ESCALONAMENTO BASEADO EM LOCALIDADE NO AMBIENTE WATERSHED

BRUNO CERQUEIRA HOTT

ESCALONAMENTO BASEADO EM LOCALIDADE NO AMBIENTE WATERSHED

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: DORGIVAL OLAVO GUEDES NETO

Belo Horizonte, MG

Julho de 2016

Hott, Bruno Cerqueira

H834e Escalonamento baseado em localidade no ambiente Watershed / Bruno Cerqueira Hott. — Belo Horizonte, MG, 2016
xxii, 43 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais
Orientador: Dorgival Olavo Guedes Neto

1. Computação – Teses. 2. Big data. 3. Sistemas distribuídos. I. Orientador. II. Título.

CDU 519.6*31(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Escalonamento baseado em localidade no ambiente watershed

BRUNO CERQUEIRA HOTT

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. DORGIVAL OLAVO GUEDES NETO - Orientador
Departamento de Ciência da Computação - UFMG

PROF. ÍTALO FERNANDO SCOTÁ CUNHA
Departamento de Ciência da Computação - UFMG

PROF. RENATO ANTÔNIO CELSO FERREIRA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 15 de julho de 2016.

Dedico este trabalho à minha mãe, Sandra, fonte de força e inspiração.

Agradecimentos

Agradeço à UFMG e aos professores do DCC pela oportunidade oferecida e pelos ensinamentos passados. Agradeço especialmente ao professor Dorgival Guedes pela paciência, tempo dedicado e por toda ajuda para que este trabalho fosse realizado. E aos demais professores que direta ou indiretamente fizeram parte do processo.

Aos amigos que fiz durante essa jornada e que foram imprencindíveis para o meu crescimento, em especial: Bruno dos Santos e Rodrigo Rocha. Agradeço também aos amigos de longa data que sempre me apoiaram, em especial: Victor, Fernando e Artur.

À toda minha família que me incentivou a buscar um futuro melhor e aos meus pais por sonharem e batalharem por meu futuro. E à minha filha, Marcela, por ser meu maior incentivo.

"Um dia... Pronto!... Me acabo.
Pois seja o que tem de ser.
Morrer: Que me importa?
O diabo é deixar de viver."
(Mário Quintana)

Resumo

O aumento da conectividade e da banda na Internet, combinados com a redução do custo de equipamentos eletrônicos em geral, têm causado uma explosão do volume de dados que trafegam pela rede. Ao mesmo tempo, recursos para armazenar esses dados vêm crescendo, o que levou ao surgimento de sistemas especialmente desenvolvidos para processá-los, tendo como um exemplo inicial o modelo MapReduce da Google, que foi seguido por diversas implementações de código aberto, como Hadoop, e novos modelos, como Spark. Além disso, tornou-se necessário uma solução para o armazenamento desse enorme conjunto de dados e sistemas de arquivos distribuídos, como HDFS e Tachyon, foram surgindo. Como os dados agora representam um volume muito grande e estão distribuídos por diversas máquinas em um cluster, surge o problema de levar as aplicações para perto das bases de dados de forma eficaz. Caso isso não seja feito, o preço de mover os dados pelo sistema pode ser muito alto e prejudicar o desempenho final da aplicação.

Dependendo da localização, o acesso aos dados pela aplicação pode ser realizado diretamente no disco da máquina local, pela memória local (via caching), ou a partir da memória de outra máquina do cluster (via rede). Os diversos compromissos em termos de capacidade de armazenamento, tempo de acesso e custo computacional envolvidos tornam não trivial uma decisão de posicionamento.

Este trabalho implementa e analisa o escalonamento baseado em localidade de dados no ambiente de processamento Watershed. Para essa análise foi feita uma integração do Watershed ao ecossistema Hadoop, criando-se canais de comunicação com os sistemas de arquivos distribuídos HDFS e Tachyon. Com base nas informações de localidade providas por esses sistemas, implementamos um escalonador de processo baseado em localidade para aplicações Watershed sobre aqueles sistemas de arquivos. Por fim, experimentos foram realizados com o intuito de comparar os diversos meios de manipulação de arquivos, seja pelo sistema de arquivos local, distribuído ou em memória. Os resultados obtidos comprovam as vantagens de se levar em conta o posicionamento dos dados no escalonamento de aplicações desse tipo.

Palavras-chave: Sistemas Distribuídos, Big data, Localidade de dados .

Abstract

Increased in connectivity and bandwidth on the Internet, combined with the reduced cost of electronic equipment in general have caused an explosion in the volume of data traveling over the network. At the same time, resources to store these data have been growing, which led to the appearance of specially developed systems to process them, and as an early example the MapReduce model of Google, which was followed by several open source implementations such as Hadoop, and new models such as Spark. In addition, it was necessary a solution to the storage of this huge data set and distributed file systems like HDFS and Tachyon, were emerging. Because the data are now a very large volume and are distributed over multiple machines in a cluster, the problem arises of getting applications close to the databases in a effectively way. If this is not done, the price of moving the data through the system can be very high and impair the final performance of the application.

Depending on location, the data access application may be performed directly on the disk of the local machine, the local memory via caching of memory or from another cluster machine via network. The various commitments in terms of storage capacity, access time and computational cost involved make nontrivial a positioning decision.

This work implements and analyzes the scheduling based on data locality in the Watershed processing environment. For this analysis was made an integration of Watershed to Hadoop ecosystem, creating channels of communication with the HDFS distributed file systems and Tachyon. Based on the location information provided by these systems, we have implemented a process scheduler based on locality for Watershed applications on those file systems. Finally, experiments were conducted in order to compare the various means of manipulating files, either by the local file system, distributed or in memory. The results show the advantages of taking into account the placement of data in scheduling such applications.

Keywords: Distributed Systems, Big data, Data locality .

Lista de Figuras

Lista de Algoritmos

Sumário

Capítulo 1

Introdução

O aumento da conectividade e da banda na Internet, combinados com a redução do custo de equipamentos eletrônicos em geral, têm causado uma explosão do volume de dados que trafegam pela rede. Ao mesmo tempo, recursos para armazenar esses dados vêm crescendo, o que levou ao surgimento de sistemas especialmente desenvolvidos para processá-los, tendo como um exemplo inicial o modelo MapReduce da Google [Dean & Ghemawat, 2008], que foi seguido por diversas implementações de código aberto, como Hadoop [White, 2009], e novos modelos, como Spark [Zaharia et al., 2010b]. Além disso, tornou-se necessária uma solução para o armazenamento desse enorme conjunto de dados e sistemas de arquivos distribuídos, como HDFS [Shvachko et al., 2010] e Tachyon [Li et al., 2014], foram surgindo. Como os dados agora representam um volume muito grande e estão distribuídos por diversas máquinas em um cluster, surge o problema de levar as aplicações para perto das bases de dados de forma eficaz. Caso isso não seja feito, o preço de mover os dados pelo sistema pode ser muito alto e prejudicar o desempenho final da aplicação.

Apesar desse problema ser reconhecido, ainda existe pouco entendimento sobre a interferência da localidade dos dados no desempenho desses frameworks. Dependendo da localização, o acesso aos dados pela aplicação pode ser realizado diretamente no disco da máquina local; pela memória local, via caching; ou a partir da memória de outra máquina do cluster, via rede. Os diversos compromissos em termos de capacidade de armazenamento, tempo de acesso e custo computacional envolvidos tornam não trivial uma decisão de posicionamento. Por esses motivos, muitas vezes esses fatores podem ser melhor avaliados em tempo de execução da aplicação, quando se pode encontrar um melhor aproveitamento dos recursos do ambiente ou gastos com hardware.

O ambiente de processamento Watershed [Ramos et al., 2011] é um sistema desenvolvido no Departamendo de Ciência da Computação da UFMG que implementa

o modelo filtro-fluxo para processamento distribuído. Nesse modelo, a informação é transmitida pelas máquinas por meio de fluxos de dados e são processados pelos filtros. O filtro é um elemento de processamento que contém um conjunto de portas de entrada e um conjunto de portas de saída, onde pode-se adicionar um canal de comunicação a cada uma dessas portas. Quando um filtro deve consumir os dados produzidos por outro filtro, um canal de comunicação vincula-os a partir de suas portas de entrada e saída, respectivamente. Damos a esses canais de comunicação entre filtros nome de fluxos de dados. Além de fazer a conexão entre dois filtros, os fluxos de dados podem entregar os dados da Internet para serem processados ou armazenar a saída da aplicação no disco, por exemplo. Na prática, cada filtro pode ter várias instâcias que processam dados em paralelo, o que gera o processamento distribuído do modelo.

Em nossa experiência anterior com o ambiente de processamento Watershed e com seu antecessor, o Anthill [Ferreira et al., 2005], observamos sempre um impacto ao se dividir um fluxo de processamento de forma que o processamento inicial de dados lidos do disco fosse executado em um nó diferente daquele de onde os dados foram lidos. Em versões anteriores desses ambientes, o posicionamento dos dados e do processamento era deixado como tarefa do programador da aplicação. Com base nessa observação, decidimos implementar uma abstração de acesso a arquivos de forma transparente para o programador e um escalonador sensível à localidade de dados, garantindo que a primeira tarefa de processamento definida para qualquer dado fosse, sempre que possível, executada em uma máquina onde aqueles dados estivessem disponíveis localmente, seja em disco ou em memória. E com isso, aumentar a eficácia do acesso aos dados, assim como minimizar o volume de dados transferido pela rede.

Temos então o problema de realizar o máximo de leitura local possível. Para isso, devemos ser capazes de saber onde os arquivos de entrada estão armazenados e, de posse dessa informação, executar a tarefa que utilizará esses arquivos na máquina onde eles se encontram. Na maioria dos sistemas de arquivos distribuídos, os arquivos estão divididos e espalhados pelas máquinas do cluster. Por esse motivo, também precisamos dividir a tarefa de leitura em diversas instâncias, onde cada uma delas acessará uma porção dos dados.

Este trabalho avalia os aspectos de localidade no ambiente Watershed durante o desenvolvimento do escalonador. Com base na descrição da tarefa de processamento, que indica as etapas de processamento a serem executadas e o(s) arquivo(s) de entrada a ser(em) utilizado(s), das informações providas pelo sistema de arquivos sobre a localização dos dados e dos recursos oferecidos pelo gerente de recursos do cluster, nosso escalonador, sempre que possível, executa tarefas em máquinas onde aquele dado esteja acessível, seja em memória ou disco. Realizamos diversos experimentos com o

intuito de comparar os resultados com outros escalonamentos possíveis. Os resultados obtidos comprovam as vantagens de se levar em conta o posicionamento dos dados no escalonamento de aplicações desse tipo.

1.1 Objetivos

Essa dissertação tem como principal objetivo integrar o ambiente Watershed ao ecossistema Hadoop, com destaque ao seu sistema de arquivos distribuídos, HDFS. Um segundo objetivo é realizar um estudo sobre os aspectos de localidade de dados no ambiente de processamento de dados massivos Watershed. Além disso, integrá-lo ao Tachyon, um sistema de arquivos em memória.

1.2 Contribuições

Durante o trabalho de re-engenharia do ambiente Watershed, agora chamado de Watershed-ng [Rocha et al., 2016], realizamos a integração da nova implementação com o ecossistema Hadoop, utilizando o gerente de recursos YARN [Vavilapalli et al., 2013] e o sistema de arquivos HDFS.

Continuando aquele trabalho e ampliando-o no que diz respeito ao escalonamento com base em localidade de tarefas, este trabalho apresenta as seguintes contribuições:

- Implementação dos módulos de leitura e escrita do HDFS para o Watershed-ng.
- Implementação dos módulos de leitura e escrita do Tachyon para o Watershed-ng.
- Implementação do módulo escalonador baseado em localidade integrado para o Watershed-ng ao YARN.
- Avaliação de desempenho que demonstra o impacto da localidade para aplicações naquele ambiente.

Um artigo com uma versão inicial deste trabalho foi aceito para publicação e apresentado durante o Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos de 2016 [Hott et al., 2016]. Além disso, a parte inicial da implementação da integração com o HDFS foi também descrita em um artigo publicado no periódico *Concurrency: Practice and Experience* [Rocha et al., 2016].

1.3 Organização

Com isso em mente, o restante deste trabalho está organizado da seguinte forma: a seção 2 fornece mais detalhes sobre os elementos do ecossistema Hadoop utilizados (HDFS, YARN e Tachyon), assim como discute trabalhos relacionados. A seção 3 descreve a implementação do nosso escalonador, cujos resultados de avaliação são apresentados na seção 4. Finalmente, a seção 5 conclui com algumas observações finais e discussão de trabalhos futuros.

Capítulo 2

Contextualização

Para o desenvolvimento desta dissertação é importante compreender bem o funcionamento do Watershed-ng, dos sistemas de arquivos HDFS e Tachyon, e do escalonador YARN. Esses elementos são discutidos a seguir, bem como os principais trabalhos relacionados.

2.1 Hadoop YARN

Idealmente, aplicações distribuídas decidem dinamicamente em qual nó de computação cada tarefa será executada, monitorando o estado de cada tarefa durante a execução. Monitorar o estado das tarefas possibilita, por exemplo, saber se as tarefas foram concluídas, bem como tomar medidas de tolerância à falhas sempre que necessário. Para ser capaz de realizar um escalonamento inteligente dessas tarefas distribuídas, considerando maior nível de detalhes do ambiente de execução, é necessário também gerenciar os recursos disponíveis oferecidos pelos nós de computação.

O YARN [Vavilapalli et al., 2013], escalonador de recursos do ecossistema Hadoop, oferece o serviço de uma plataforma básica para a gestão das aplicações distribuídas em um nível mais alto de abstração, enquanto a semântica específica da aplicação é gerenciada por cada um dos *frameworks* que venham a utilizá-lo. Permite, assim, que diferentes aplicações distribuídas possam coexistir em um mesmo ambiente de execução em *cluster*.

Especificamente, tem-se um *ResourceManager* (RM) que é responsável por gerenciar o uso de recursos de um determinado cluster. É ele quem atribui porções dos recursos para todas as aplicações do sistema. Fica a cargo do *ApplicationMaster* (AM), que é o módulo desenvolvido especificamente por cada aplicação, coordenar o plano lógico da própria aplicação requisitando recursos ao RM, submetendo e coordenando a

execução da tarefa. Também, em cada máquina do cluster são executados os *NodeManagers* (NMs), que são responsáveis por executar as tarefas da aplicação nos recursos disponibilizados pelo RM.

Mais especificamente, o *Resource Manager* executa como um *daemon* em uma máquina dedicada, agindo como autoridade central, atribuindo recursos entre as várias aplicações concorrentes do cluster. Dependendo da demanda da aplicação, prioridades de escalonamento e disponibilidade de recursos, o RM aloca dinamicamente *contêineres* para as tarefas executarem em determinados nós. Contêineres são pacotes lógicos de recursos (ex.: $<$ 2 GB RAM, 1 CPU$>$) amarrado a um nó em particular.

As aplicações do cluster fazem suas requisições de recursos ao RM. Uma vez que este possua recursos suficientes, contêineres são alocados para a aplicação, que serão executados em diferentes nós do cluster.

O escalonador acompanha, atualiza e satisfaz as requisições com recursos disponíveis, como anunciado nos *heartbeats* do *NodeManager*. Porém, ele não realiza nenhum monitoramento ou rastreamento para a aplicação, sem garantias de reiniciar tarefas que não são adequadamente finalizadas devido a qualquer falha da própria aplicação ou falhas de hardware. Ao invés disso, o *ResourceManager* repassa o status de saída de cada contêiner que termina, como reportado pelo NM, para o AM responsável. AMs também são notificados quando um novo NM se junta ao cluster para que ele possa requisitar recursos nos novos nós.

Cada aplicação tem um *ApplicationMaster*, que é quem gerencia todos os aspectos do ciclo de vida da aplicação, requisita recursos ao ResourceManager, aumenta e diminui o consumo de recursos dinamicamente, gerencia o fluxo de execução, trata falhas e realiza outras otimizações locais. Porém, ele executa no cluster como qualquer outro contêiner, podendo executar qualquer código de usuário e ser escrito em qualquer linguagem de programação, uma vez que toda comunicação entre RM e NM seja codificada utilizando os protocolos de comunicação esperados.

Tipicamente, um AM vai necessitar de recursos (CPUs, RAM, disco, etc) disponíveis em múltiplos nós para completar a aplicação. Depois de construir um modelo dos seus requisitos, o AM codifica suas preferências e envia para o RM. Nessas requisições são incluídas especificações de preferências de localidade e propriedades dos contêineres. O RM fará então um esforço para satisfazer as requisições de recursos vindas de cada aplicação de acordo com as políticas de disponibilidade e escalonamento. Quando o AM recebe um contêiner disponível para seu uso, ele codifica uma requisição de execução de aplicação no contêiner.

O AM codifica sua necessidade por recursos em termos de um ou mais *ResourceRequests*, sendo que cada um contém:

1. número de contêineres (ex.: 200 contêineres),
2. recursos dos contêineres (ex.: 2 CPUs, 4GB RAM),
3. preferências de localidade (ex.: máquina X_1 ou rack X) e
4. prioridade da requisição para a aplicação.

Se necessário, contêineres em execução podem comunicar diretamente com o AM através de um protocolo específico para reportar seu status e receber comandos específicos do framework.

Por fim, temos os *NodeManagers*(NMs) que são *daemons* "trabalhadores" do YARN. Os NMs são responsáveis por monitorar a disponibilidade de recursos na sua máquina, reportar falhas e gerenciar o ciclo de vida dos contêineres locais. Eles, então, disponibilizam uma gama de serviços aos seus contêineres, sendo alguns deles: inicialização, encerramento, autenticação, gerência de dependências, monitoramento da execução, entre outros.

Todos os contêineres no YARN, incluindo AMs, são descritos por um *container launch context* (CLC). Esse registro inclui um mapa das variáveis de ambiente, dependencias, chaves de segurança e o comando necessário para criar o processo. Para executar, o NM configura o ambiente para o contêiner e copia todas as dependências necessárias para o armazenamento local.

O NM irá finalizar contêineres se assim solicitar o RM ou o AM. Contêineres podem ser mortos quando o RM reporta que sua aplicação terminou, quando o escalonador decidir desalojá-lo para outra aplicação ou quando o NM detecta que a tarefa do contêiner excedeu seus limites de recursos. AMs podem requisitar que seus contêineres sejam destruídos quando não são mais necessários.

O NM também monitora periodicamente a saúde do nó, sendo capaz de diagnosticar problemas de hardware e software. Quando um problema é descoberto, o NM muda seu estado para "doente" e reporta ao RM, que pode decidir matar seus contêineres e não mais alocar recursos naquele nó até que o problema seja resolvido.

Qualquer um que queira escrever aplicações para o YARN deve se ater às seguintes responsabilidades:

1. Submeter a aplicação enviando um CLC do AM para o RM.
2. Quando o RM iniciar o AM, ele deve se registrar com o RM e periodicamente alertar sobre suas necessidades sobre um protocolo de *heartbeat*.

3. Uma vez que o RM aloque um contêiner, o AM pode construir um CLC para executar o contêiner no NM correspondente. O monitoramento do progresso do trabalho realizado dentro de um contêiner é responsabilidade estrita do AM.

4. Uma vez que o AM termina seu trabalho, ele deve se desregistrar do RM e terminar sua execução.

Podemos ver na figura 2.1 um exemplo com duas aplicações executando no cluster. Primeiramente, o cliente entra em contato com o RM para registrar e iniciar a execução do AM da sua aplicação. Conforme a necessidade da tarefa, o AM realiza a requisição de contêineres ao RM para instanciar os demais processos da aplicação que, por sua vez, executam nas diversas máquinas do cluster. Vale observar que as aplicações executam concorrentemente no cluster, que o gerenciamento da execução das tarefas é dada pelos AMs e que o RM é responsável somente por gerenciar os recursos do cluster.

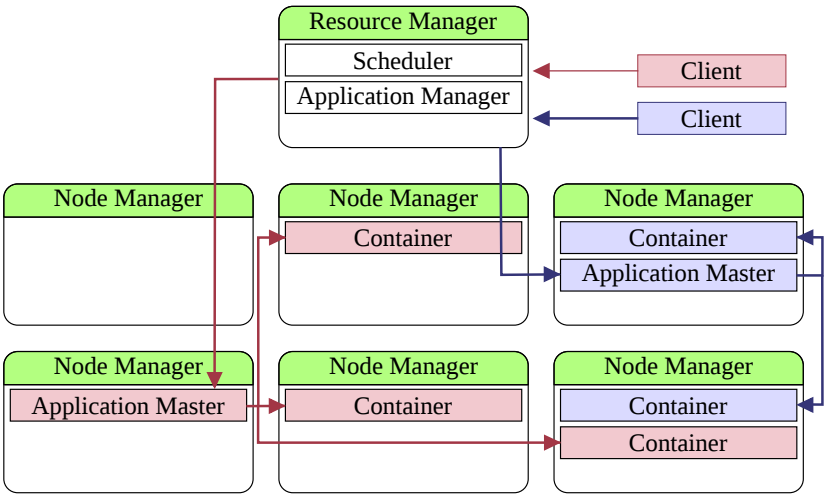


Figura 2.1: YARN gerenciando duas aplicações distintas em um ambiente de cluster.

O escalonador discutido neste trabalho tira proveito do YARN para obter as informações sobre recursos disponíveis (nós de processamento), mas processa a informação de forma a garantir que os nós selecionados para processamento contenham os dados a serem processados. Isso é feito utilizando a informação fornecida pelos sistemas de arquivos utilizados na aplicação – apresentaremos dois desses sistemas nas seções 2.2 e 2.3. Ao disparar os processos da aplicação, cada processo é configurado com a informação necessária para que ele tenha acesso aos dados locais naquela máquina.

2.2 Hadoop HDFS

Sistemas para processamento de dados massivos lidam com coleções de dados que podem esgotar a capacidade de armazenamento de uma única máquina. Além disso, processar tais coleções de dados demanda grande poder computacional. Por esse motivo, grandes coleções de dados são armazenadas de maneira distribuída em diversos nós computacionais, permitindo que grandes volumes de dados sejam armazenados além da capacidade individual de cada máquina, bem como o processamento e acesso distribuído de tais coleções de dados.

O Hadoop File System (HDFS) [Shvachko et al., 2010] é o componente de sistema de arquivos do Hadoop. HDFS é uma implementação em código aberto baseada no Google File System [Ghemawat et al., 2003]. Ele foi concebido para armazenar conjuntos de dados muito grandes com segurança e para transmitir esses conjuntos de dados com uma boa largura de banda às aplicações dos usuários.

No HDFS, o arquivo é replicado em múltiplos nós para confiabilidade. Essa estratégia também tem a vantagem de multiplicar a largura de banda para transferência de dados, e criar mais oportunidades para localizar a computação perto dos dados.

O *namespace* do HDFS é uma hierarquia de arquivos e diretórios. Arquivos e diretórios são representados em um nó de controle, o *NameNode*, por *inodes*, que armazenam atributos como permissão, modificação e acesso. O conteúdo do arquivo é dividido em grandes blocos (normalmente 128MB) e cada bloco é armazenado independentemente nos nós de armazenamento, os *DataNodes*. Para confiabilidade, esses blocos podem ser replicados em vários *DataNodes* diferentes. O *NameNode* mantém a árvore de nomes e o mapeamento dos blocos do arquivo para os *DataNodes* (a localidade física dos dados do arquivo).

Para cada cluster existe um único *NameNode*, mas podem haver milhares de *DataNodes* e clientes HDFS. O HDFS mantém todo o *namespace* em RAM. Os dados do *inode* e a lista de blocos que pertencem a cada arquivo compreendem os meta-dados do sistema de nomes chamado de *image*. O registro persistente da *image* armazenada no sistema de arquivos local é chamado de *checkpoint*. O *NameNode* também armazena o log de modificações da *image*, chamado *journal*, no sistema de arquivos local.

Cada réplica de bloco em um *DataNode* é representado por dois arquivos no sistema de arquivos local do host. O primeiro arquivo contém os próprios dados, cujo tamanho é igual ao tamanho real do bloco, e o segundo arquivo contém os metadados do bloco.

Durante a inicialização, cada *DataNode* se conecta ao *NameNode* e realiza um *handshake*. O propósito aqui é verificar o *NameSpace ID* e a versão do software do

DataNode Um *DataNode* novo que é inicializado e não possui nenhum *NameSpace ID* pode se juntar ao cluster e receber seu *NameSpace ID*. Depois do *handshake* o *DataNode* se registra com o *NameNode*. Então, ele armazena seu *Storage ID* único que é um identificador interno para reconhecimento mesmo se o *DataNode* for reiniciado com um endereço IP diferente.

Um *DataNode* informa ao *NameNode* instâncias de blocos em sua posse enviando um *block report* que contém o *block id*, a data de criação e o tamanho para cada réplica de bloco que está em sua posse. O *DataNode* também envia *hearbeats* regularmente ao *NameNode* para confirmar que está executando. Caso o *NameNode* não receba um *heartbeat* por um longo período de tempo, o *DataNode* é considerado como estando fora de serviço e seus blocos indisponíveis. O *NameNode* então escala a criação de novas réplicas desses blocos em outros nós. *Heartbeats* do *DataNode* também carregam informações sobre a capacidade total de armazenamento, a fração em uso e o número de transferências em progresso. Essas estatísticas são usadas para decisões de alocação de espaço e balanceamento do *NameNode*;

As aplicações do usuário acessam o sistema de arquivos pelo cliente HDFS, uma biblioteca de código que exporta a interface do sistema de arquivos HDFS. Similar aos sistemas de arquivos mais convencionais, HDFS suporta operações para ler, escrever e apagar/remover arquivos, e operações para criar e apagar/remover diretórios. O usuário referencia os arquivos e diretórios pelos caminhos no espaço de nomes. As aplicações do usuário geralmente não precisam saber que os metadados do sistema de arquivos e o armazenamento estão em servidores diferentes ou que os blocos possuem múltiplas réplicas.

Quando uma aplicação deseja ler um arquivo, o cliente HDFS primeiramente contata o *NameNode* em busca da lista de *DataNodes* que hospedam as réplicas dos blocos do arquivo. Em seguida contata cada *DataNode* diretamente, requisitando a transferência do bloco desejado. Quando um cliente escreve, ele primeiramente solicita que o *NameNode* escolha *DataNodes* para hospedarem as réplicas do primeiro bloco do arquivo. O cliente então organiza um pipeline dos nós e envia os dados. Quando o primeiro bloco é preenchido, o cliente requisita novos *DataNodes* para as réplicas do próximo bloco. A figura 2.2 ilustra como um arquivo é particionado em blocos de tamanhos iguais (geralmente 64 ou 128 MB), de maneira que cada bloco é replicado nas diferentes máquinas que compõem o cluster.

Depois que o arquivo é fechado, os bytes escritos não podem ser alterados ou removidos, mas novos dados podem ser anexados ao fim do arquivo. Nesse caso, o HDFS implementa um modelo de único escritor e múltiplos leitores.

Diferentemente dos sistemas de arquivos convencionais, o HDFS provê uma API

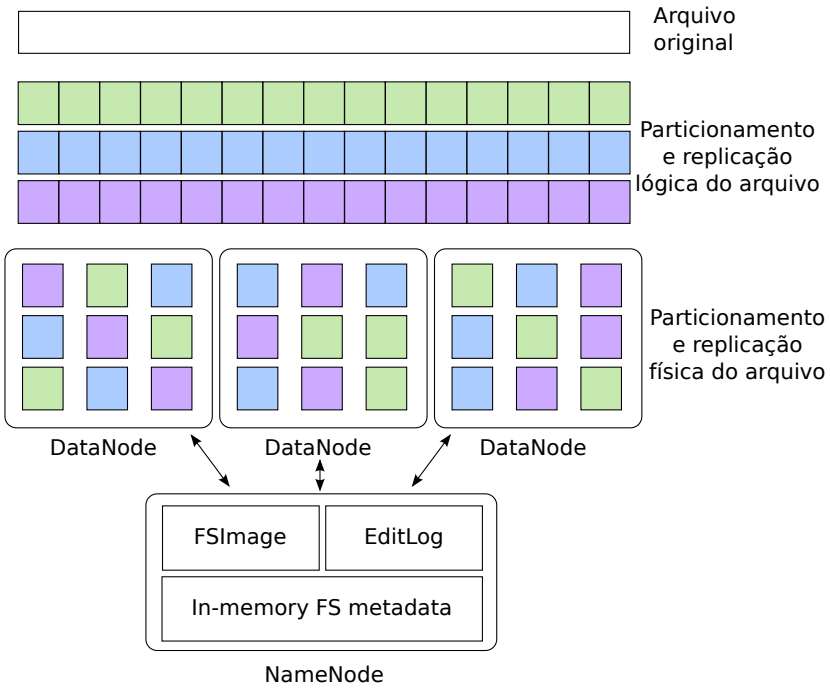


Figura 2.2: Visão geral da organização do HDFS, ilustrando o particionamento e replicação distribuída de um arquivo.

que expõe a localidade dos blocos do arquivo. Isso permite que aplicações possam escalonar a tarefa para onde os dados estão localizados, evitando transferência de dados na rede.

A política padrão de posicionamento de blocos do HDFS provê um compromisso entre minimizar o custo de escrita e maximizar a confiança dos dados, disponibilidade e largura de banda agregada de leitura. Quando um novo bloco é criado, a primeira réplica é colocada no nó onde o escritor está localizado, a segunda e terceira réplicas são armazenados em dois nós diferentes em outro rack, e o restante é colocado em nós aleatórios.

2.3 Tachyon

Em aplicações big data, operações de entrada e saída se tornam o gargalo. Quando utilizamos armazenamento distribuído, a taxa de leitura de arquivos pode ser melhorada utilizando *caching* de dados, porém a escrita é limitada pela rede e pelo disco dado que replicação de dados é utilizada para tolerância à falhas.

Os autores do Tachyon [Li et al., 2014] argumentam que os conjuntos de dados de interesse a qualquer instante (*working set*) na maioria dos datacenters são relativamente pequenos se comparados ao conjunto total de dados existentes. Por isso, advogam que

seria possível manter esses dados em memória.

Frameworks existentes armazenam dados intermediários e os dados de entrada como cache em memória para cada tarefa (job). Entretanto, o compartilhamento de dados em memória entre tarefas não é comumente implementado nesse tipo de ferramenta. Em particular, armazenar a saída de uma tarefa com segurança com a intenção de compartilhá-la com outras tarefas é um processo lento, já que o dado é replicado através da rede e discos para tolerância a falhas. Isto faz com que os sistemas de armazenamento em cluster atuais sejam ordens de magnitude mais lentos do que escrever em memória. Essa é a principal motivação dos criadores do Tachyon.

Os autores também discutem que o avanço do hardware não tende a resolver esse problema. Sendo a banda da memória até três ordens de magnitude maior do que a banda do disco. A distância entre as bandas de memória e disco tem se tornado maior. Os discos de estado sólido tem pequeno impacto nesse problema pois sua maior vantagem sobre os discos está na latência do acesso aleatório e não no acesso sequencial, que é o que a maioria das aplicações big data necessitam.

O Tachyon leva em consideração que os dados da aplicação nas bases de trabalho no cenário de big data atual são imutáveis, uma vez que os sistemas de armazenamento que estão por baixo, como GFS e HDFS, suportam operações de escrita apenas uma vez. Para a recuperação de uma tarefa que venha a falhar, diversos frameworks já utilizam recomputação, o que significa que o código dessas aplicações deva ser determinístico. Característica esta que também é presente nas tarefas que executam no Tachyon. No processamento de big data, uma mesma operação é repetida em cima de muitos dados, logo, replicar o programa é menos custoso que replicar os dados. É considerado também que, mesmo quando a base completa é demasiado grande e precise ser armazenada em discos, o *working set* de diversas aplicações pode caber em memória.

Tachyon utiliza uma arquitetura padrão de mestre-escravo similar ao HDFS, onde cada trabalhador gerencia blocos locais e os compartilha com as aplicações. Arquivos em Tachyon são organizados em uma hierarquia de árvore e são identificados por seus caminhos. Além disso, cada arquivo também contém um único e imutável ID global, chamado de FID. Em linhas gerais, a sua estrutura e interface são semelhantes às do HDFS.

Cada trabalhador executa um daemon que gerencia os recursos locais e reporta periodicamente seu status para o master. Além disso, cada trabalhador faz uso de um disco em memória RAM (RAMDisk) para armazenar arquivos mapeados em memória. Uma aplicação do usuário pode contornar o daemon e interagir diretamente com o RAMDisk. Dessa forma, uma aplicação com localidade de dados pode acessar dados à velocidade da memória.

Um dos objetivos do Tachyon é tornar menos relevante a localidade do disco, já que ele leva os dados para a memória mais próxima, distanciando o sistema de arquivos da aplicação por meio de cache. Porém, Tachyon não possui conceito de localidade, ficando a cargo da aplicação escalonar suas tarefas de maneira a usufruir dos arquivos armazenados em cache. Nesse sentido, é indispensável que a aplicação possua um escalonador de localidade baseado no Tachyon.

Do ponto de vista do nosso trabalho, Tachyon pode ser visto como uma cache com características semânticas especializadas para lidar com sistemas de arquivos distribuídos. O nosso escalonador foi desenvolvido para também interfacear com esse sistema, a fim de poder explorar não só a localidade em disco, mas também aquela em memória, potencialmente agregando o benefício da localidade ao acesso em memória local, ao invés de exigir uma transferência pela rede.

2.4 Zookeeper

Aplicações distribuídas requerem diferentes tipos de coordenação distribuída, tais como: configurações estáticas e dinâmicas de parâmetros operacionais, eleição de lider, definição de grupos e seus elementos, geralmente definindo atributos e disponibilidade de cada elemento, implementação de acesso mutuamente exclusivo por meio de *locks* distribuídos.

ZooKeeper [Hunt et al., 2010] é um serviço para coordenação de processos de aplicações distribuídas. Ele oferece uma abstração de um conjunto de nós de dados, chamados *znodes*, organizados de acordo com um espaço de nomes hierárquico. Esses espaços são definidos como estruturas de árvores comumente usadas em sistemas de arquivos. Assim como os sistemas de arquivos, para se referenciar a um dado *znode*, é utilizado a notação padrão para caminho de arquivos em sistemas UNIX. O ZooKeeper implementa um mecanismo de monitoramento que permite aos clientes receberem notificações de eventos que acontecem em um determinado caminho do espaço de nomes.

O Watershed-ng (seção 2.5) utiliza do ZooKeeper para sincronizar e coordenar a execução das instâncias de cada filtro de processamento de uma aplicação Watershed. Essa coordenação consiste basicamente em iniciar a execução de todas as instâncias de cada filtro de maneira atômica e monitorar o término de cada fluxo de dados, bem como o término de cada instância dos filtros.

2.5 Watershed-ng

O Watershed-ng [Rocha et al., 2016] é um framework para processamento distribuído de dados que nasceu de uma reengenharia do Watershed original [Ramos et al., 2011]. Este, que por sua vez, teve como base a plataforma Anthill [Ferreira et al., 2005]. Todos estes foram criados no DCC/UFMG e são baseados na abstração filtro/fluxo.

No Watershed, uma aplicação é definida como um conjunto de filtros conectados por fluxos de dados. Os dados então trafegam por meio dos fluxos e são processados por filtros. Um dos objetivos do processo de reengenharia foi transformar os fluxos em abstrações de primeira classe, que podem ser estendidas pelos usuários. Tais elementos e suas interfaces serão discutidos a seguir.

2.5.1 Filtros

Na abstração do Watershed, um elemento de processamento de dados é representado como um filtro, que possui um conjunto de portas de entrada e saída, como ilustrado pela figura 2.3. Esses filtros são ligados entre si por meios de canais de comunicação denominados fluxos.

Figura 2.3: Abstração de um elemento de processamento em Watershed.

Em uma aplicação Watershed, os filtros possuem diversas instâncias, cada uma sendo responsável por uma porção do processamento total do filtro. Obtem-se assim o paralelismo das aplicações. As instâncias de um filtro recebem os dados através de suas portas de entrada. Depois de um filtro processar os dados, ele pode enviar seus resultados através de suas portas de saída, que podem estar conectadas a outro filtro ou ao sistema de arquivos, por exemplo.

2.5.2 Fluxos de dados

Os filtros de processamento são conectados por canais de comunicação que criam a abstração de fluxo de dados. Por sua vez, os fluxos de dados são formados por um conjunto de objetos que inclui: um *sender*, um *deliverer*, uma lista de *encoders* e uma lista de *decoders*, como ilustrado na figura 2.4.

Quando um fluxo de dados é estabelecido, a comunicação real é realizada entre o *sender* atribuído ao primeiro filtro e o *receiver* do próximo. O *sender* deve implementar

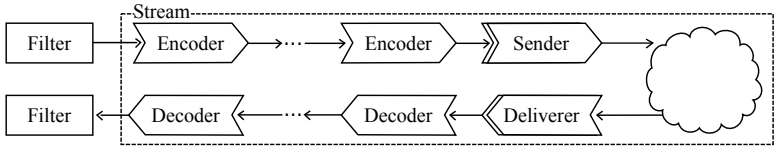


Figura 2.4: Representação da abstração de um fluxo de dados conectando dois filtros.

um método *send*, que é chamado sempre que houver dados a serem enviados pelo canal de comunicação. O protocolo de comunicação é implementado dentro desse objeto. Por outro lado, o *deliverer* é responsável por extrair dados a partir do meio de transmissão específico e então entregar para o próximo objeto da cadeia de transmissão.

Os *encoders* e *decoders* sáo opcionais e podem ser utilizados como elementos de transformação dos dados durante a transmissão, por exemplo, para criptografia, compressão de dados, agregação de mensagens, etc. Em muitos casos os *encoders* e *decoders* deverão ser inseridos no fluxo de dados como pares, por exemplo, para codificar/decodificar mensagens criptografadas ou comprimir/descomprimir dados. Porém não é necessário, por exemplo, para agregar um conjunto de mensagens.

2.5.3 Arquitetura

Em sua implementação original, o framework Watershed foi implementado de maneira monolítica em um sistema único. Todos os elementos de execução, tais como alocação de recurso, escalonamento, tolerância à falhas e interfaces de I/O, eram controladas internamente. Naquela versão, utilizava-se escalonamento estático: o usuário atribuía as instâncias de cada filtro a máquinas específicas previamente selecionadas. Também era tarefa do usuário dividir a carga de trabalho da aplicação manualmente entre as diversas máquinas do cluster.

Visando evitar tais sobrecargas ao usuário, e para desacoplar tarefas de gerencia do cluster do núcleo do Watershed, sua reengenharia utilizou dos componentes do ambiente Hadoop. Isso também teve o efeito positivo de permitir a integração do Watershed àquele ambiente. O YARN foi utilizado como escalonador e alocador de recursos, permitindo que aplicações de diferentes plataformas coexistam em um mesmo ambiente de cluster. ZooKeeper [Hunt et al., 2010] foi utilizado para implementar o serviço de coordenação dos múltiplos processos de cada aplicação, o HDFS foi utilizado tanto como um meio de distribuição dos arquivos executáveis quanto como um sistema de arquivos distribuídos para a camada de aplicação.

Além dos serviços descritos anteriormentes, a plataforma Watershed-ng compreende três componentes específicos: (*i*) o *JobClient*, responsável por submeter uma aplicação ao Watershed-ng; (*ii*) o *JobMaster*, que controla a execução dos processos

que compõem a aplicação; e (*iii*) os *InstanceDrivers*, que executam e monitoram cada instância dos filtros. Uma visão geral da arquitetura está ilustrada na figura 2.5.

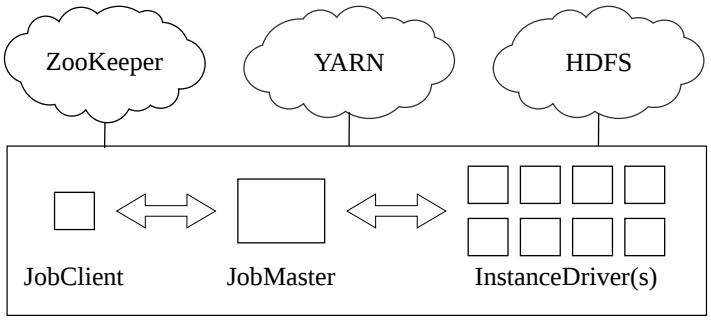


Figura 2.5: Visão geral em alto nível da arquitetura do Watershed-ng implementada sobre o ecossistema hadoop.

2.5.4 Iniciando uma Aplicação

Nesta seção nós descreveremos a sequência de etapas executadas ao iniciar a execução de uma aplicação em Watershed-ng, incluindo a interação com os módulos do ecossistema Hadoop. Essa sequência é mostrada na figura 2.6.

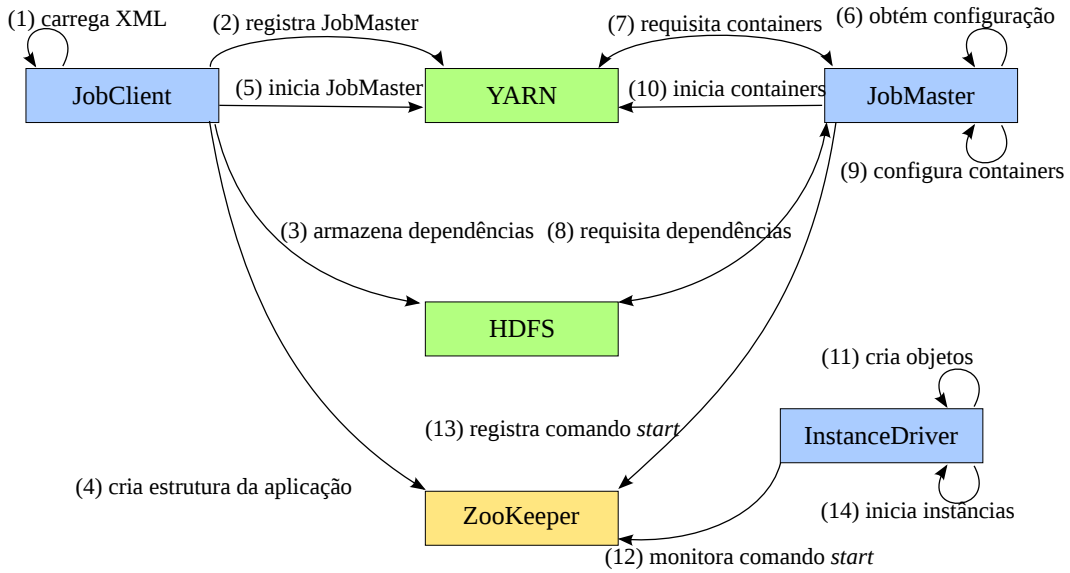


Figura 2.6: Diagrama com a sequência dos principais eventos realizados pelos componentes da plataforma Watershed-ng para iniciar uma aplicação.

Quando um cliente deseja submeter uma aplicação para executar no Watershed-ng, ele deve iniciar um JobClient com os arquivos de configuração que descrevem quais os filtros que formam aquela aplicação e a comunicação entre eles. Fica a cargo do JobClient analisar os arquivos de configuração (1) e registrar a aplicação junto ao

YARN (2). Além disso, o JobClient deve armazenar as dependências da aplicação, como arquivos executáveis e bibliotecas, no HDFS (3), criar a estrutura da aplicação no Zookeeper para o espaço de nomes da aplicação a ser executada (4), necessário para a coordenação distribuída das tarefas. Só então a aplicação é submetida junto ao YARN (5).

Uma aplicação recém iniciada pelo YARN é composta pelo *JobMaster*, que é responsável pela gerência da aplicação no cluster. O JobMaster basicamente recebe e decodifica os descritores dos módulos (6), requisita contêineres para execução das instâncias dos filtros (7), requisita as dependências previamente armazenadas no HDFS (8), realiza a configuração dos contêineres (9) e então inicia os contêineires no cluster YARN (10).

Em cada um dos contêineres é iniciado um InstanceDriver que é responsável por criar os objetos que formam aquela instância do filtro (11) registrar seu estado inicial no ZooKeeper (12) e, por fim, aguardar um sinal do *JobMaster* para iniciar a execução das instâncias (13). Quando o sinal é recebido, cada *InstanceDriver* gerencia a execução da sua instância do filtro.

Por meio do ZooKeeper, o *JobMaster* gerencia a execução de todas as instâncias dos filtros da aplicação. Uma vez terminadas, o *JobMaster* retorna o controle ao *JobClient* que termina por limpar o ambiente de execução da aplicação.

2.6 Demais trabalhos relacionados

Os artigos que descrevem Hadoop, Spark, HDFS (e GFS) e Tachyon, já mencionados, discutem alguns dos aspectos de decisão envolvidos na escolha de nós de processamento ou armazenamento em cada caso. Nessa seção, apresentaremos outros trabalhos que estão relacionados com a nossa pesquisa.

Um dos primeiros trabalhos a discutir o impacto da localidade no processamento de grandes volumes de dados no contexto de datacenters modernos é devido a Barroso & Hölzle [2009]. Nele os autores chamam a atenção para as diferenças em termos de acesso entre discos e memória RAM em uma mesma máquina, em um mesmo rack ou em racks diferentes. Essas diferenças, que se tornam mais significativas entre máquinas em racks diferentes, foram a base para decisões de projeto como as do Hadoop.

No artigo de Schwarzkopf et al. [2013] é apresentado que a maioria das aplicações (mais de 80%) são tarefas em *batch*, porém a maioria dos recursos ($55 - 80\%$) são alocados para aplicações de serviços. Estas, tipicamente, executam por muito mais tempo e possuem menos tarefas do que as aplicações em *batch*. Essa informação é

importante para levarmos em conta a heterogeneidade da carga de trabalho desses tipos de ambientes de processamento. Enquanto as aplicações em *batch* são curtas e dispensam um escalonador mais sofisticado, pois o *overhead* adicionado não se justifica, as aplicações de longa duração se aproveitam de uma melhor localidade.

Segundo o artigo, o escalonador apresentado neste trabalho é classificado como um **escalonador de dois níveis** e pode ser descrito como um escalonador dinâmico que utiliza um coordenador central que decide quanto recurso cada sub-cluster pode ter. Nosso escalonador se encaixa nessa descrição pois ele possui interface com o YARN, que é o gerenciador de recursos central do cluster. No escalonador do Watershed-ng, o escalonamento é feito e as máquinas selecionadas são requisitadas ao YARN, que pode não atender por falta de recursos, por exemplo. Diferentemente, no Omega não existe um alocador de recursos centralizado, todas as decisões de alocação de recursos são realizadas pelos escalonadores presentes no cluster.

Para sustentar os escalonadores sofisticados dos frameworks atuais, Mesos [Hindman et al., 2011] introduz um mecanismo de escalonamento de dois níveis chamado de *oferta de recursos*. Sua arquitetura consiste de um processo mestre que gerencia *daemons* escravos que executam em cada nó do cluster. O mestre decide a quantidade de recursos que ele deve oferecer para cada *framework* enviando para cada um deles uma lista de slaves disponíveis. Nesse momento, os escalonadores dos frameworks decidem quais desses recursos oferecidos usar e, então, passa para o mestre uma descrição das tarefas que ele deseja executar. Mesos também dá aos frameworks a abilidade de rejeitar ofertas que não satisfaçam suas restrições e então aguardar por outras. Assim como o YARN, Mesos é um gerenciador de recursos e as políticas de escalonamento que incidirão sobre o cluster são implementadas separadamente.

Na literatura, temos alguns escalonadores que têm como objetivo utilizar eficientemente os recursos de ambientes multiusuário. O HaSTE [Yao et al., 2014] é um escalonador integrado ao YARN, desenvolvido especificamente para aplicações MapReduce, cujo foco principal é reduzir o tempo total de conclusão de múltiplas aplicações em execução concorrente, além de também reduzir a utilização de recursos do cluster. Esse objetivo é alcançado considerando os recursos requisitados, a capacidade geral do cluster e as dependências entre as tarefas de cada aplicação. Ainda para ambientes multiusuário, Zaharia et al. [2010a] apresentam uma técnica de escalonamento centrado em localidade de dados, chamada de *delay scheduling*, também desenvolvendo um escalonador para tarefas MapReduce no ambiente Hadoop. Os autores discutem uma heurística onde as tarefas são adicionadas à uma fila enquanto as máquinas que armazenam os dados necessários estiverem ocupadas. Essa abordagem contribui para o uso equilibrado dos recursos entre os usuários ao mesmo tempo que visa localidade

dos dados.

Isard et al. [2009] apresentam um escalonador chamado Quincy para a plataforma distribuída Dryad [Isard et al., 2007]. O Quincy mapeia o problema de escalonamento como um grafo de fluxos, onde as arestas representam os diversos custos envolvidos no escalonamento de cada tarefa entre os diversos nós disponíveis no cluster, considerando aspectos como localidade de dados e balanceamento de carga de trabalho entre os nós.

Ousterhout et al. [2010] advogam o uso de sistemas de arquivos em RAM (Ramclouds) para evitar o custo do acesso a disco. Nesse sentido, aquele trabalho é ortogonal ao apresentado aqui, já que nosso objetivo é levar o processamento para a mesma máquina onde o dado se encontra, seja em memória ou disco. Os experimentos com Tachyon mostram que essa exploração do conceito de localidade pode ser até mais benéfico se o dado já se encontra em memória.

O trabalho de Ananthanarayanan et al. [2011] discute que o cerne para o uso de localidade de disco é que a banda de disco excede a banda da rede. Também é dito que I/O do disco constitui em uma fração considerada menor quando comparada ao tempo de execução total de uma tarefa. Porém é apresentado que a tecnologia de rede está evoluindo em um ritmo mais acelerado do que a tecnologia do disco, sendo que, para um hardware típico, ler do disco local é somente 8% mais rápido do que ler de um disco de outro nó que esteja no mesmo rack. Outro ponto importante de ser observado é que os dados estão sendo armazenados comprimidos nos datacenters, o que faz com que a leitura deles pela rede ofereça um menor impacto para a aplicação. Porém, localidade de memória ainda desempenha um papel fundamental e será a nova peça do jogo para o design de frameworks para computação em cluster. Em uma primeira análise, este trabalho se contrapõe ao trabalho de Ananthanarayanan et al. [2011], já que ele vai contra a preocupação com localidade de discos. Porém, nosso argumento neste trabalho não tem por objetivo contrariar aquele nessa previsão. Apenas mostramos que, em condições atuais, a localidade ainda é significativa. Além disso, mostramos também que em um contexto com dados armazenados em memória, o escalonamento baseado na localidade dos dados em memória também pode resultar em ganhos de desempenho.

Capítulo 3

Implementação

A implementação de nossa solução envolve o desenvolvimento dos seguintes módulos principais: o escalonador de tarefas, que integra informações do HDFS à interação do framework com o YARN, o leitor e o escritor para HDFS/Tachyon.

3.1 Escalonador HDFS/Tachyon

A execução de uma aplicação dentro do ecossistema Hadoop, seja ela baseada no Hadoop MapReduce, Spark ou outros ambientes de programação, implica na definição de pelo menos duas informações principais: a identificação dos dados de entrada a serem processados (p.ex., o arquivo de entrada) e das tarefas que devem ser executadas (inclusive quantas instâncias de cada tarefa devem existir).

De posse dessas informaçãoes, o funcionamento do escalonador proposto é dado segundo o desenho esquemático observado na figura 3.1. Primeiramente, o escalonador acessa o sistema de arquivo e obtém as informações sobre os arquivos envolvidos, contendo a lista de blocos e a identificação da localização de cada réplica dos mesmos. De posse dessas informações, é possível solicitar ao YARN os contêineres nos locais exatos desejados. Pela forma de operação do YARN, os contêineres para as solicitações enviadas podem ser retornados fora de ordem, então o escalonador deve processar cada resposta para identificar o nó obtido e associá-lo aos blocos presentes naquele nó. Finalmente, os processos podem ser disparados nos diversos nós com a informação sobre quais blocos eles devem acessar (que serão blocos locais).

O escalonador então utiliza um mecanismo de requisição sem relaxamento na localidade dos contêineres YARN, forçando que os mesmos sejam alocados considerando exatamente as localizações especificadas, derivadas das informações obtidas das fontes de dados (HDFS/Tachyon).

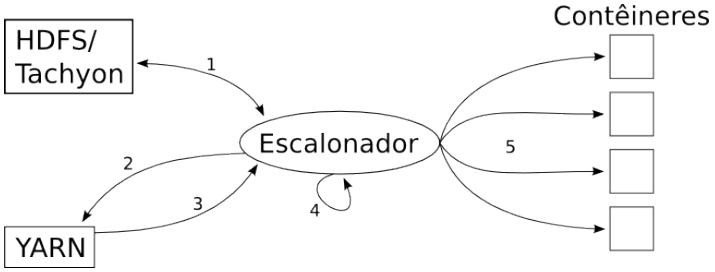


Figura 3.1: Passos de operação do escalonador: (1) busca dos metadados do arquivo no HDFS/Tachyon com lista de blocos, (2) pedido dos contêineres, (3) recebimento dos contêineres fora de ordem, (4) organização dos contêineres em função dos blocos e (5) execução dos processos da aplicação.

O Tachyon possui um funcionamento conceitualmente equivalente ao HDFS, porém os blocos de um dado arquivo podem em qualquer momento ser encontrados em blocos no disco ou já carregados na memória do sistema. Dessa forma, seu escalonador foi desenvolvida de maneira semelhante. Porém, agora são priorizados os nós que contenham blocos do arquivo em memória principal. Caso esses blocos em memória não existam, então a localização dos dados em disco é considerada.

3.1.1 Assinalamento de tarefas e blocos

O processo de escolha dos hosts que irão receber os contêineres de leitura é apresentado no algoritmo 3.1. O método responsável pela execução do escalonador é o `runScheduler`. Nele, os hosts são escolhidos de maneira que a leitura do arquivo seja realizada, sempre que possível, localmente. Primeiramente, o escalonador faz uma consulta ao sistema de arquivos distribuído (HDFS/Tachyon) para obter a localização dos blocos (incluindo todas as réplicas) que armazenam a porção dos dados de entrada (linha 17). A partir da lista de blocos retornada pelo sistema de arquivos, identificamos os nós que os hospedam e criamos uma estrutura de dados que representa cada um desses hosts. Essa estrutura, denominada `BSHost`, armazena a lista de blocos que cada uma das máquinas hospeda, além de manter registro sobre quais dos seus blocos já foram escolhidos pelo escalonador.

Os nós identificados são posteriormente ordenados pela quantidade de blocos existentes em cada um, de forma a priorizar o nó do cluster com a maior quantidade de blocos – essa tarefa é feita internamente ao método `toHosts`, que pode ter sua utilização observada na linha 21. E então os blocos são distribuído entre os hosts de maneira uniforme. O número de blocos que cada instância receberá pode ser verificado pela variável `bpi`.

Passamos então para o processo de escalonamento dos blocos que é dividido em

```
1  public class HDFSBlockScheduler extends WSScheduler{
2    int[] bpi;               //blocks per instance
3    List[] instanceBlocks;   //blocos para cada instancia
4    String[] instanceHosts; //hosts para hospedar cada instancia
5
6    /**Decide onde os blocos serao executados. */
7    public void runScheduler(){
8      FileSystem fs = FileSystem.get( conf );
9      FileStatus file = fs.getFileStatus( path );
10     BlockLocation[] blocks =
11       fs.getFileBlockLocations( file, 0, file.getLen() );
12     //cria uma lista de Hosts ordenada pelo número de blocos
13     //que cada host possui.
14     BSHost[] hosts = toHosts( blocks );
15     bpi = distribute( blocks.length, numInstances );
16
17     //blocos locais
18     for( int i = 0; i < numInstances; i++ ){
19       instanceBlocks[i] = hosts[i].markLocals( bpi[i] );
20       instanceHosts[i] = hosts[i];
21     }
22
23     //restante dos blocos a serem escalonados
24     BSBlock[] unscheduledBlocks = getNonLocals( hosts );
25     for( i = 0; i < numInstances; i++ )
26       //enquanto faltar blocos para instancia i
27       while( instanceBlocks[i].size() < bpi[i] )
28         instanceBlocks[i].add( unscheduledBlocks.next() );
29
30     //envia blocos ao filtro leitor
31     super.setAttribute( schedulerChannel, "schedulerBlocks",
32                         instanceBlocks );
33   }//runScheduler
34
35   /**@param hosts hosts a serem escalonados
36     *@return      blocos não escalonados */
37   BSBlock[] getNonLocals( BSHost[] hosts ){
38     List<BSBlock> nonLocals; //lista para os blocos não locais.
39     for( BSHost host : hosts )
40       nonLocals.addAll( host.getNonLocals() );
41     return nonLocals;
42   }
43
44   /**Interface para que o JobMaster acesse o escalonamento.
45     *@param instanceId instancia do filtro.
46     *@return           host escolhido pelo escalonador */
47   public String getInstanceLocation( int instanceId ){
48     return instanceHosts[ instanceId ].hostName();
49   }
50 }//HDFSBlockScheduler
```

Algoritmo 3.1: Trecho de código do escalonador baseado em localidade, implementação utilizando HDFS.

duas fases: a primeira é a escolha dos blocos locais, onde, para cada instância, o escalonador seleciona os blocos que serão processados localmente por um determinado host. Para isso, o método `markLocals` da classe `BSHost` é utilizado. Esse método seleciona uma quantidade desejada de blocos pertencente ao host, ou uma quantidade menor caso o host não possua o número necessário de blocos disponíveis. Depois que os blocos locais são selecionados em um determinado host, o escolhemos para hospedar a instância em questão.

Na segunda fase, realizamos o escalonamento dos blocos que serão lidos remotamente. Como o armazenamento dos blocos é realizado pelo HDFS, não podemos garantir que eles estejam distribuídos uniformemente pelo cluster. É possível gerar situações em que, para determinados hosts, o número de blocos armazenados seja inferior à quantidade de blocos necessária à instância que executará naquela máquina. Por esse motivo, o escalonador gera uma lista de blocos que ainda não foram selecionados por meio do método `getNonLocals`. Esse método pode ser observado nas linhas 44-49 e, simplesmente, gera uma lista concatenando os blocos disponíveis para o escalonador em cada host. De posse desses blocos, o escalonador realiza uma segunda passada completando cada uma das listas de blocos selecionados para cada uma das instâncias. Até que todas as instâncias tenham recebido a quantidade de blocos que lhe cabe.

Por fim, os blocos selecionados para todas as instâncias são passados para o filtro leitor por meio da função `setAttribute` que faz parte da API do Watershed-ng. A implementação do escalonador também deve possuir a função `getInstanceLocation` (linhas 54-56) que faz parte da interface utilizada pelo JobMaster para acesso ao escalonamento.

3.1.2 Interface Watershed-ng

O Watershed-ng foi criado para ser modular, permitindo que o usuário possa adicionar novas funcionalidades ao framework. Dessa forma, o escalonador também foi pensado para permitir que o usuário crie suas próprias políticas ou expanda as já existentes.

Se o usuário necessitar de outros escalonadores ele pode implementá-los herdando da classe abstrata `WSScheduler`. Essa classe contém métodos para manipular possíveis atributos relacionados ao escalonador, assim como o número de instâncias do filtro atrelado à ele. Fica a cargo do programador a implementação dos métodos abstratos `runScheduler` e `getInstanceLocations`. Voltando a observar o trecho de código 3.1, verificamos que esses métodos foram implementados no código do escalonador propostos (linhas 7-33 e 47-49).

`runScheduler` é o método principal do escalonador. Ele é responsável por entrar

em contato com o gerenciador de arquivos (HDFS/Tachyon), descobrir a localidade dos dados e decidir onde cada instância do filtro irá executar. Depois de executado o cerne do escalonador, o método `getInstanceLocations` é responsável por retornar uma lista de máquinas hábeis a receber alguma instância em particular do filtro.

No Watershed-ng, a classe `JobMaster` é responsável por gerenciar a execução da aplicação e enviar seus módulos para serem executados nas máquinas disponíveis do cluster YARN. Logo, é nesse ponto que o escalonador é executado e, uma vez que temos informação sobre quais máquinas foram escolhidas para executar cada instância da aplicação, os contêineres são requisitados. Por fim, o `JobMaster` deve iniciar a execução das instâncias nos contêineres disponibilizados pelo YARN. Mais informações serão vistas na próxima seção.

3.1.3 Disparo da execução

Como descrito anteriormente, o `JobMaster` é responsável pela coordenação de uma aplicação no cluster YARN, incluindo fazer requisições de contêineres e disparar todas as instâncias de cada filtro pertencente à aplicação. É sua responsabilidade, então, interagir com o escalonador para alocar os contêineres da melhor forma. Podemos observar pelo algoritmo 3.2 que é nesse momento que o escalonador passado pelo cliente é instanciado, seus atributos são configurados e então sua execução é iniciada por meio da chamada do método `runScheduler`.

Com base no escalonamento realizado, um contêiner para cada instância é requisitado ao YARN. Pelas linhas 8-10, podemos observar que os hosts escolhidos pelo escalonador são obtidos pela função `getInstanceLocation` e, então, as requisições aos contêineres são realizadas através do método `containerRequest` que faz parte da API do YARN.

Passado algum tempo, um contêiner é recebido pelo Watershed-ng de forma assíncrona. Porém, não existem garantias de que as requisições que foram feitas serão atendidas pelo YARN, nem que os contêneires serão disponibilizados na mesma ordem em que foram requisitados. Por esse motivo, uma vez que um contêiner é recebido, o método `onContainerAllocated` é executado, e sua primeira tarefa é selecionar a melhor instância para ser executada naquele contêiner. Para isso, temos o método `getBestFilterInstance` (linhas 24-28), que pode tomar uma das seguintes ações: (1) primeiramente ele procura uma instância que tenha sido escalonada para o host que hospeda o contêiner em questão. Caso não encontre, (2) ele procura alguma tarefa da aplicação que não precise ser escalonada e, portanto, possa ser executada nesse contêiner sem maiores implicações. Por último, caso as opções anteriores tenham falhado,

```
1  //inicialização do escalonador
2  WSScheduler wsScheduler =
3    (WSScheduler) Class.forName( "schedulerClassName" );
4  wsScheduler.setAttributes( "schedulerAttributes" );
5  wsScheduler.runScheduler();
6
7  //requisição dos contêineres para execução das instâncias.
8  for( i = 0; i < numInstances; i++ ){
9    String location = wsScheduler.getInstanceLocation( i );
10   YARN.containerRequest( location );
11 }
12
13 /**submissão da instância.
14  * @param container contêiner recebido */
15 onContainerAllocated( Container container ){
16   bestInstance = getBestFilterInstance( container );
17   configureContainer( container, bestInstance )
18   YARN.startContainer( container );
19 }
20
21 /**Decide qual é a melhor instância a ser executada no contêiner.
22  * @param container contêiner disponível
23  * @return          instância escolhida */
24 InstanceInfo getBestFilterInstance( Container container ){
25   escolhe a instância que foi selecionada para o host;
26   senão, escolhe tarefa que não necessita de escalonamento;
27   senão, escolhe uma tarefa qualquer;
28 }
```

Algoritmo 3.2: Trecho de código da classe JobMaster do Watershed, responsável pela requisição e disparo dos contêneires baseados no escalonador.

(3) o método escolhe uma tarefa qualquer para ser executada naquele host, portanto, o acesso aos dados não será realizado localmente. O último cenário ocorre quando a requisição pela máquina pretendida não é satisfeita. Esta rejeição pode ocorrer caso o YARN entenda que um contêiner naquele host não deva ser instanciado pois está indisponível ou sobrecarregado, por exemplo.

Por fim, é realizada a configuração do contêiner, contendo o filtro que será executado, informações a respeito da instância específica, etc. e, então, o contêiner é disparado. Também é necessário que o Watershed tenha conhecimento da tabela de escalonamento gerada anteriormente e gerencie a execução das tarefas, mantendo um registro das instâncias que ainda devem ser executadas. Além disso, o ZooKeeper é utilizado para disparar a execução de um determinado filtro somente quando todos os módulos anteriores a ele (suas dependências) tiverem executado.

```
1  <filter name="wsreader" file="sample.jar"
2    class="sample.benchmark.Reader" instances="2">
3    <input>
4      <channel name="reader">
5        <deliverer class="hws.channel.hdfs.HDFSLineReaderBS">
6          <attr name="path" value="/datasets/in.dat"/>
7        </deliverer>
8      </channel>
9    </input>
10
11   <scheduler class="hws.scheduler.HDFSBlockScheduler">
12     <attr name="channel" value="reader"/>
13     <attr name="path" value="/datasets/in.dat"/>
14   </scheduler>
15
16   <output>
17     <channel name="writer"/>
18       <sender class="hws.channel.hdfs.HDFSLineWriter">
19         <attr name="path" value="/reader-out.dat"/>
20       </sender>
21     </channel>
22   </output>
23 </filter>
```

Algoritmo 3.3: Arquivo de configuração que define um módulo Watershed-ng (XML).

3.1.4 Utilização

A arquitetura de uma aplicação Watershed é descrita via arquivos xml que descrevem os módulos da aplicação. Esses arquivos possuem a descrição do filtro, a quantidade de instâncias que executarão no ambiente, os canais de comunicação utilizados por ele e o escalonador utilizado.

Para utilizar um escalonador atrelado a um determinado filtro é necessário informar a classe que o implementa e os atributos que forem necessários, como caminho do arquivo e endereço do servidor de arquivos, por exemplo. O algoritmo 3.3 apresenta a descrição de um módulo Watershed com um escalonador HDFS. Pelo código, percebemos que é necessário informarmos a classe que implementa o filtro e a quantidade desejada de instâncias que serão executadas no cluster.

Primeiramente, foi descrito o canal de comunicação responsável pela entrada de dados. Podemos observar que o leitor HDFS `HDFSLineReaderBS`, apresentado na seção 3.2, foi utilizado como `deliverer` do canal de comunicação `reader`. Seu arquivo de entrada é passado como um atributo chamado `path`. Em seguida, o escalonador `HDFSBlockScheduler`, apresentado na seção 3.1.1, é utilizado. Note que, além da classe que o implementa, temos dois atributos que informam qual arquivo será escalonado,

assim como o canal de comunicação que vai ler esses dados. Essa última informação é importante para que o escalonador possa enviar a lista de blocos escalonados para o leitor correto.

Por fim, é descrito o canal de comunicação de saída, aqui chamado de `writer`. Nesse exemplo, foi utilizado o `sender` chamado `HDFSLineWriter`, que tem como funcionalidade escrever o resultado do processamento do filtro no arquivo do HDFS descrito pelo atributo `path`.

3.2 Módulos de integração com HDFS/Tachyon

Como descrito na seção 2.5.2, no Watershed-ng, a interface de dados é implementada por `senders`/`deliverers` especiais. Para a criação dos canais de comunicação descritos aqui, implementamos classes `Deliverer` do Watershed-ng, utilizando elementos das APIs do HDFS/Tachyon.

Para que o processamento dos dados seja realizado de maneira distribuída, é importante que o canal de leitura seja capaz de dividir a leitura do arquivo de entrada entre as diversas instâncias que o filtro leitor possa ter. Como foi visto anteriormente, essa divisão é realizada pelo escalonador em termos de uma lista de blocos que é repassada a cada leitor. Cada instância do filtro lerá, então, o conjunto de blocos que lhe couber, ficando a cargo do escalonador passar a lista de blocos que serão lidos por cada instância.

O código do leitor HDFS para leitura de blocos pode ser visto no algoritmo 3.4. Temos então a classe `HDFSLineReaderBS` que é responsável pela leitura de blocos de um arquivo armazenado em HDFS, blocos estes que foram previamente escolhidos pelo escalonador proposto. Toda a leitura dos blocos é realizada pelo método `start`. Primeiramente, obtemos um descritor do arquivo do HDFS. A lista de blocos é passada ao leitor codificada como um string e sua primeira tarefa é decodificar esse atributo (o processo de decodificação não foi apresentado aqui por simplificação). De posse da lista de blocos que o leitor deve acessar, realizamos a leitura, via API HDFS, de cada bloco, registro a registro, e enviamos ao filtro correspondente via API Watershed-ng. É importante ressaltar que a API HDFS não leva a localidade do arquivo em consideração diretamente. Portanto o comando *seek*, utilizado no leitor, simplesmente prepara o leitor para realizar a leitura em qualquer DataNode, daí a importância do escalonador Watershed fazer o assinalamento dos blocos de forma a garantir a localidade.

É importante levar em consideração que as linhas do arquivo possivelmente ultrapassam os limites entre os blocos. Por esse motivo, o método `nextLine` vai além do

```
1  import org.apache.hadoop.fs.FileSystem;
2  import org.apache.hadoop.fs.FSDataInputStream;
3  import org.apache.hadoop.io.Text;
4  import org.apache.hadoop.util.LineReader;
5
6  public class HDFSLineReaderBS extends Deliver{
7    private FileSystem fs; //api do sistema de arquivos HDFS
8    private LineReader in; //leitor de arquivo do hadoop
9    private Text line;     //ultima linha lida
10
11   /**Estabelece comunicação com o sistema de arquivos e realiza
12    * a leitura dos blocos sob responsabilidade da instância */
13   public void start(){
14     fs = FileSystem.get( conf );
15     FSDataInputStream fileIn = fileSystem.open( path );
16     blocksAttr = getAttribute( "schedulerBlocks" );
17     BSBlock[] blocks = decode( blocksAttr );
18
19     //leitura dos blocks
20     for( BSBlock block : blocks ){
21       fileIn.seek( block.start() );
22       in = new LineReader( fileIn );
23
24       //se esse não é o primeiro bloco, sempre descartamos a
25       //primeira linha porque sempre (exceto no último bloco)
26       //lemos uma linha extra no método next().
27       if (block.pos() != 0)
28         block.pos() += in.readLine(null, maxBytesToConsume());
29
30       //o bloco é lido e enviado ao filtro linha à linha
31       while( nextLine( block ) ){
32         deliver( line.toString() );
33       }
34     }
35   }//start()
36
37   /**Lê a próxima linha do bloco e armazena em line.
38    * @return leitura realizada? */
39   private boolean nextLine( block ){
40     //verifica se leitura chegou ao fim do bloco
41     if( block.pos() < block.end() ){
42       block.pos += in.readLine(line, maxBytesToConsume());
43       return true;
44     }else
45       return false;
46   }
47
48   /**Certifica que a função não lerá mais do que lhe cabe.
49    * @return número de bytes a serem lidos*/
50   private int maxBytesToConsume() {
51     return (int) ( block.end() - block.pos() );
52   }
53 }// lineReader
```

Algoritmo 3.4: Trecho de código do leitor de arquivos Watershed-ng, implementação para HDFS.

bloco atual para terminar a leitura da última linha do bloco. Logo, temos de descartar a primeira linha do bloco – exceto no primeiro bloco – antes de começar a leitura. Essa tarefa é realizada pelas linhas 27-28 do código 3.4.

Observamos no código (linhas 31-33) que a leitura de cada registro é realizado pelo método `nextLine` e que o envio da leitura é enviada ao filtro por meio do método `deliver`. O código de `nextLine` é descrito pelas linhas 39-46. Antes da leitura de cada registro, o limite do bloco atual é verificado. E a leitura é realizada utilizando o método `readLine` disponível na API HDFS, ficando o registro armazenado na variável `line`. Esse método também leva em consideração os limites do bloco que é calculado pela função `maxBytesToConsume` (linhas 50-52). Já o método `deliver` é implementado pela API do Watershed-ng.

Também foi implementado um leitor de arquivos integrado ao sistema de arquivos distribuídos Tachyon. Seu funcionamento se dá de maneira semelhante àquele implementado para o HDFS. Por esse motivo, a explicação deste módulo será omitida deste trabalho.

3.3 Considerações finais

Para a implementação do escalonador proposto, a reengenharia do Watershed e seu acoplamento com o ecossistema Hadoop foram imprescindíveis. Com a integração à API HDFS, o sistema agora é capaz de acessar os blocos de um arquivo de forma simples e independente para cada instância de filtro. A informação sobre posicionamento dos blocos passou a estar disponível para o processo controlador do Watershed, que pode então escolher o melhor posicionamento para cada filtro que acessa tais blocos.

Também vale ressaltar que o objetivo deste trabalho foi de analisar o impacto da localidade dos dados no ambiente Watershed. Por esse motivo, implementamos escalonadores para os filtros que acessam os dados armazenados em disco, esses que usualmente se encontram nas pontas da aplicação. Porém, o escalonador também é útil para os filtros intermediários, reduzindo o tráfego de rede no cluster, e para filtros que utilizem outros meios de comunicação. Para isso, o escalonador deve, de alguma forma, verificar onde os dados são gerados e instanciar, naquele nó, o filtro consumidor por intermédio do YARN. Por exemplo, se temos um filtro que irá consumir dados da Internet, o escalonador deverá conseguir informação sobre qual máquina do cluster está capturando os dados da Web e então tentar instanciar o filtro naquele nó.

Capítulo 4

Desempenho

Este capítulo descreve nossos experimentos para validar o escalonador e avaliar o impacto da localidade. Para fins de comparação, e para validação, foram criadas duas versões do escalonador: o escalonador proposto e o aleatório. O escalonador proposto, como descrito anteriormente, verifica onde os blocos dos arquivos estão armazenados e escalona as instâncias dos filtros para que a entrada de dados seja feita localmente. Por sua vez, o escalonador aleatório ignora a localidade dos blocos e os distribui aleatoriamente entre as instâncias, que também são alocadas de modo aleatório entre todas as máquinas do cluster.

Para avaliar o desempenho do escalonador, executamos uma aplicação que lia os dados de um arquivo de entrada e computava estatísticas simples sobre o mesmo. Essa aplicação é semelhante aquela discutida na seção 3.1.4 e o código do módulo que a descreve pode ser visto no trecho de código 3.3. Porém, com a diferença que, para os experimentos, não utilizamos o canal de saída de antes. Aqui, o resultado do processamento do filtro é descartado.

Além do escalonador proposto, a aplicação foi executada ainda com o escalonador aleatório. Nas figuras a seguir, os resultados do nosso escalonador e do aleatório são identificados pelas siglas “esc” e “ale” respectivamente.

Os testes foram executados em um cluster com 10 máquinas virtuais (VMs), onde cada uma delas possuía 2 VCPUs de 2,5 GHz e 4 GB de memória RAM. Cada máquina virtual foi associada a um disco na própria máquina física em que a VM foi criada, para garantir o controle preciso de localidade física dos blocos. Todos os testes foram executados cinco vezes. Os tempos de execução apresentados se referem à média das execuções; o desvio padrão de cada conjunto de execuções é sempre apresentado nas figuras.

4.1 Validação

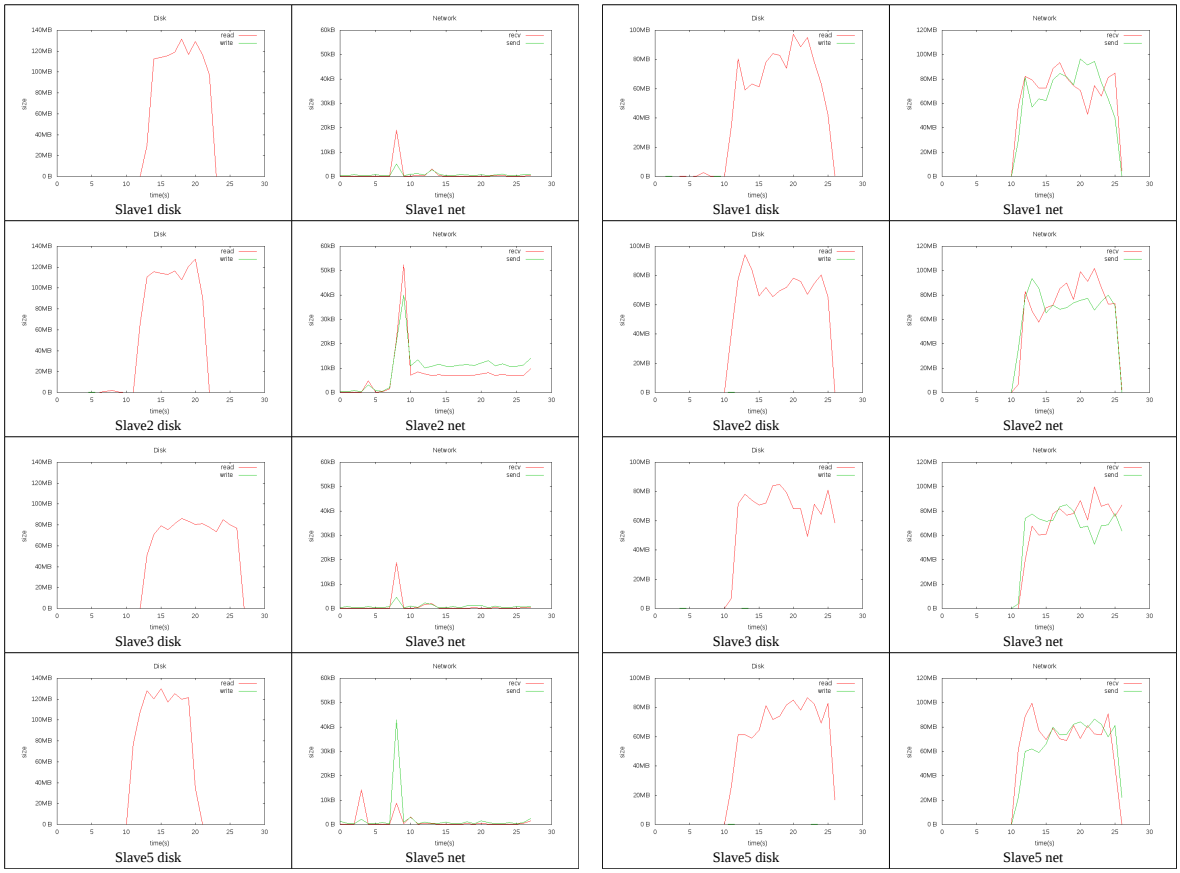
Para confirmar se o escalonador estava operando corretamente, utilizamos o comando linux `dstat` para coletar dados sobre o tráfego de disco e de rede em cada máquina durante os testes. Os dados coletados para essas duas grandezas comprovam que, para o escalonador proposto, os nós apresentavam um acesso a disco condizente com os dados locais a serem lidos, sem tráfego de rede significativo, exceto por operações de sincronização no ambiente de processamento. Já no caso da política que sempre assinalava um processo para executar em um nó onde os dados não estavam, os dados mostravam que o padrão de acesso a discos era o mesmo do tráfego de rede enviado pelo nó – isto é, todas as leituras de disco eram destinadas a atender um processo em outro nó. Ao mesmo tempo, o padrão de dados lidos pela rede correspondia ao padrão de leitura de disco na máquina que continha os dados exigidos pelo processo da aplicação executando naquele nó.

As figuras 4.1a e 4.1b ilustram, respectivamente, uma execução de uma aplicação com quatro nós de processamento, com o escalonador sensível à localidade de dados e com uma política de escalonamento que sempre posiciona o processamento em um nó diferente daquele que contém os dados. Pode-se ver que no primeiro caso não há tráfego de rede significativo, enquanto no segundo há sempre um volume de tráfego enviado semelhante ao padrão de leitura local.

4.2 HDFS

Para avaliar o comportamento do escalonamento sensível à localidade dos dados, executamos o programa de leitura de arquivos em diversas situações e medimos o tempo de execução total. Dois elementos importantes na análise foram o tamanho do cluster considerado e a carga de dados processada por cada máquina. O tamanho do arquivo usado era determinado em termos do tamanho do cluster e da carga por nó, então experimentos com maior processamento por máquina liam arquivos proporcionalmente maiores. Todos os arquivos foram gerados com fator de replicação 2.

Nos testes utilizando o HDFS, para que a cache do sistema de arquivos do sistema operacional não interferisse nos experimentos, efetuamos a limpeza de cache antes de cada execução. Nos testes usando o Tachyon, como seu princípio de operação é exatamente o de tentar se aproveitar ao máximo de todas as possibilidades de cacheamento dos dados, sua cache era limpa apenas antes do início de cada experimento. A aplicação era executada uma primeira vez para carregar a cache do Tachyon e depois era medido o conjunto de cinco execuções.



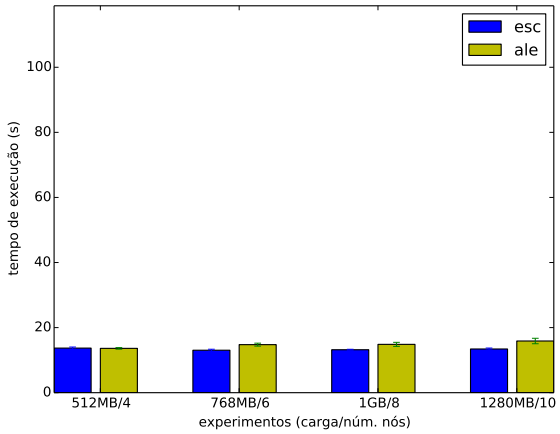
(a) Execução com o escalonador proposto (b) Execução sem localidade

Figura 4.1: Padrões de acesso a disco e de dados enviados e recebidos pela rede para uma execução de uma aplicação com quatro nós de processamento, com o escalonador sensível à localidade dos dados (a) e com uma política de escalonamento que sempre posiciona o processamento em um nó diferente daquele que contém os dados (b). Pode-se ver que no primeiro caso não há tráfego de rede significativo, enquanto no segundo há sempre um volume de tráfego enviado semelhante ao padrão de leitura local.

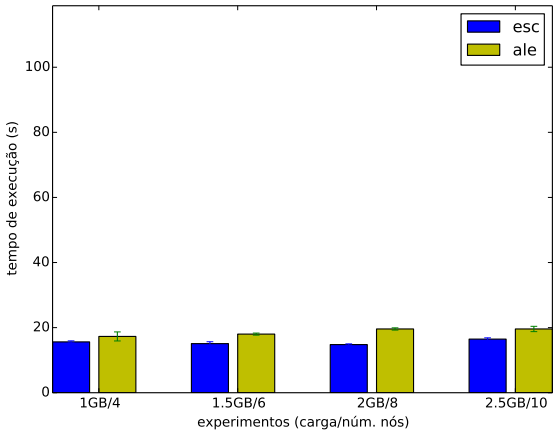
A figura 4.2 mostra o efeito da variação de carga por máquina para *clusters* configurados com diversos números de máquinas. Tempos de execução para *clusters* maiores são sempre superiores, pois há mais overhead na inicialização e na sincronização entre os processos na aplicação distribuída. Claramente, o escalonador proposto apresenta bom desempenho em todos os casos.

Vale aqui ressaltar que juntamente com o tempo de execução da aplicação, existe um custo constante presente em todas as execuções relacionado à inicialização do ambiente. Este custo tem como principais fatores a instanciação dos filtros e a comunicação realizada entre o escalonador e o sistema de arquivos distribuído para obtenção da localidade dos dados.

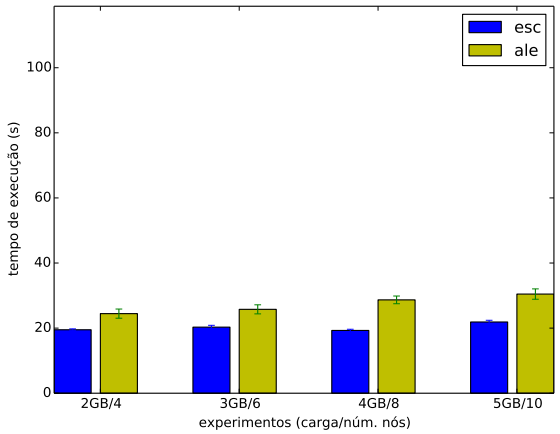
Para os clusteres menores (4 e 6 máquinas), praticamente não há diferença entre



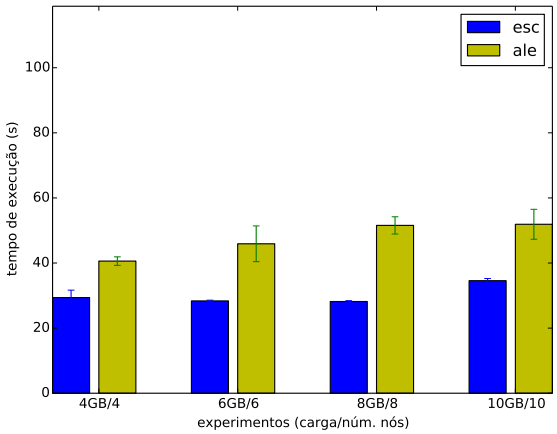
(a) Dados por nó: 128MB



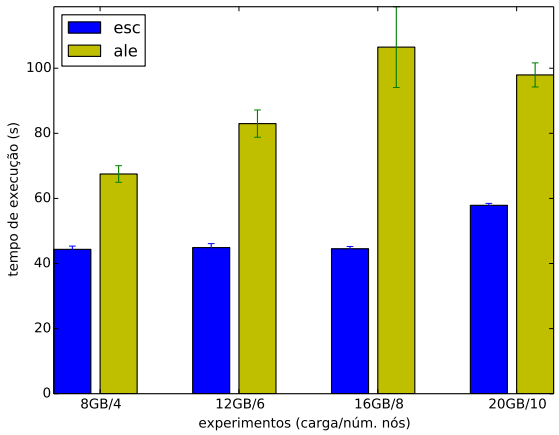
(b) Dados por nó: 256MB



(c) Dados por nó: 512MB



(d) Dados por nó: 1GB



(e) Dados por nó: 2GB

Figura 4.2: Resultados experimentais utilizando o escalonador integrado ao HDFS para diferentes números de máquinas, agrupado pelo volume de dados por máquina.

os tempos do escalonador baseado em localidade e do escalonador aleatório, exceto para blocos maiores (1GB e 2GB). Observando os logs da execução, um dos motivos é que,

com 4 máquinas, por exemplo, e fator de replicação 2, o escalonador aleatório tinha pelo menos 50% de chance de alocar um processador junto ao dado que ele iria processar. Já para o cluster de 8 máquinas, por exemplo, a chance de um processo ser escalonado aleatoriamente em um nó com o dado associado a ele cai para 25% e o escalonador sensível à localidade dos dados passa a ter um desempenho sensivelmente melhor, o que comprova que executar o processamento inicial dos dados na mesma máquina que os contém ainda é vantajoso em relação ao custo de acesso pela rede. Além disso, mesmo para o cluster menor, no caso de blocos de leitura/arquivos maiores, o ganho de processar os dados localmente também é visível.

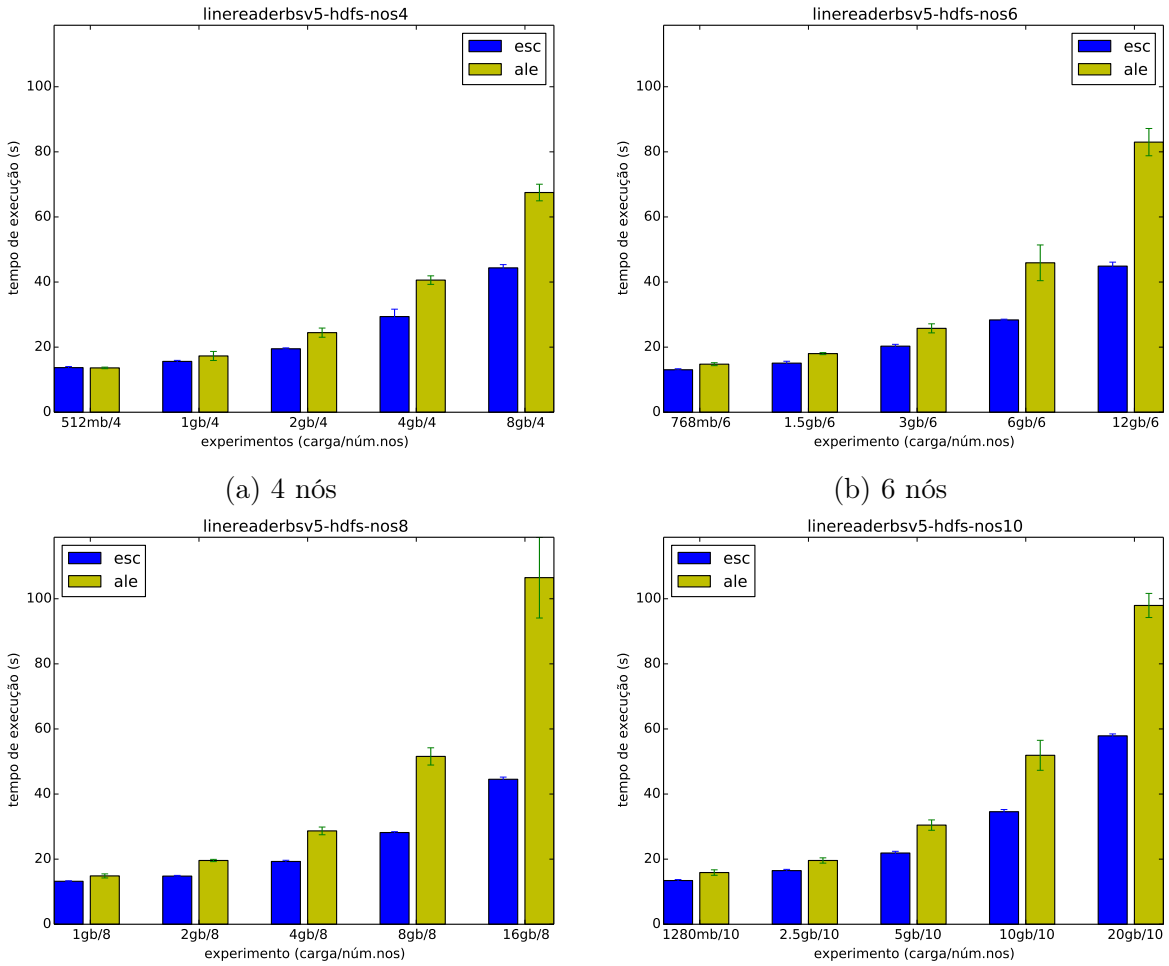
Já nos gráficos da figura 4.3, podemos observar o impacto da carga sobre as diversas configurações de clusters. Como seria esperado, observamos que o tempo de execução da aplicação aumenta enquanto a carga de dados processados cresce, pois temos um tempo de processamento maior por cada nó. Outro ponto demonstrado nos gráficos é que, com o aumento do tamanho do arquivo a ser processado, o escalonador baseado em localidade apresenta resultados mais expressivos na diminuição do tempo de execução das aplicações.

Isso se dá pois aumenta-se a fração de tempo de processamento de dados em relação ao restante da aplicação, logo os ganhos com localidade de dados se mostram ainda mais significativos.

4.3 Tachyon

Recentemente, diversos trabalhos têm sugerido a adoção de mecanismos de armazenamento em memória, como o projeto RamCloud [Ousterhout et al., 2010]. Nesse caso, o tempo de acesso e as taxas de transmissão de dados seriam aqueles da memória e não do disco, o que pode alterar significativamente o desempenho do sistema. Considerando esse cenário, o escalonador também foi configurado para trabalhar com o Tachyon, já que esse oferece o recurso de manter em memória arquivos já acessados. Como mencionado anteriormente, para os experimentos, executamos a aplicação uma primeira vez, causando a carga do arquivo para a memória do Tachyon, e medíamos os tempos de cinco execuções depois que os dados já se encontravam na memória de algum nó. Devido ao comportamento do Tachyon e da aplicação, não havia replicação de blocos em memória: cada bloco era lido por apenas um processo parte da aplicação, então o dado era carregado na memória de apenas uma máquina de cada vez.

A figura 4.4 mostra os resultados nesse caso (não foi utilizado os tamanhos de blocos de 1GB e 2GB devido a limitações do Tachyon na configuração usada nos clusters



(a) 4 nós

(b) 6 nós

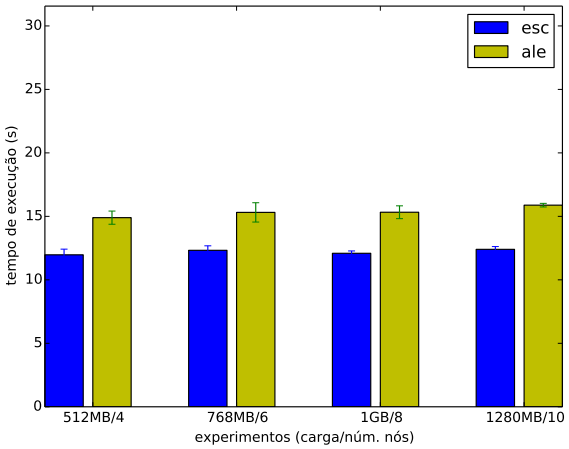
(c) 8 nós

(d) 10 nós

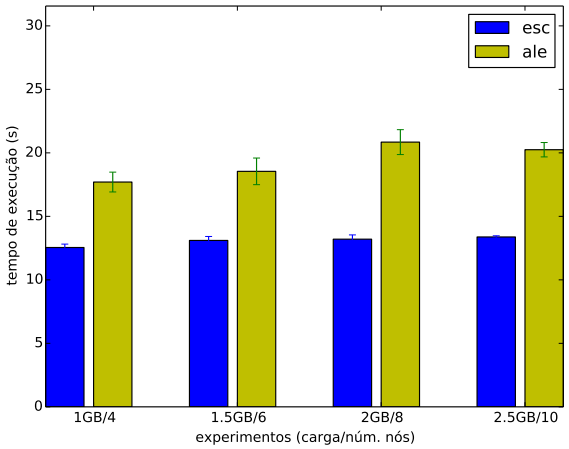
Figura 4.3: Resultados experimentais utilizando o escalonador integrado ao HDFS para diferentes tamanhos de arquivos com diferentes números de máquinas.

do experimento). Claramente, os ganhos do escalonador baseado em localidade são ainda mais significativos quando os arquivos já se encontram em memória. Já que não há replicação de blocos nos dados, o escalonador aleatório já não é capaz de obter o mesmo desempenho na maior parte dos casos. Além disso, o fato dos dados já estarem na memória do nó local torna o tempo de execução bem mais regular (e menor) para o escalonador proposto. Apesar do aumento do número de máquinas do cluster ainda ter um impacto, os tempos de execução ainda caem significativamente devido às taxas de transferências mais altas. Por fim, assim como nos experimentos utilizando HDFS, podemos observar que o ganho do escalonador proposto com relação ao aleatório aumenta à medida em que a carga de processamento cresce.

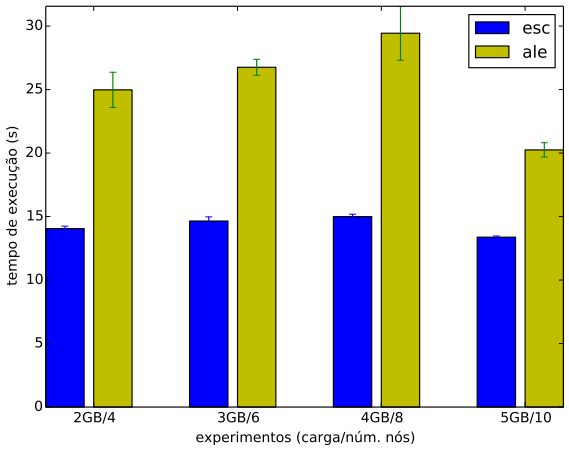
Também para o Tachyon, apresentamos os experimentos organizados com foco no impacto da carga sobre as diversas configurações de clusters, presentes na Figura 4.5.



(a) Dados por nó: 128MB



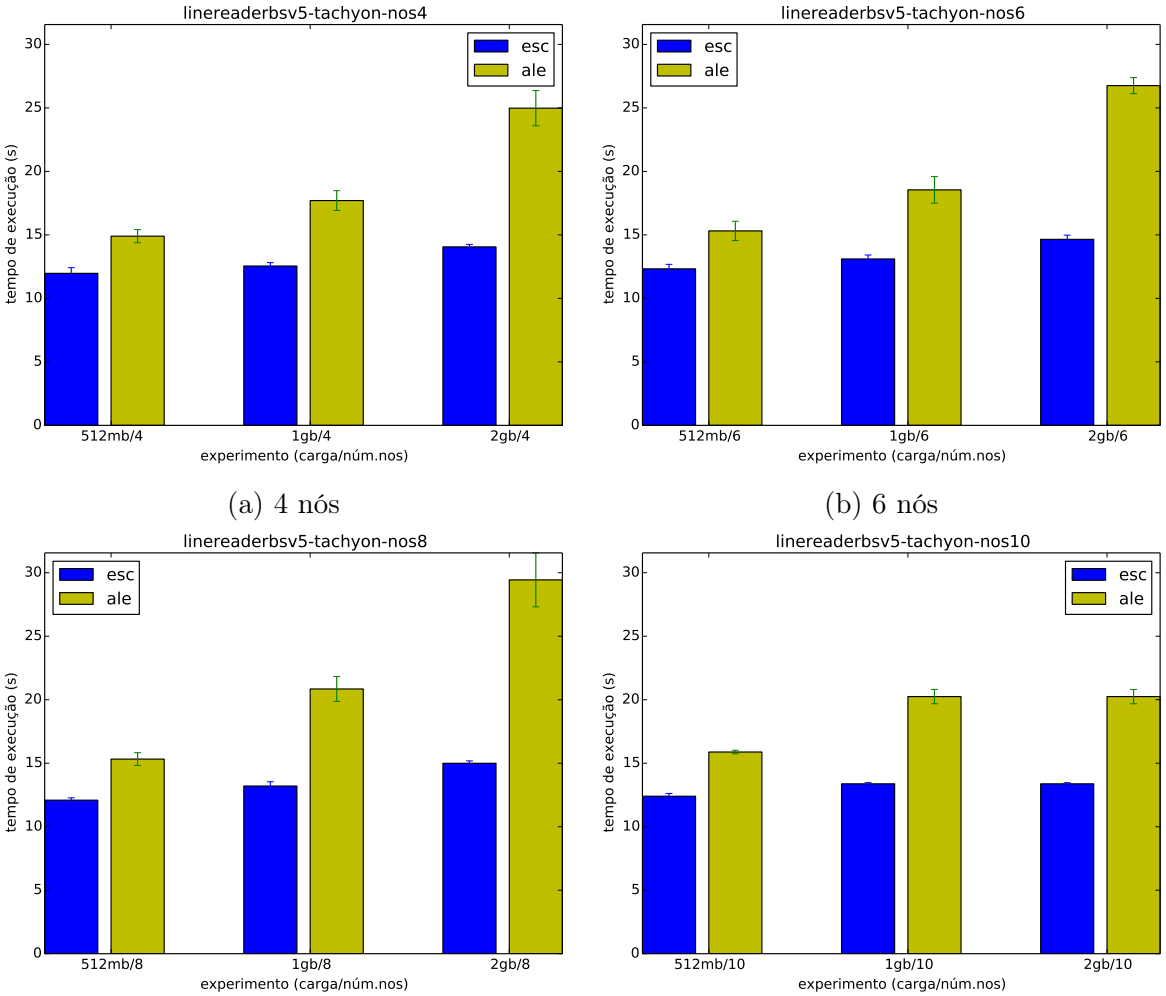
(b) Dados por nó: 256MB



(c) Dados por nó: 512MB

Figura 4.4: Resultados experimentais utilizando o escalonador integrado ao Tachyon para diferentes números de máquinas, fixando o volume de dados acessado por cada nó.

E pudemos observar que, também, o tempo de execução da aplicação cresce a medida que a carga de dados processados aumenta, pois temos um tempo de processamento maior por cada nó. Porém, no caso dos dados já estarem em memória, a execução com o escalonador proposto possui um crescimento mais tímido a medida em que os dados crescem, pois a velocidade de leitura em memória é superior e o aumento da carga não é expressivo. Também é observado que o escalonador aleatório se degrada bastante com o aumento da carga, tornando os ganhos do escalonador proposto ainda mais significativos quando trabalham com arquivos maiores.



(a) 4 nós

(b) 6 nós

(c) 8 nós

(d) 10 nós

Figura 4.5: Resultados experimentais utilizando o escalonador integrado ao Tachyon para diferentes tamanhos de arquivos, com diferentes números de máquinas.

Capítulo 5

Conclusão e trabalhos futuros

Neste trabalho avaliamos o impacto de uma política de alocação de recursos sensível à localidade dos dados em termos de máquinas. O argumento nesse caso é que, pelo menos para a tecnologia de Ethernet Gigabit atualmente comum na maioria dos datacenters e clusters, processar dados em uma máquina que não aquela que já os mantém gera um overhead de comunicação pela rede, mesmo que não haja gargalos entre fonte e destino. Para avaliar esse argumento, desenvolvemos um escalonador sensível à localidade dos dados integrado ao HDFS e ao YARN durante a reengenharia do ambiente de processamento Watershed.

Os resultados comprovam que o escalonamento sensível à localidade beneficia o desempenho das aplicações na medida em que o tamanho do cluster cresce assim como aumenta-se a carga de trabalho. O que notamos é que nosso escalonador não prejudica o desempenho do sistema em clusters pequenos e com blocos menores e beneficia significativamente a execução em clusters maiores e quando os blocos de dados são maiores. Além disso, os ganhos se tornam ainda mais significativos quando os dados já estão em memória, como no caso do Tachyon. Isso pode ser um resultado ainda mais importante considerando-se propostas recentes de sistemas de armazenamento baseados em memória.

Para avaliação de desempenho, executamos os testes com uma carga de trabalho relativamente pequena para o cenário de processamento massivo de dados. Por esse motivo, resolvemos realizar os experimentos utilizando um programa de leitura de arquivos simples de modo a evidenciar o impacto da localidade de dados na execução dos testes. Caso contrário, o ônus do acesso aos dados se tornaria mínimo se comparado ao custo de processamento de uma aplicação de maior complexidade, pois o ganho com a localidade de dados se diluiria entre o tempo de execução total da aplicação.

Como trabalhos futuros ficará a realização de experimentos com cargas de tra-

balhos significativamente maiores, assim como a utilização de aplicações com maior complexidade de processamento.

Demais sistemas de processamento distribuído, como Hadoop e Spark, possuem seus próprios escalonadores e tem suas próprias políticas para lidar com localidade de dados. Nosso intuito com este trabalho não foi de criar escalonadores melhores para tais ferramentas, mas sim implementar um escalonador para o ambiente Watershed e então utilizá-lo para analizar o impacto da localidade de dados nesta ferramenta. Todavia, o escalonador proposto é facilmente extensível e pode ser integrado a outros ambientes de processamento que usam o ecossistema Hadoop.

Logo, seria possível realizar a inclusão do mesmo no Hadoop MapReduce. Para isso é necessário inserir a chamada do escalonador no código JobMaster da aplicação. O que ocorrerá de forma natural, uma vez que o MapReduce é implementado em cima da plataforma YARN e utiliza HDFS para armazenamento de dados. Outro ambiente de processamento que utiliza a dupla YARN/HDFS é o Spark. Também sendo possível realizar a inclusão do escalonador descrito neste trabalho e avaliarmos seu impacto.

Outro ponto importante de ser analisado futuramente, é a extensão da nossa política de escalonamento para incluir heurísticas que possam melhorar seu desempenho. Como políticas que levem em consideração a carga de trabalho já presente no cluster, ou mesmo que procure balancear o disparo das instâncias da aplicação de modo a não sobrecarregar alguma porção das máquinas. Também pretendemos tratar os cenários multiusuários e com maior contenção de recursos, incluindo heurísticas que levem em consideração o uso concomitante dos recursos do cluster. Uma direção para a criação de políticas mais sofisticadas seria utilizar técnicas de escalonamento baseadas em modelos de otimização, assim como a realizada no trabalho de Nascimento et al. [2005].

Referências Bibliográficas

Ananthanarayanan, G.; Ghodsi, A.; Shenker, S. & Stoica, I. (2011). Disk-locality in datacenter computing considered irrelevant. Em *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, pp. 1–5, Berkeley, CA, USA. USENIX Association.

Barroso, L. A. & Hölzle, U. (2009). The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 4(1):1–108.

Dean, J. & Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Ferreira, R. A.; Meira, W.; Guedes, D.; Drummond, L. M. d. A.; Coutinho, B.; Teodoro, G.; Tavares, T.; Araujo, R. & Ferreira, G. T. (2005). Anthill: A scalable run-time environment for data mining applications. Em *Computer Architecture and High Performance Computing, 2005. SBAC-PAD 2005. 17th International Symposium on*, pp. 159–166. IEEE.

Ghemawat, S.; Gobioff, H. & Leung, S.-T. (2003). The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43. ISSN 0163-5980.

Hindman, B.; Konwinski, A.; Zaharia, M.; Ghodsi, A.; Joseph, A. D.; Katz, R.; Shenker, S. & Stoica, I. (2011). Mesos: A platform for fine-grained resource sharing in the data center. Em *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pp. 295–308, Berkeley, CA, USA. USENIX Association.

Hott, B.; Rocha, R. C. & Guedes, D. (2016). Escalonamento de processos sensível à localidade de dados em sistemas de arquivos distribuídos. Em *Anais do Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, SBRC 2016, pp. 1–14, Salvador. SBC.

Hunt, P.; Konar, M.; Junqueira, F. P. & Reed, B. (2010). Zookeeper: wait-free coordination for internet-scale systems. Em *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8.

Isard, M.; Budiu, M.; Yu, Y.; Birrell, A. & Fetterly, D. (2007). Dryad: Distributed data-parallel programs from sequential building blocks. Em *Proceedings of the 2007 Eurosys Conference*, Lisbon, Portugal. Association for Computing Machinery, Inc.

Isard, M.; Prabhakaran, V.; Currey, J.; Wieder, U.; Talwar, K. & Goldberg, A. (2009). Quincy: fair scheduling for distributed computing clusters. Em *22nd symposium on Operating systems principles on Proceedings of the ACM SIGOPS*, pp. 261–276. ACM.

Li, H.; Ghodsi, A.; Zaharia, M.; Shenker, S. & Stoica, I. (2014). Tachyon: Reliable, memory speed storage for cluster computing frameworks. Em *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pp. 6:1–6:15, New York, NY, USA. ACM.

Nascimento, L. T.; Ferreira, R. A.; Meira, W. & Guedes, D. (2005). Scheduling data flow applications using linear programming. Em *2005 International Conference on Parallel Processing (ICPP'05)*, pp. 638--645. IEEE.

Ousterhout, J.; Agrawal, P.; Erickson, D.; Kozyrakis, C.; Leverich, J.; Mazières, D.; Mitra, S.; Narayanan, A.; Parulkar, G.; Rosenblum, M.; Rumble, S. M.; Stratmann, E. & Stutsman, R. (2010). The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105. ISSN 0163-5980.

Ramos, T.; Silva, R.; Carvalho, A. P.; Ferreira, R. A. C. & Meira, W. (2011). Watershed: A high performance distributed stream processing system. Em *23rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2011*, pp. 191–198. IEEE.

Rocha, R.; Hott, B.; Dias, V.; Ferreira, R.; Meira, W. & Guedes, D. (2016). Watershedng: an extensible distributed stream processing framework. *Concurrency and Computation: Practice and Experience*. ISSN 1532-0634.

Schwarzkopf, M.; Konwinski, A.; Abd-El-Malek, M. & Wilkes, J. (2013). Omega: flexible, scalable schedulers for large compute clusters. Em *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 351–364. ACM.

Shvachko, K.; Kuang, H.; Radia, S. & Chansler, R. (2010). The hadoop distributed file system. Em *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pp. 1–10, Washington, DC, USA. IEEE Computer Society.

Vavilapalli, V. K.; Murthy, A. C.; Douglas, C.; Agarwal, S.; Konar, M.; Evans, R.; Graves, T.; Lowe, J.; Shah, H.; Seth, S.; Saha, B.; Curino, C.; O'Malley, O.; Radia, S.; Reed, B. & Baldeschwieler, E. (2013). Apache hadoop YARN: Yet another resource negotiator. Em *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pp. 5:1–5:16, New York, NY, USA. ACM.

White, T. (2009). *Hadoop: the definitive guide: the definitive guide*. O'Reilly Media, Inc.

Yao, Y.; Wang, J.; Sheng, B.; Lin, J. & Mi, N. (2014). Haste: Hadoop yarn scheduling based on task-dependency and resource-demand. Em *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pp. 184–191. IEEE.

Zaharia, M.; Borthakur, D.; Sen Sarma, J.; Elmeleegy, K.; Shenker, S. & Stoica, I. (2010a). Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. Em *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pp. 265–278, New York, NY, USA. ACM.

Zaharia, M.; Chowdhury, M.; Franklin, M. J.; Shenker, S. & Stoica, I. (2010b). Spark: cluster computing with working sets. Em *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pp. 10–10.