slots* de `map` só podem ser utilizados para executar tarefas de `map`, e os *slots* de `reduce` só podem ser utilizados por tarefas de `reduce`. Já no YARN, o *Node Manager* gerencia um *pool* de recursos, em vez de um número fixo de *slots*. Dessa forma, não existem limitações de *slots* de `map` e de `reduce`, sendo que, se houverem recursos disponíveis, eles poderão ser alocados para a aplicação, independentemente do tipo de tarefa. Além disso, no YARN, os recursos são mais granulares, permitindo que aplicações façam

requisições apenas para os recursos que são necessários, em vez de *slots* indivisíveis, que podem ser muito grandes ou muito pequenos (WHITE, 2015).

Segundo White (2015), um dos maiores benefícios do YARN é que ele abriu o Hadoop para outros tipos de aplicações distribuídas além do MapReduce. Com o YARN, é possível até que usuários executem diferentes versões do MapReduce no mesmo *cluster*, tornando o processo de atualização do MapReduce mais gerenciável.

A Figura 3 mostra como funciona a execução de uma aplicação MapReduce dentro do Hadoop 2 com o YARN. O processo se inicia com a submissão e a validação da aplicação para o *Resource Manager*, a computação dos *input splits* para o *job*, e a cópia dos recursos necessários para executar o *job*. O *Resource Manager* aloca um *container* para a execução do *Application Master*, que cria diversos objetos para controlar o progresso do *job*, além de obter o número de *splits* de entrada, criar as tarefas para as operações de `map` para cada *split*, bem como criar as tarefas de `reduce` conforme definido nas configurações do *job* (WHITE, 2015).

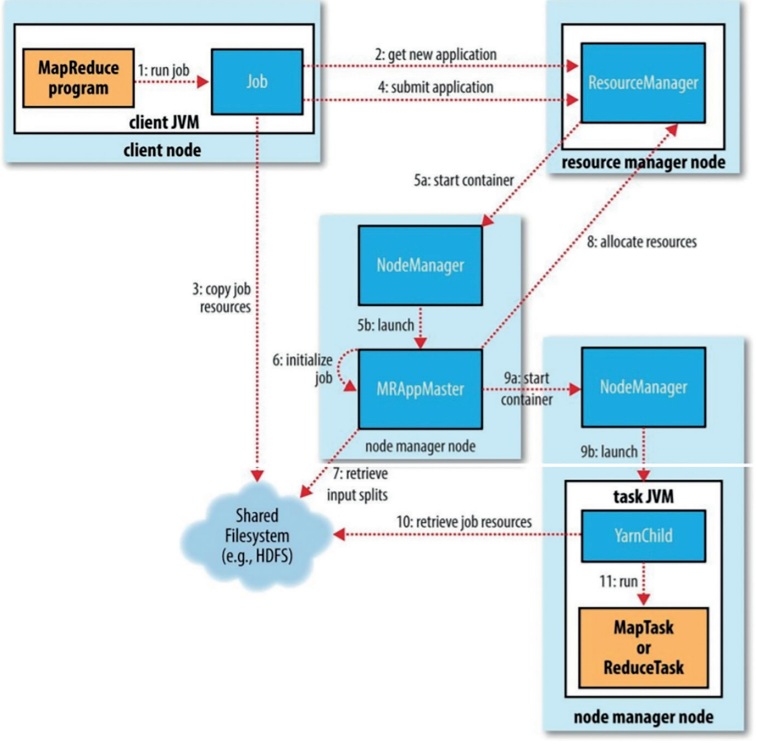

Figura 3. Como o Hadoop executa uma aplicação MapReduce.
Fonte: White (2015).

As requisições para as tarefas de `map` são feitas antes e com prioridade maior do que as tarefas de `reduce`, porque todas as tarefas de `map` precisam finalizar antes que a fase de ordenação do `reduce` comece. As requisições para os recursos das tarefas de `reduce` só serão feitas quando 5% das tarefas de `map` estiverem faltando para serem completadas.

Ao contrário das tarefas de `map`, as tarefas de `reduce` podem executar em qualquer parte do *cluster*. Já as tarefas de `map` possuem restrições de localidade de dados, que o escalonador tenta atender. Idealmente, todas as tarefas de `map` têm leitura de dados apenas locais, garantindo um melhor desempenho. Caso alguma tarefa não consiga fazer a leitura local dos dados, é possível dar prioridade para que a tarefa seja alocada ao menos no mesmo *rack* de computadores. No pior dos casos, a tarefa será alocada em uma máquina em outro *rack* (WHITE, 2015). As requisições de recursos para uma aplicação MapReduce também especificam o tamanho de memória e CPU que deverá ser alocado para as tarefas.

Quando uma tarefa é alocada em um *container* em um nó específico, o *Application Master* inicializa o *container* ao contatar o *Node Manager*. A tarefa é executada por uma aplicação Java cuja classe principal é o `YarnChild`. Antes que a tarefa inicie a execução, ela localiza os recursos que a tarefa precisa, como as configurações e os arquivos JAR (Java ARchive), além de qualquer arquivo do *cache* distribuído que ele possa precisar.

O `YarnChild` executa em uma máquina virtual dedicada, evitando, dessa forma, que erros no código das tarefas de `map` ou `reduce` afetem o *Node Manager*. Além disso, cada tarefa pode executar ações de instalação e de *commit*, que executam na mesma máquina virtual Java em que a tarefa está executando e são determinadas pelo *OutputCommitter* do *job*. Ações de *commit* de *jobs* baseados em arquivos incluem mover arquivos temporários para suas localizações finais.

Tanto as tarefas de `map` como as de `reduce` se comunicam de tempos em tempos com o *Application Master*, enviando relatórios de progresso da tarefa, garantindo, dessa forma, uma visão completa do progresso geral do *job*. Quando a última tarefa em execução é finalizada, o *Application Master* altera seu *status* para bem-sucedido. Quando a aplicação cliente, que fica verificando de tempos em tempos o *status* da aplicação, verifica que ela finalizou, o processo pode então finalizar. O *Application Master* e os *containers* das tarefas limpam seus estados, e os *OutputCommitters* comitam o *job*. Por fim, o histórico do *job* é arquivado, para poder ser acessado no futuro pelo *JobHistory server*.

Referências

ELSHATER, Y. *et al*. A study of data locality in YARN. *In*: INTERNATIONAL CONGRESS ON BIG DATA, 2015, New York. *Anais* [...]. New York: IEEE, 2015. p. 174-181.

KULKARNI, A. P.; KHANDEWAL, M. Survey on Hadoop and Introduction to YARN. *International Journal of Emerging Technology and Advanced Engineering*, [S.l], v. 4, n. 5, p. 82-87, May 2014.

LIN, J. C. *et al*. ABS-YARN: A formal framework for modeling Hadoop YARN clusters. *In*: INTERNATIONAL CONFERENCE ON FUNDAMENTAL APPROACHES TO SOFTWARE ENGINEERING, 19., 2016, Eindhoven, The Netherlands. *Anais* [...]. Berlin: Springer, 2016. p. 49-65.

PERWEJ, Y. *et al*. An empirical exploration of the Yarn in big data. *International Journal of Applied Information Systems*, New York, v. 12, n. 9, p. 19-29, Dec. 2017.

VAVILAPALLI, V. K. *et al*. Apache hadoop Yarn: yet another resource negotiator. *In*: ANNUAL SYMPOSIUM ON CLOUD COMPUTING, 4., 2013, Santa Clara, CA. Proceedings of the 4th Annual Symposium on Cloud Computing, SoCC 2013. New York: Association for Computing Machinery, 2013. p. 1-16.

WHITE, T. *Hadoop*: the definitive guide, 4th ed. Sebastopol, CA: O'Reilly, 2015.

Leituras recomendadas

FASALE, A.; KUMAR, N. *Yarn essentials*. Birmingham: Packt Publishing, 2015.

MATHIYA, B. J.; DESAI, V. L. Apache hadoop Yarn parameter configuration challenges and optimization. *In*: INTERNATIONAL CONFERENCE ON SOFT-COMPUTING AND NETWORKS SECURITY, 1., 2015, Coimbatore, India. *Anais* [...]. New York: IEEE, 2015. p. 1-6.

POP, F.; KOŁODZIEJ, J.; DI MARTINO, B. *Resource management for big data platforms*. Singapore: Springer, 2016.

VERSACI, F.; PIREDDU, L.; ZANETTI, G. Scalable genomics: from raw data to aligned reads on Apache YARN. *In*: IEEE INTERNATIONAL CONFERENCE ON BIG DATA, 2016, Washington, D.C. *Anais* [...]. New York: IEEE, 2016. p. 1232-1241.

Fique atento

Os *links* para *sites* da *web* fornecidos neste capítulo foram todos testados, e seu funcionamento foi comprovado no momento da publicação do material. No entanto, a rede é extremamente dinâmica; suas páginas estão constantemente mudando de local e conteúdo. Assim, os editores declaram não ter qualquer responsabilidade sobre qualidade, precisão ou integralidade das informações referidas em tais *links*.

Conteúdo:

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS