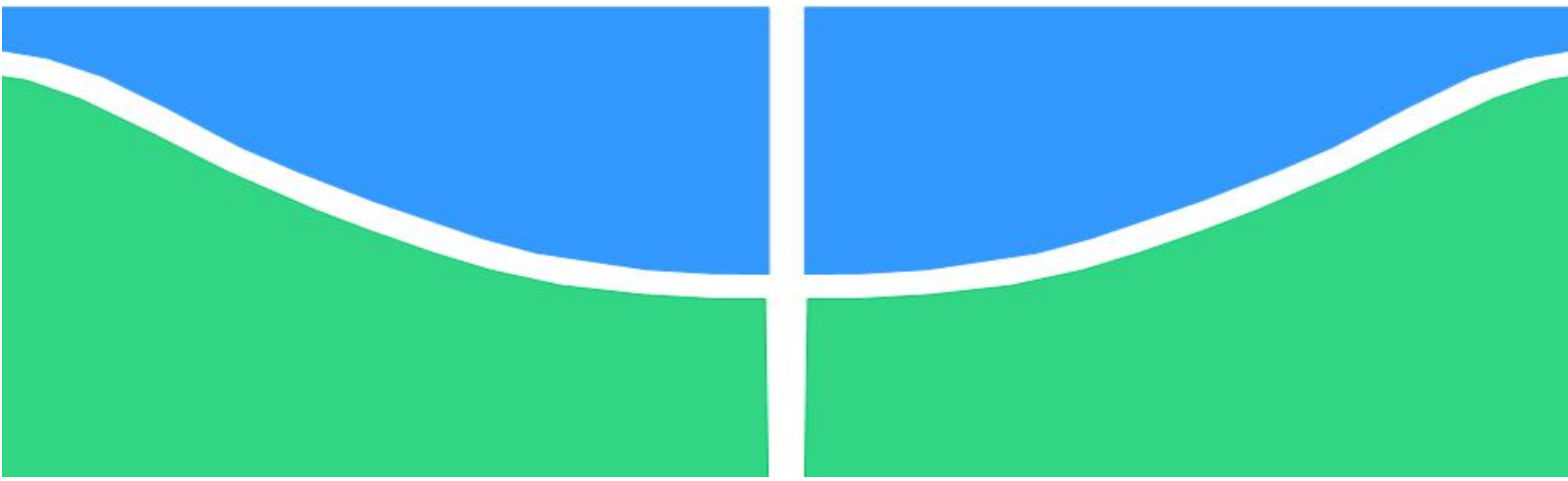


Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Fundamentos de Sistemas Distribuídos

Relatório de Experimentação em Hadoop

Phelipe Wener (12/0132893)
Alexandre Torres(13/0099767)

Professor:
Fernando W. Cruz

Brasília, DF
06 de Julho de 2017

Introdução

Dentro do curso de sistemas distribuídos, foram estudadas diversas tecnologias para distribuição de tarefas computacionais: Sockets, MPI, Message-Queue e Web Services. Dentro desses modelos podemos perceber várias formas de se resolver problemas de comunicação, desempenho, ociosidade e facilidade de implementação. O Hadoop resolve de forma elegante e poderosa o problema de *Bag Of Task* o qual essas outras arquiteturas também tem a possibilidade de resolver.

De uma constante necessidade do Google em manter uma nuvem que precisava lidar com BigData, nasce uma percepção que daria a luz ao Hadoop. Precisamente, é fácil ver que os dois módulos centrais do Hadoop (Sistemas de Arquivo Distribuído e MapReduce) podem ter nascido de problemas diferentes relacionados ao BigData, tal que executando os mesmo são ferramentas poderosas na sua forma independente, juntos, devem compor a base do que há de mais completo em sistemas distribuídos.

No andamento deste relatório, pretende-se explorar questões de configuração, performance, escalabilidade, entre outros requisitos desejaveis para um sistema distribuído usando Hadoop. Almeja-se que ao final deste, seja possível ter uma visão consistente do Hadoop, seus módulos e suas aplicações.

Hadoop Distributed File System (HDFS)

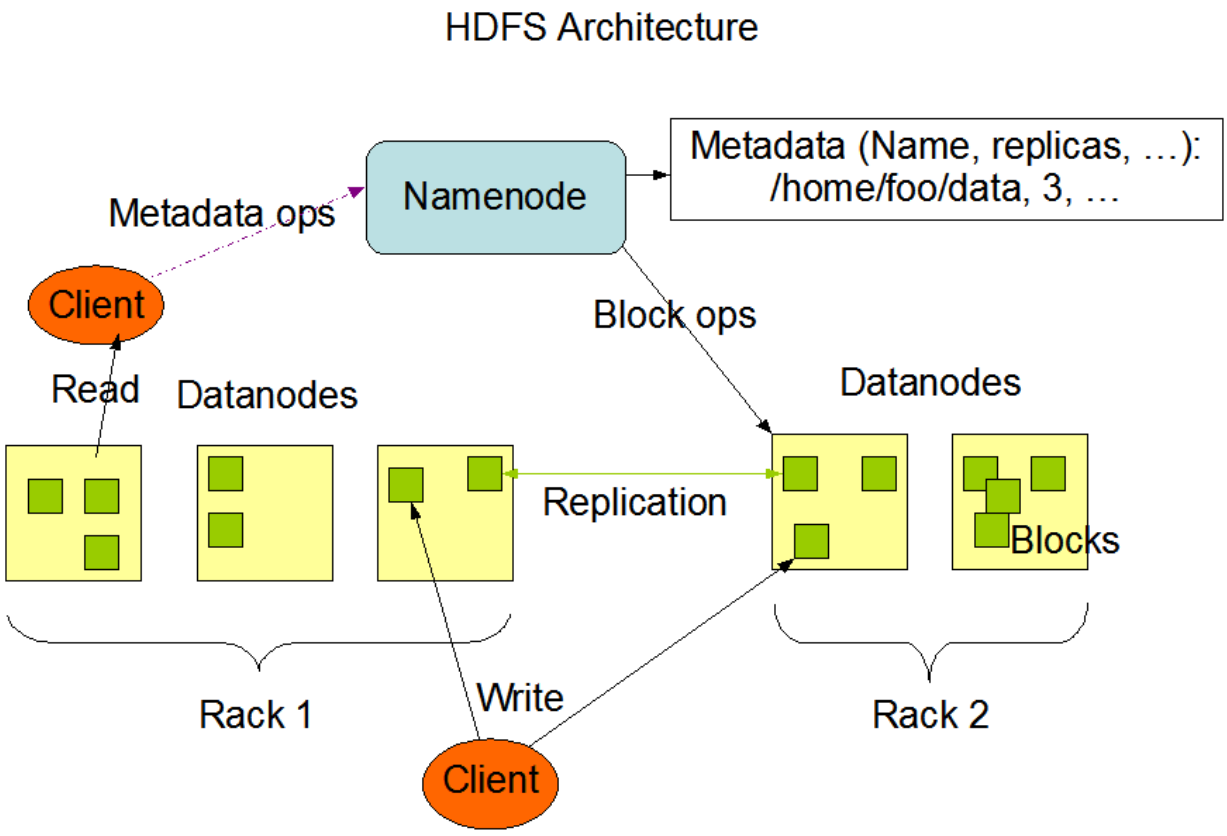
O Hadoop é ideal para armazenar grandes quantidades de dados, do porte de terabytes e pentabytes, e usa o HDFS como sistema de armazenamento. O HDFS permite a conexão de nós (computadores pessoais padrão) contidos nos clusters por meio dos quais os arquivos de dados são distribuídos [1].

O HDFS ao lado do MapReduce é a base do Hadoop. Sem ele seria impossível trabalhar com grandes quantidades de dados de forma consistente, devido aos diversos contratempos da computação distribuída, como o mal funcionamento dos nós, rendimento de processamento IO de grandes conjuntos de dados, bem como sincronismos de gravação.

O HDFS é baseado numa arquitetura de Master/Slave, que são chamados por *NameNode* e *DataNode* respectivamente. O primeiro é resposável por regular acesso aos arquivos pelos clientes, bem como manter a árvore de diretórios em todo cluster, percebe-se assim como o *NameNode* é crítico. Em um cluster, pode-se ter as mais diversificadas configurações: único *NameNode*, replicado(por segurança) e até independentes. Até o esquema independente pode haver compartilhamento de *DataNodes*, que será explicado a seguir.

Os *DataNodes* são os nós que armazenam conjuntos de blocos de arquivos criados para distribuição do sistema, costumam ser blocos de tamanho fixo. Não são críticos, caso caia o *NameNode* irá reparar a perda balanceando os dados entre os *DataNodes* restantes. Esses duas partes de Software, *NameNode* e *DataNode* estão continuamente conversando para realização de qualquer ação necessária. Um bom artigo que explica melhor as caracteristicas dos dois: [2]

Para configuração de *DataNode* e *NameNode* tem-se o arquivo `hdfs-site.xml` onde tem-se uma ampla variedade de configurações [3].



Vantagens

- Redundância de dados
  - Você pode escolher o nível de redundância e várias configurações acerca disso.
  - Redistribuição de carga de dados dinâmica, se um nó cai, outros assumem como se ele não tivesse existido
- Tolerância a falhas
  - Muito eficiente em lidar com as falhas dos DataNodes de forma automática
- Portabilidade de sistemas
  - Foi possível operar em diferentes versões do SO sem problemas.
- Interface para operar diretamente nesse sistema de arquivos
  - Qualquer nó pode interagir com o HDFS
- Suporte para acompanhamento do ecosistema de dados
  - Interface web para acompanhar o status do cluster
  - Comandos terminal para acompanhar o status do cluster

Desvantagens

- Falta de transparência quanto ao processamento dos dados.
  - Não parece ser possível acompanhar em tempo real o direcionamento dos dados
- Apesar de poder ter redundância, o NameNode é uma parte crítica, o HDFS depende dele.
  - Segundo [2], o hdfs não pode funcionar sem um NameNode.
- Atraso ou deficiência de alerta no caso de queda dos DataNodes.
  - Quando um DataNode cai, o Hadoop não emite nenhum alerta. Após demasiado tempo ele notifica que o Nó está morto.

Onde usar?

O HDFS pode ser usado como módulo independente. Uma situação bem típica é o armazenamento de dados em sistemas de grandes volumes de arquivos de mídia como provavelmente é feito no Youtube, Google Drive e Instagram, tal que os arquivos dos usuários estejam sempre disponíveis e integros.

Qualquer aplicação que precise salvar imagens, vídeos ou qualquer tipo arquivo, é recomendado suporte a um sistema de arquivos distribuído, uma vez que os bancos relacionais tem dificuldades em persistir esses dados.

Usando o HDFS

O HDFS pode ser usado em qualquer instância do hadoop, ou seja, em qualquer *Node* ativo, por exemplo, é possível usar dentro de um Data Node os comandos do fs, por exemplo:

```
$ hadoop fs -ls /
Found 3 items
drwxr-xr-x   - fsdadmin supergroup          0 2017-07-06 18:12 /input
drwxr-xr-x   - fsdadmin supergroup          0 2017-07-07 15:04 /output_2
drwx------   - fsdadmin supergroup          0 2017-07-05 11:10 /tmp
kuwener@wenerianus:/usr/hadoop-2.8.0$ hadoop fs -mkdir /folder_test
kuwener@wenerianus:/usr/hadoop-2.8.0$ hadoop fs -ls /
Found 4 items
drwxr-xr-x   - kuwener  supergroup          0 2017-07-07 16:17 /folder_test
drwxr-xr-x   - fsdadmin supergroup          0 2017-07-06 18:12 /input
drwxr-xr-x   - fsdadmin supergroup          0 2017-07-07 15:04 /output_2
drwx------   - fsdadmin supergroup          0 2017-07-05 11:10 /tmp
```

Executando o comando `$ hadoop fs` é possivel ver uma série de opções para operar dentro do HDFS.

Referências

[1] https://www.ibm.com/developerworks/br/library/wa-introhdfs/index.html

[2] http://hadoopinrealworld.com/namenode-and-datanode/

MapReduce

O Core do Hadoop em suas primeiras versões consistia basicamente de dois "subsistemas":

- HDFS: Usado para armazenar os dados
- MapReduce: Usado para Processar os dados

Apesar de não ser mais necessário utilizar o MapReduce a partir da versão 2.0 do Hadoop, devido a introdução do YARN, ele ainda é bastante utilizado, e descreveremos seu funcionamento neste relatório.

Contexto

Um dos primeiros obstáculos enfrentados pelo Google foi encontrar alguma forma para indexar um grande volume de dados da web que estava crescendo de forma exponencial. Para resolver esse problema, o Google inventou um novo estilo de processamento de dados chamado de MapReduce, isso permitiu que grande quantidades de dados fossem processados em um grande cluster composto por vários servidores. MapReduce é um modelo, um paradigma de programação com uma implementação associada que facilita paralelismo durante processamento dos dados em grandes clusters, com uma grande quantidade de máquinas.

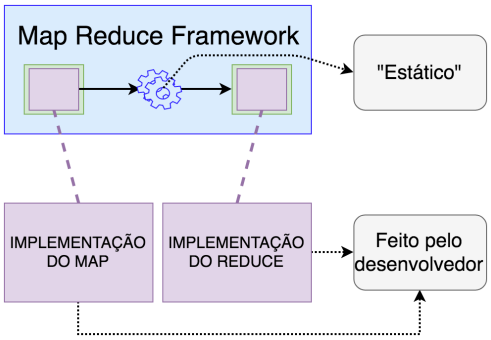
O Google, em 2004, lançou um artigo descrevendo o MapReduce. Este papel em conjunto com o artigo que descrevia o Google File System, foram utilizados como base para a criação do Hadoop.

Como é utilizado pelo programador.

MapReduce é um modelo, um paradigma de programação em que desenvolvedor se preocupa somente em descrever duas funções, o Map e Reduce.

Essas duas funções são executadas por um Framework que realiza passos intermediários e é responsável por distribuir o processamento entre as máquinas do cluster.

Para facilitar a explicação de como funciona o MapReduce foi criada a seguinte imagem abaixo.



Conforme pode ser visto na imagem, o desenvolvedor se preocupa basicamente em implementar a função Map e a função Reduce, estas funções podem ser vistas como pontos extensíveis do framework. Partes do core do framework também pode ser modificadas através de arquivos de configuração, porém ele é bem mais estático.

Quando utilizar MapReduce

Obviamente, só faz sentido utilizar Hadoop e MapReduce para contextos de Big Data, ou seja, quando uma grande quantidade de dados precisa ser processada.

De forma resumida, MapReduce é utilizado em situações em que a entrada, após ser processada, pode ser descrita como um par Chave-Valor e, posteriormente, essa o conjunto desses pares puderem ser processados pelos Reducers a fim de gerar um resultado. Abaixo será descrito um exemplo a fim de exemplificar como esse modelo de programação funciona.

Exemplo de MapReduce

Para explicar o funcionamento do MapReduce é interessante utilizar um exemplo. Abaixo será detalhado um cenário em que o MapReduce seria bem aplicado.

Cenário do exemplo

Suponha o seguinte cenário:

- Você foi contratado por uma grande empresa internacional com uma grande quantidade de lojas em diversos locais e cada uma realiza muitas vendas.
- Esta empresa tem um arquivo contendo todas as vendas do ano de 2016. Esse arquivo possui o seguinte formato:

```
Data da compra | Localização da loja | O que foi vendido | Preço

2016-01-01  London    Clothes   19.90
2016-01-01  Miami     Music     14.90
2016-01-01  Brasilia  Toys      29.90
2016-01-01  NYC       Music     12.90
2016-01-01  Miami     Clothes   14.90
...
```

- Dado esse arquivo com inúmeras linhas, o seu trabalho é extrair informações desse dado como por exemplo:

```
Qual foi o faturamento em cada loja?
```

Descrição da solução

Uma forma de resolver esse problema seria utilizando um par Chave-Valor, onde a chave seria a localização da loja e o valor o preço do produto vendido. Dessa forma, quando a chave repete, o valor é atualizado.

Então ao processar a primeira linha, obteríamos a seguinte tabela:

Chave	Valor
London	19.90

Ao processar as quatro primeiras linha, obteríamos a seguinte tabela:

Chave	Valor
London	19.90
Miami	14.90
Brasilia	29.90
NYC	12.90

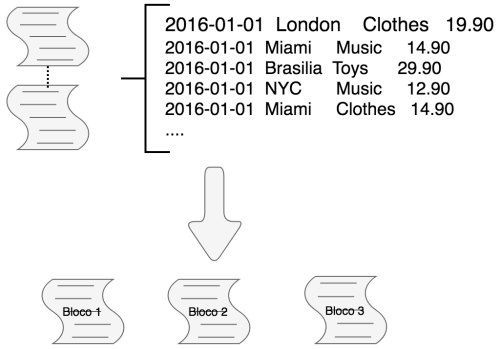
Ao processar a última linha, obteríamos a seguinte tabela:

Chave	Valor
London	19.90
Miami	29.80
Brasilia	29.90
NYC	12.90

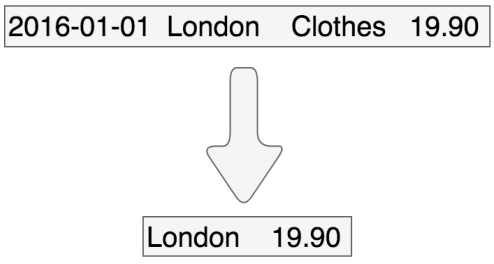
Observe que foi atualizado o valor da Chave "Miami".

Caso uma grande quantidade de dados estivesse sendo processada por um programa convencional, provavelmente, demoraria muito tempo para obtermos e resultado e haveria um estouro de memória. Nesse contexto poderíamos aplicar o paradigma MapReduce.

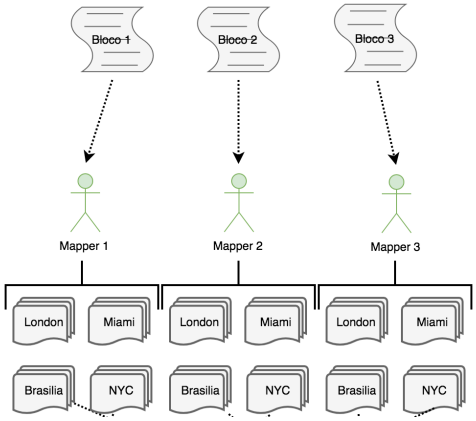
Primeiramente, o arquivo seria quebrado em blocos menores (isso é feito pelo próprio HDFS). Essa etapa pode ser observada na imagem abaixo.



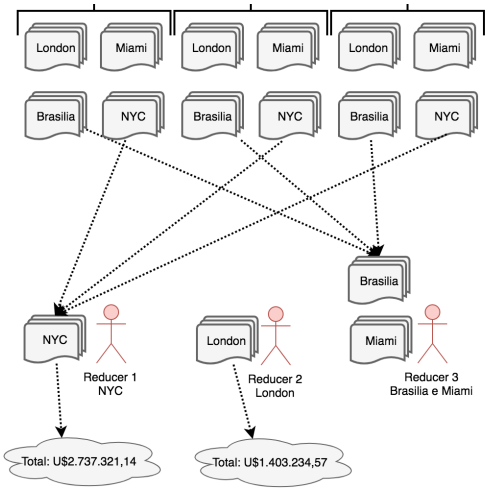
Posteriormente, cada Mapper (quem executa a função Map), iria ler cada linha do arquivo e gerar uma saída indicando qual a loja e o valor.



O Framework MapReduce irá ordenar o output dos Mappers antes deles serem utilizados como input para os Reducers. Ou seja, agrupará, todos os dados de acordo com as chaves. O resultado pode ser visto abaixo.



Agora que todos os Dados estão agrupados de acordo com as chaves, o papel dos Reducers é somar os valores relacionados a cada par e gerar a receita total da loja de cada região.



Map

Cada Mapper trata de uma parte dos dados e eles trabalham em paralelo. O output gerado por eles é chamado de "Intermediate Records", e este resultado é da forma Chave-Valor (no exemplo acima, a chave é a localização da loja e o valor o preço da venda).

A quantidade de Mappers está relacionada a quantidade de "input splits", que é um conjunto de "records" e normalmente tem o tamanho aproximado de um bloco (mas não exatamente), visto que um único "record" não pode ser quebrado e armazenado em blocos diferentes.

Exemplo de função Map em python

Normalmente e por padrão, os códigos para o Hadoop são escritos em Java. Porém, o Hadoop Streaming, permite que sejam utilizadas outras linguagens, nesse caso, utilizaremos python.

Exemplo de linha da entrada:

```
2016-01-01  London    Clothes   19.90
```

Baseado no formato das entradas demostrado abaixo, foi produzido o seguinte código para o Mapper.

Código:

```
def mapper():
  for line in sys.stdin:
    data = line.strip().split("\t")

    if len(data) == 4:
      date, store, item, cost = data
      print "{0}\t{1}".format(store, cost)
```

Shuffle and Sort (Simplificado)

Após executar a função Map, o framework executa as funções Shuffle e Sort. De forma resumida, a função Shuffle está relacionado ao movimento dos "Intermediate Records" dos Mappers para os Reducers. A função Sort está relacionada ao fato de que estes dados serão organizados de forma ordenada Este tópico será melhor detalhado a seguir no relatório.

Reduce

Finalmente, cada Reducer trabalha em um conjunto de Registros de cada vez. Ele recebe a chave e um conjunto de valores relacionados aquela chave, processa esses dados de alguma forma (nesse exemplo, somamos todos os valores), depois ele escreve o resultado. É interessante notar que, caso somente um Reducer esteja trabalhando, o resultado obtido provavelmente estará organizado de forma ordenada.

A quantidade de Reducers pode ser determinada pelo usuário. É possível, inclusive, não possuir nenhum Reducer para realizar determinada tarefa.

Exemplo de função Reduce em python

Normalmente e por padrão, os códigos para o Hadoop são escritos em Java. Porém, o Hadoop Streaming, permite que sejam utilizadas outras linguagens, nesse caso, utilizaremos python. O código Mapper pode ser visto abaixo.

Exemplo de linha da entrada:

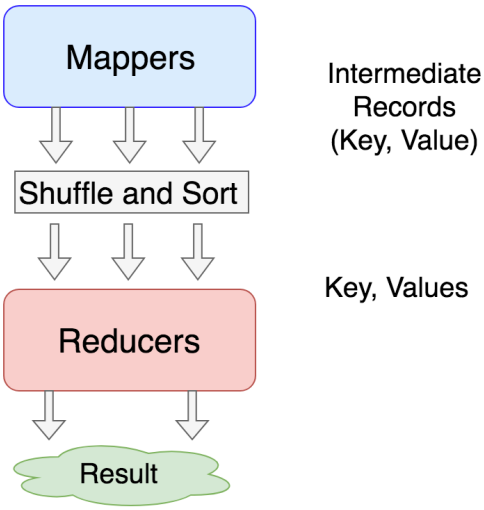
```
London    19.90
```

Baseado no formato das entradas demostrado abaixo, foi produzido o seguinte código para o Reducer.

Código:

```
def reducer():

  salesTotal = 0
  oldKey = None

  for line in sys.stdin:
    data = line.strip().split("\t")

    if len(data) != 2:
      continue

    thisKey, thisSale = data

    if oldKey and oldKey != thisKey:
      print "{0}\t{1}".format(oldKey, salesTotal)

      salesTotal = 0

    oldKey = thisKey
    salesTotal += float(thisSale)

  if oldKey != None:
    print "{0}\t{1}".format(oldKey, salesTotal)
```

Abaixo está uma figura exemplificando o processo.



Rodar em Python

```
hadoop jar <path to hadoop streaming .jar> -mapper <mapper.py / mapper code> -reducer <reducer.py / reducer code> -fi
```

Como testar o Código sem o Hadoop

É possível testar o código escrito em Map Reduce sem utilizar o Hadoop. Isto pode ser feito com a ajudo do comando sort do linux.

```
cat testfile | ./mapper.py | sort | ./reducer.py
```

Para gerar o "testfile" a partir do arquivo original podemos fazer o seguinte:

```
head -50 <path do arquivo original> > testfile
```

Outras aplicações

Outras aplicações interessante de MapReduce são:

- Processar logs.
- Sistemas de Recomendação
- Detecção de Fraude.
- Classificações.
- etc

Todas essas aplicações tem características em comum, o trabalho pode ser paralelizado, e muitos dados a devem ser processados.

Core do Framework MapReduce

A principal contribuição do framework MapReduce não são as funções map e reduce, mas a escalabilidade e tolerância a falhas obtidas, para uma série de aplicações, ao otimizar o core de execução. Dessa forma, uma implementação single-threaded do MapReduce, normalmente, não será mais rápido do que uma implementação tradicional.

Ao comentar sobre o MapReduce é essencial descrever o funcionamento as funções Shuffle e Sort.

Shuffle

A fase de shuffle consiste em transferir os dados dos Mappers para os Reducers.

Essa fase pode começar antes da fase de mapping terminar para economizar algum tempo. É por isso que as vezes vemos que que o status da fase de Mapping não está em 100% e a fase de Reducing está com o status maior que 0% (mas menor que 33%).

A fase de shuffle se preocupa em agrupar os pares Chave-Valor que contém a mesma chave, para que um conjunto de chaves iguais não seja dividido entre vários reducers. Durante a fase de Map, cada conjunto de chaves iguais é armazenado em um "Partioner". O número desses "Partitioners" é igual ao número de Reducers. Cada chave com um valor específico irá para a partição associada. E reducer ficará responsável por uma partição.

Sort

Assim que as chaves são salvas na partição correta, durante a fase de "Partitioning", o framework do Hadoop começa a ordenar os pares de acordo com as chaves.

Ao garantir que as chaves estão ordenadas, economiza-se o tempo do Reducer, tornando mais simples identificar quando uma nova tarefa de reducing deve começar. Ou seja, ao comparar a chave da tarefa atual com a chave da última tarefa, e elas forem diferentes, sabe-se que uma nova tarefa de reduce deve ser aplicada ao novo conjunto de dado.

Dessa forma, a garantia de que as chaves estão ordenadas é fundamental para o correto funcionamento de um programa escrito seguindo o paradigma MapReduce.

Combiner (Opcional)

Durante a fase de Mapping também seria possível acrescentar uma fase intermediária, chamada de "Combine". Combiner são utilizados para reduzir a quantidade de dados que são enviados para a fase "Reduce". Eles funcionam como "Reducers", mas atuam durante a fase de Sort. Suponha o exemplo citado no tópico anterior, poderíamos calcular quanto cada loja vendeu durante a fase de sort e enviar esse total para a fase de Reduce. Isto seria interessante caso o "reducer" estivesse sobrecarregado.

MapReduce Design Patterns (Resumo)

Existem uma série de Design Patterns utilizados para MapReduce, aqui serão descritos somente três destes padrões.

Filtering Patterns

Este padrão serve para filtragem de dados, ou seja, ele não modifica as entradas dos dados. Um exemplo, seria escolher somente algumas das linhas do nosso arquivo original e ignorar outras.

Alguns exemplos de utilização desse padrão são:

- Amostragem de dados
- Lista de Top-N

No exemplo citado de Top-N, cada Mapper gera o seu Top-N e, por fim, o reducer combina os Top-N de todos os Mapper, gerando um Top-N global.

Summarization Patterns

Esses são padrões que dão um entendimento simples, rápido do conjunto de dados com o qual estamos trabalhando.

Alguns exemplos de utilização desse padrão são:

- Contagem de dados / registros (Ex: Contador de palavras)
- Encontrar Mínimo / Máximo
- Calcular dados estatísticos (Ex: Média, Desvio Padrão, Mediana, ...)
- Criar Index (Ex: Em quais locais do livro posso encontrar uma palavra específica?)

Structural Patterns

É um padrão bastante utilizado quando deseja-se migrar dados de um Banco de Dados Relacional para o Hadoop. Dessa forma, dados que estão relacionados via chave secundária, podem ser combinados de forma hierárquica. Dessa forma, esse padrão normalmente é utilizado para combinar conjuntos de dados, tabela de bancos relacionais.

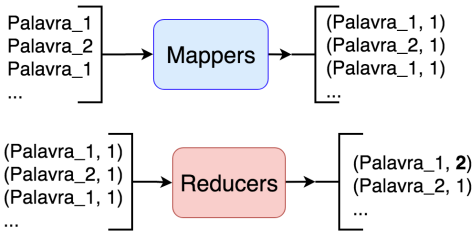
Aplicação de MapReduce neste projeto

Neste projeto foi elaborada uma aplicação bastante simples de MapReduce, que consiste de um contador de palavras.

O papel dos Mappers é de basicamente criar, a partir das palavras de entrada, um par chave-valor. A chave sendo a própria palavra e o valor sendo o número de ocorrências, nesse caso, sempre 1.

O papel do Reducer será receber esses pares de Chave-Valor já ordenados e somar o valor de todas as chaves semelhantes.

A imagem abaixo demonstra como funcionará este processo.



O código java pode ser encontrado em anexo.

YARN (Yet Another Resource Negotiator)

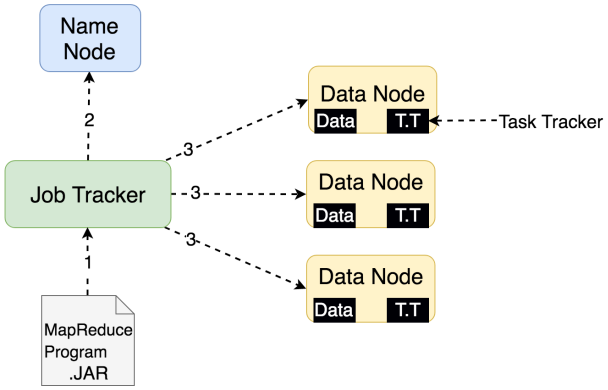
Contexto

YARN Surgiu como uma solução para grande parte dos problemas presentes no Hadoop 1.0. Dessa forma, neste relatório será descrito resumidamente como funcionava a primeira versão do Hadoop, assim como os problemas relativos a essa versão e como o YARN os solucionou.

Entendendo Hadoop 1 (Resumidamente)

O HDFS, ou a parte do Hadoop responsável por armazenar dados, mudou pouco da primeira para a segunda versão do Hadoop. A Maior mudança foi com relação a parte do Hadoop relacionada a processamento de dados, e este será o foco desta tópico do relatório.

A imagem abaixo será utilizada para auxiliar na explicação do funcionamento do Hadoop.



Após escrever nosso código seguindo o padrão MapReduce, ele será executado pelo Hadoop na seguinte ordem:

- i. Primeiramente, o programa MapReduce é submetido ao Job Tracker.
  - O Job Tracker verifica se o programa contém as informações necessárias para ser executado pelo framework. Caso positivo ele verifica qual conjunto de dados (dataset) o programa utilizará, isto é informado pelo usuário.
  - É interessante citar que independentemente de qual local do cluster submetermos o programa, ele irá para o Job Tracker.
  - Além disso, ao submetermos um programa MapReduce para o cluster este passa a ser chamado de Job.
- ii. O Job Tracker verifica, por meio do NameNode, em quais DataNodes o conjunto de dados a ser processado está.
- iii. Após identificar os DataNodes onde estão o dataset, o Job Tracker copia o programa do MapReduce para estes nós.
- iv. A fim de executar o programa, o Job Tracker realiza um chamado para cada um dos Task Trackers, que estão em cada um dos DataNodes, e os passa uma tarefa.

- Os Job Trackers também monitoram o ciclo de vida dos Tasks Trackers.

- Além de tudo, eles coletam o resultados dos Task Trackers e retornam o resultado para o cliente.

Alguns Problemas com Hadoop 1

Após descrever o funcionamento do Hadoop 1, é possível perceber alguns problemas com esta arquitetura. Estes problemas arquiteturais serão detalhados nos tópicos abaixo.

Job Tracker sobrecarregado

Como o Job Tracker fica centralizado em somente uma máquina, ele fica muito sobrecarregado quando está sendo utilizado em clusters muito grandes.

Suponha um cenário em que estão rodando 4.000 Jobs em um cluster e cada um desses possui 10 Tasks. Isso significa que um único Job Tracker, rodando em uma máquina, seria responsável por monitorar 40.000 Task Trackers, coletar os resultados deles e submetendo novos Jobs.

Percebe-se, portanto, que isso é um gargalo nessa arquitetura e é por isso que o Hadoop 1 é limitado a cerca de 4000 nós por cluster.

Possível utilizar somente Map-Reduce

Um outro problema, ou limitação do Hadoop está relacionado ao fato desse framework funcionar somente com MapReduce, não havendo outras alternativas.

Além disso, inicialmente, tudo deveria ser escrito em Java. Para contornar esta última limitação, foram criadas ferramentas como o Hive que permite com que scripts para a manipulação dos dados sejam escritos usando linguagem SQL, facilitando, portanto, o trabalhos de administradores de Banco de Dados. Porém, esses scripts são traduzidos em código MapReduce para que possam rodar no Cluster.

Percebe-se, portanto, que mesmo novas ferramentas sendo criadas para o ecossistema Hadoop, este continua limitado ao paradigma MapReduce.

Controle sobre Recursos das Máquinas

Outro aspecto negativo está relacionado ao fato do Job Tracker não possuir nenhum controle sobre os recursos da máquina dos DataNodes. Sem este conhecimento, algumas máquinas pode ser subutilizadas e outras sobreutilizadas.

Suponhamos um cenário em que temos 2 máquinas, uma com 4GB e outra com 8GB de RAM e que rodarão um total de 12 mappers, cada utilizando 1GB. Caso cada DataNode possua 6 "Input Splits", a máquina com 4GB rodará 4 Mappers e terá 2 esperando para serem executados, enquanto a máquina com 8GB rodará os 6 e terá 2GB sobrando.

Percebe-se, portanto, que não há muito planejamento para o escalonamento dos Jobs. Também não há o monitoramento de recursos.

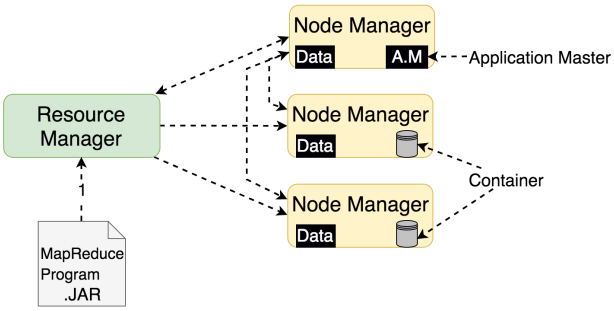
Hadoop 2 + YARN

A "parte" do Hadoop 2 responsável por processamento é chamado de YARN (Yet Another Resource Negotiator). Portanto, pode-se dizer que o framework responsável pelo armazenamento de dados do hadoop é o HDFS, já o framework responsável pelo processamento desses dados é o YARN.

Funcionamento

O papel de Job Tracker e Task Tracker é substituído por outros 3 Daemons no Hadoop 2:

- Resource Manager.
- Node Manager.
- Application Master.



De forma resumida, o funcionamento do Hadoop 2.0 ocorre da seguinte forma:

- i. Quando for submetido um novo Job para o Hadoop, o Resource Manager o aceita e contacta algum dos Node Managers e o informa que um novo Job deverá ser processado.
- ii. O Node Manager cria um Daemon chamado Application Master na máquina em que se encontra.
  - É interessante citar que o Número de Application Masters será sempre igual ao número de Jobs rodando no cluster.
- iii. O Daemon Application master é responsável por executar o Job relacionado a ele. Ele verifica se este Job é pequeno (Uber), se for ele executa na mesma máquina. Caso contrário, ele pede para o Resource Manager para localizar outros nós com recursos disponíveis. O resource Manager, obviamente, leva em conta a localização dos dados ao disponibilizar recursos.
- iv. De acordo com a resposta do Resource Manager, o Application Master contacta os Node Managers e pede para que eles criem um Container. Esse container também é chamado de Yarn Child, e é onde as tasks serão executadas.
- v. Através do HDFS, os containers obtém os recursos necessários para executar a tarefa e então o programa é executados.
- vi. Ao finalizar a tarefa, o Yarn Child avisa o App Master que por sua vez envia o resultado para o Cliente.

Resource Manager

O Resource Manager tem basicamente duas responsabilidades:

- Scheduler
- Application Manager

Scheduler está relacionado somente ao escalonamento dos Jobs. Enquanto o Application Manager é responsável pelo monitoramento e controles das aplicações. O Resource Manager, também contém informações globais relacionado ao estados de cada um dos nós e informação sobre seus recursos. Estas informações são enviadas pelos Node Managers.

Tipos de Scheduling

Um problema erro comum ao configurar um cluster Hadoop é não analisar qual o tipo de escalonamento ideal. Muitas vezes são executados Jobs muito demorados no cluster e estes impedem a execução de outros mais importantes e rápidos, isso é comum quando é utilizado um escalonador do tipo FIFO. Existem vários outros tipos de escalonadores, os mais utilizados para o Hadoop são:

- Fair Scheduler
- Capacity Scheduler

Node Manager

O Node Manager está presente em todos nós escravos e eles são responsáveis por criar e gerenciar containers. Além disso, informam o Resource manager sobre o estado dos recursos na máquina.

Application Master

O Application master é o responsável por executar o Job, é ele quem monitora cada uma das tasks sendo realizadas nos diferentes nós. Após a execução do Job esse processo é finalizado.

Como existe um Application Master para cada job que está sendo executado, a introdução do Application Master foi bem interessante, pois resolveu o gargalo relacionado ao Job Tracker que existia na arquitetura anterior, isso permite uma maior escalabilidade.

Yarn Child / Container

O Yarn Child é responsável por executar as tasks. Além disso, ele atualiza o application master com informações, como o progresso da execução da task.

Características

Algumas características importantes trazidas pelo YARN ao Hadoop são:

Alternativas ao MapReduce

Com o Surgimento do YARN, o Hadoop não está mais limitado ao MapReduce e pode utilizar diversas outros framework.

Alguns exemplos são:

- Spark: Diferentemente do MapReduce, os dados são armazenados em Memória RAM, isso faz com que o processamento dos dados seja bem mais rápido.
- Storm: Framework interessante para aplicações que demandam processamento em tempo-real. Ele processa os dados assim que entram no sistema, ou seja, é um "Stream processing system".

Dentre outros.

Maior escalabilidade

Conforme citado anteriormente, ao resolver o gargalo associado ao Job Tracker, o YARN melhorou bastante a escalabilidade do framework Hadoop.

Melhor controle dos recursos do Cluster

Conforme citado anteriormente, o controle de recursos do cluster melhorou significativamente.

Containers

Possibilitam uma melhor utilização de memória do cluster e podem rodar tasks genéricas, diferentemente do Hadoop 1, em que as únicas tasks possíveis de serem executadas eram Map e Reduce.

Instalando hadoop em Standalone

- Você pode entrar na página do hadoop e escolher a versão.
- Nesse experimento usamos o seguinte pacote
- Para descompactar o arquivo use: `tar -xvzf hadoop-2.x.tar.gz`
- Onde `x` é a subversão do Hadoop 2.
- O tutorial usado para configuração do ambiente foi uma página do site oficial
- Primeiramente é necessário ter instalado o Java instalado, nesse presente experimento será utilizado a plataforma Java Oracle 8, portanto basta escolher qual JVM deseja usar(open-jdk, oracle-jdk, etc). Você pode conferir se o seu ambiente já está devidamente configurado utilizando no seu terminal o seguinte comando:

```
$ echo $JAVA_HOME
/usr/lib/jvm/java-8-oracle`
```

- Se obtiver uma saída parecida, significa que seu ambiente Java já está devidamente configurado, caso contrário siga algum tutorial sobre a instalação do ambiente java e da configuração de suas variáveis de ambiente, necessários para o uso do hadoop. Existem vários desses tutorais, não vamos nos reter a isso.
- Além do java, existem duas dependências de sistema que é preciso configurar, `ssh e rsync`. Numa distro debian:
- `$ sudo apt-get ssh rsync`
- Primeiramente vamos tentar executar o ambiente em Standalone, modo local não distribuído.
- Para isso vamos mover a pasta descompactada acima para `/usr/`, alguns tutoriais usam a pasta `/etc`, isso não tem grande relevância:

- $ sudo mv hadoop-2.8.0/ /usr/
- Agora é preciso editar o bashrc para conseguir executar o Hadoop em qualquer terminal.
- `$ sudo vim ~/.bashrc`
- Você pode usar qualquer editor de texto para isso, uma vez em modo de edição, adicione ao final do arquivo os seguintes comandos:

```
# HADOOP VARIABLES - considering hadoop folder into /usr/
export HADOOP_HOME=/usr/hadoop-2.8.0
export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
export HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar

export HADOOP_OPTS=-Djava.net.preferIPv4Stack=true
export HADOOP_CONF_DIR=${HADOOP_HOME}/etc/hadoop

export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
```

- Salve o arquivo, digite `bash` no seu terminal para as redefinir as configurações de bash e confira o resultado executando:
- `$ hadoop version`
- Deve aparecer uma tela mostrando a versão do hadoop seguido de outras informações.
- Para uma simples demonstração Standalone, vamos entrar dentro da nossa pasta do Hadoop `/usr/hadoop-2.8.0` e executar os seguintes comandos(Perceba que estamos dentro de `/usr/hadoop-2.8.0`):

```
/usr/hadoop-2.8.0$ cp etc/hadoop/*.xml input/

/usr/hadoop-2.8.0$ hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.8.0.jar grep input/ output 'dfs[a-z.
```

- Após isso deve aparecer um grande log no seu terminal. Se você executar:
- `$ ls output/`
- Irá ver alguns arquivos criados na pasta pelo hadoop. Você acaba de executar seu primeiro código no hadoop.

Configuração Distribuída

Para nossos experimentos é necessário a instalação distribuída, para isso é necessário configurar várias máquinas que serão os nós do cluster. Nesse experimento foi utilizado 3 nós, sendo 1 o master e slave, e os outros dois sendo apenas slaves, um será configurado junto ao master e o outro colocado depois, para demonstrar a elasticidade do sistema. Segue abaixo os procedimentos para configuração dos mesmos.

- Os passos a seguir foram feitos com algumas adaptações, mas ainda baseado nesse tutorial: https://linoxide.com/cluster/setup-hadoop-multi-node-cluster-ubuntu/
- Para prosseguir é necessário realizar a configuração Standalone acima.
- Para facilitar trabalhar com os endereços das máquinas, vamos editar o arquivo `/etc/hosts` em cada nó.
- No master, edite `/etc/hosts` no seguinte formato:

```
<ip-master>   localhost   <nome-da-maquina>   master
<ip-slave-1>  slave-1
```

- Para ver o `<ip-master>` e `<ip-slave-1>` basta digitarmos: `ifconfig` em cada um desses nós.
- O nome da máquina pode ser obtido usando o comando `$ hostname` no terminal.
- No slaves, edite também `/etc/hosts` no seguinte formato:

```
<ip-slave-1>   localhost   <nome-da-maquina>   slave-1
<ip-master>    master
```

- Para não ser preciso reconfigurar isso toda vez que um nó for desligado ou reconectado, pode-se fixar os IPs dessas máquinas.
- Para funcionamento mínimo da aplicação, o master deve se conectar aos slaves, ou melhor dizendo, o `NameNode` necessita precisa utilizar os `DataNodes` . Portanto precisamos configurar ssh para que a autenticação não barre isso.
- Então foi configurado chave ssh para que o `NameNode` possa fazer uso dos outros nós sem autênticação:

```
fsdadmin@HPHost02:~$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/fsdadmin/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/fsdadmin/.ssh/id_rsa.
Your public key has been saved in /home/fsdadmin/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:mg6IeFhPq20/LDgvpTbDeaMIlq7fZpR+rHspsWAnQOE fsdadmin@HPHost02
The key's randomart image is:
+---[RSA 2048]----+
| ..              |
|..               |
|.E               |
|.                |
| .. ..  S        |
|.+=+*. o         |
|==o@+=o.         |
|+o@+@oB          |
|+++%*Xo.         |
+----[SHA256]-----+
...
```

- Para autorizar a chave criada no master, basta usar:

```
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

- Teste se a configuração funcionou tentando usar:

```
$ ssh localhost
```

- Para copiar a chave ssh do master para os slaves, é necessário usar:

```
fsdadmin@HPHost02:~$ ssh-copy-id -i ~/.ssh/id_rsa.pub kuwener@slave-1
```

- Observe que nesse exemplo, 'kuwener' era nome do usuário da máquina slave, 'slave-1' o endereço dessa máquina.
- Costumam criar um usuário hadoop em todos os nós para facilitar essa configuração.
- Teste sua configuração:

```
ssh slave-1
```

- Agora é possível conectar master com master e master com slave.
- Desde já podemos notar a dependencia que o hadoop tem com o `NameNode` , tal que os `DataNodes` não precisam se conectar a ele para funcionamento mínimo do sistema, mas é crítico que ele consiga ter acesso a todos `DataNodes` .
- Agora para que a configuração distribuída funcione, é necessário em todos os nós configurar/criar um arquivo `masters` em `$HADOOP_HOME/etc/hadoop` :

```
master
```

- Esse arquivo simplesmente indica quem são os masters para o hadoop. É possível ter mais de um.

- Na mesma pasta, é necessário editar/criar um arquivo `slaves` com conteúdo no seguinte formato:

```
master
kuwener@slave-1
```

- Perceba, antes de `@` temos o nome de usuário, e após o endereço ip da máquina. Caso tivesse sido configurado um usuário em comum para todo cluster, seri necessário usar apenas o endereço, como o master.
- Além do mais, estamos colocando o master como um slave também, isso significa que apesar do nó ter um `NameNode`, ele também pode ter um `DataNode`.
- Em `$HADOOP_HOME/etc/hadoop/hadoop-env.sh` é necessário atualizar a definição da variável JAVA_HOME. Coloque a que está sendo usada no SO: `echo $JAVA_HOME`.
- Deve-se também criar os diretórios que irão compor o HDFS no `NameNode` e `DataNodes`.
- Foi escolhido:

```
mkdir -p $HADOOP_HOME/hadoop2_data/hdfs/namenode
mkdir -p $HADOOP_HOME/hadoop2_data/hdfs/datanode
```

- Agora uma série de arquivos de `$HADOOP_HOME/etc/hadoop`. É recomendado que se faça em um node e depois replique via `scp`.
- Em `core-site.xml`

```
<configuration>
    <property>
            <name>fs.defaultFS</name>
            <value>hdfs://master:9000</value>
    </property>
</configuration>
```

- Em `hdfs-site.xml`

```
<configuration>
        <property>
                <name>dfs.replication</name>
                <value>2</value>
        </property>

        <property>
                <name>dfs.permissions</name>
                <value>false</value>
        </property>

        <property>
                <name>dfs.namenode.name.dir</name>
                <value>/usr/hadoop-2.8.0/hadoop2_data/hdfs/namenode</value>
        </property>

        <property>
                <name>dfs.datanode.data.dir</name>
                <value>/usr/hadoop-2.8.0/hadoop2_data/hdfs/datanode</value>
        </property>

</configuration>
```

- Em `yarn-site.xml`

```
<configuration>
    <property>
            <name>yarn.nodemanager.aux-services</name>
            <value>mapreduce_shuffle</value>
    </property>
    <property>
            <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
            <value>org.apache.hadoop.mapred.ShuffleHandler</value>
```

```
    </property>
</configuration>
```

- Em `mapred-site.xml`

```
<configuration>
  <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
  </property>
</configuration>
```

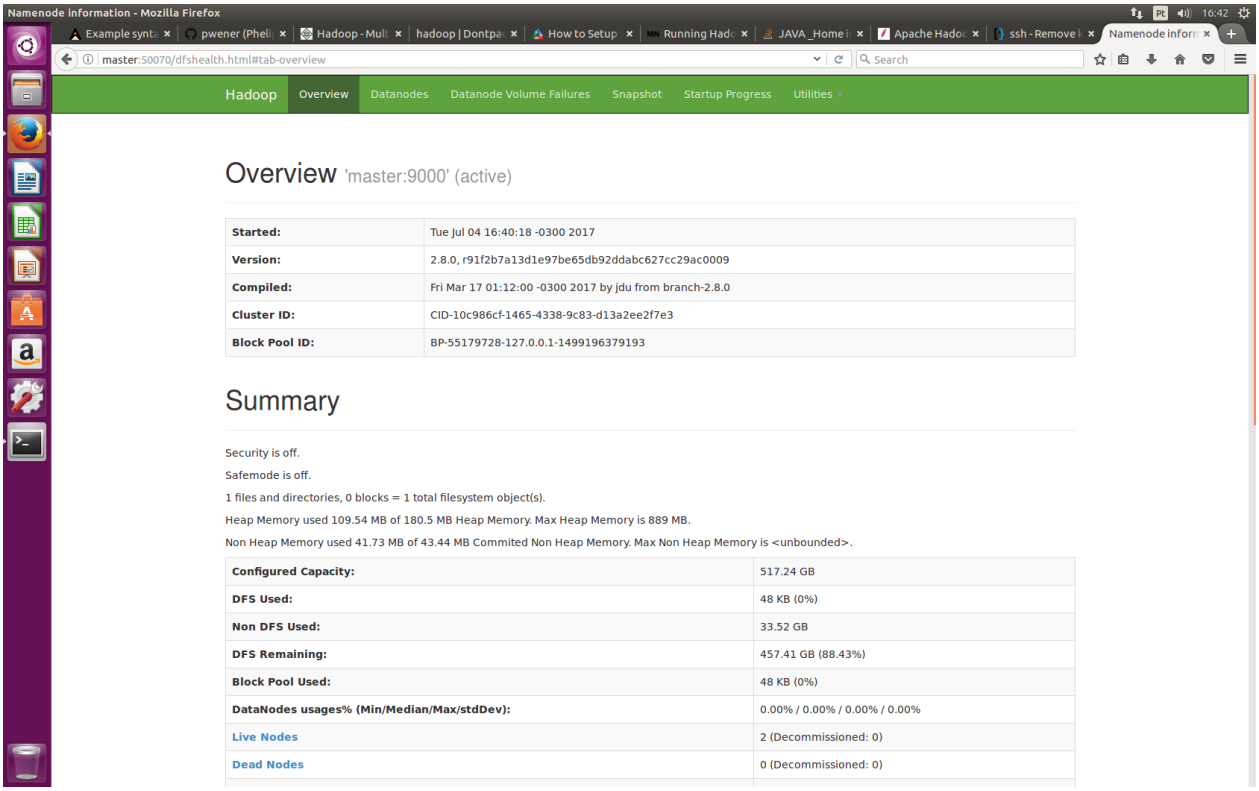
- Relembrando: Essa configuração acima deve ser igual em todos os nós.
- Agora é preciso formatar os nós para começar a usar, basta que no master se use: `hadoop namenode -format`
- Então, ainda no master, dentro da raiz do hadoop($HADOOP_HOME), use: `$ sbin/start-dfs.sh`
- Talvez será requisitado confirmação de autenticação, se nenhum erro ocorrer, o HDFS está preparado.
- Agora é necessário iniciar o yarn, para isso use: `$sbin/start-yarn.sh` .
- ATENÇÃO: Esses dois comandos acima apenas precisam ser executados no master, conhecido também por NameNode.
- Agora se executado `$ jps` no master, será obtido algo como:

```
$ jps
28640 NodeManager
27985 NameNode
28947 Jps
28137 DataNode
28330 SecondaryNameNode
28511 ResourceManager
```

- Nos slaves:

```
$ jps
20651 DataNode
20987 NodeManager
21133 Jps
```

- Se no NameNode e nos DataNodes não retornou algo parecido, algo provavelmente está errado.
- Para saber o status dos nós no cluster entre em http://master:50070/dfshealth.html
- Essa interface web tem várias informações sobre seu cluster, como espaço disponível, quais nós estão executando, onde estão eles e sua disponibilidade.



- Perceba que temos dois nós executando.

Adicionando um novo nó

- Para adicionar um novo nó é bem simples, basicamente deve-se realizar os seguintes passos:
- Configurar o ambiente em Standalone
- Configurar o `/etc/hosts`, exemplo no caso de um novo node:

```
192.168.133.193 localhost vagrant       slave-2
192.168.133.129 slave-1
192.168.133.149 master
```

- Adicionar a linha do novo slave em `/etc/hosts` no master:

```
...
192.168.133.129 slave-2
```

- Configurar ssh do master para o `Datanode`.
- Atualizar o arquivo slaves, adicionando esse novo slave.
- É preciso ainda atualizar o arquivo `$HADOOP_HOME/etc/hadoop/slaves` com o novo node, para isso basta atualizar a master e dar scp em todos `DataNodes`:

```
$ scp fsdadmin@master:/usr/hadoop-2.8.0/etc/ .
```

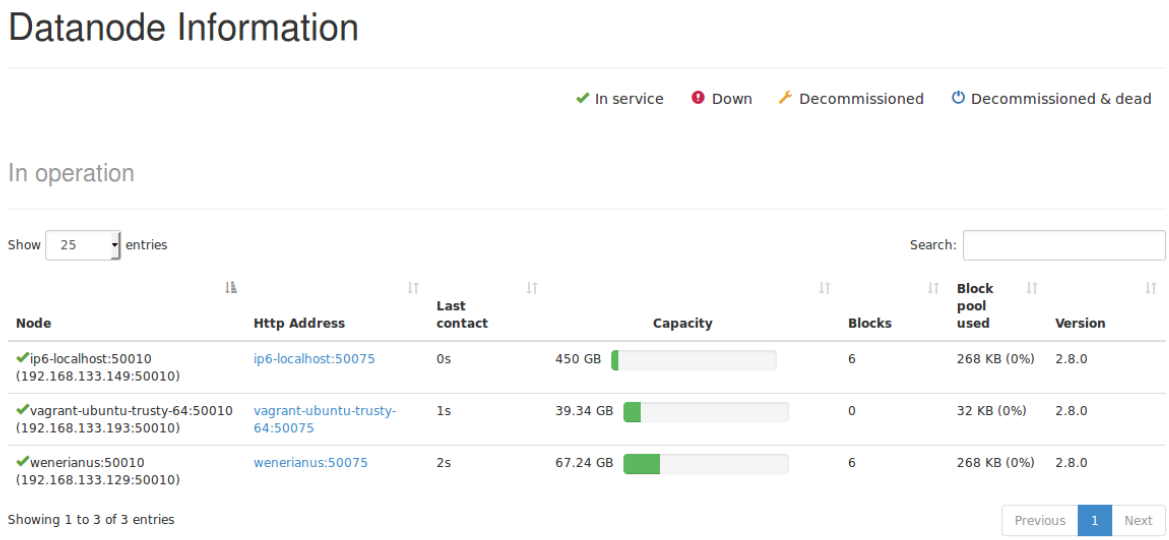
- Após devidamente configurados, deve-se restartar todo o serviço no master:

```
$ sbin/stop-all.sh
...
$ sbin/start-dfs.sh
...
$ sbin/start-yarn.sh
```

- Agora ao entrar na interface web: http://master:50070/dfshealth.html será possível ver 3 nós em execução:

DFS Remaining:	491.09 GB (88.23%)
Block Pool Used:	568 KB (0%)
DataNodes usages% (Min/Median/Max/stdDev):	0.00% / 0.00% / 0.00% / 0.00%
Live Nodes	3 (Decommissioned: 0)
Dead Nodes	0 (Decommissioned: 0)
Decommissioning Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion	0
Block Deletion Start Time	Wed Jul 05 14:22:23 -0300 2017
Last Checkpoint Time	Wed Jul 05 14:23:35 -0300 2017

- Se clicado no link "Live Nodes", é possível ainda obter mais detalhes:



Executando um Contador de Palavras

- A partir do tutorial do site oficial hadoop.apache, criamos o exemplo do contador de palavras.
- Para compilar o .java use:
- `bin/hadoop com.sun.tools.javac.Main WordCount.java`
- Entre com dois textos na pasta input, em dois arquivos separados e rode o seguinte comando:

```
$ hadoop fs -ls ~/workspaces/hadoop/web_analyser/wordcount/input/
Found 2 items
-rw-rw-r--   1 kuwener kuwener       5398 2017-06-27 01:10 /home/kuwener/workspaces/hadoop/web_analyser/wordcount/inp
-rw-rw-r--   1 kuwener kuwener       6392 2017-06-27 01:09 /home/kuwener/workspaces/hadoop/web_analyser/wordcount/inp
```

- O que acabou de ser feito? Basicamente foi executado o comando `ls` dentro do filesystem do hadoop. É o que o comando `fs` faz!
- Para ver alguns comandos do hadoop basta digitar no seu terminal `hadoop` .
- Agora podemos executar nosso jar usando:

```
$ hadoop jar wc.jar WordCount ./input ./output
```

- O caracter '.' é convertido para o diretório atual, se tudo ocorreu bem você deve receber uma longa mensagem, similar a essa:

```
17/06/27 01:15:20 INFO mapreduce.Job: Job job_local2134868558_0001 completed successfully
17/06/27 01:15:20 INFO mapreduce.Job: Counters: 30
        File System Counters
                FILE: Number of bytes read=69741
                FILE: Number of bytes written=1035046
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
        Map-Reduce Framework
                Map input records=236
                Map output records=2081
                Map output bytes=20105
                Map output materialized bytes=14431
                Input split bytes=274
                Combine input records=2081
                Combine output records=1086
                Reduce input groups=979
                Reduce shuffle bytes=14431
                Reduce input records=1086
                Reduce output records=979
                Spilled Records=2172
                Shuffled Maps =2
                Failed Shuffles=0
                Merged Map outputs=2
                GC time elapsed (ms)=10
                Total committed heap usage (bytes)=860356608
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=11790
        File Output Format Counters
                Bytes Written=9429
```

- Você pode ver o resultado da execução em:
- `$ hadoop fs -cat ./output/part-r-00000 .`
- Bem, basicamente conseguimos rodar um exemplo bem simples onde se tem 2 entradas e uma inferência de resultado. Ótimo exemplo para entender Map-Reduce no Hadoop.

Parser

- Para que serve o Parser?

Inicialmente o contexto projeto consistia em analisar o preço de algum ativo, como ações de uma empresa, preços de matérias primas, etc.

Dessa forma, baixariamos do Google o conteúdo de uma série de sites durante um período em que o ativo estava com o preço alto, e, posteriormente, durante o período em que o ativo estava barato. Dessa forma, seriam obtidos dois tipos de arquivos categorizados como "preço_alto.txt" e "preço_baixo.txt". A partir daí utilizariamos o Hadoop em conjunto com técnicas de Machine Learning para avaliar se existe correlação entre as palavras de cada arquivo com o preço daquele ativo.

A partir dessa correlação, talvez fosse possível inserir textos atuais e avaliar se o preço do ativo relacionado àquele texto aparentemente está em um período de alte ou de baixa.

- O que o parser faz?

Ele funciona como um web crawler com profundidade de busca limitada a 1 (depth of 1). Ele faria um busca no google seguindo o padrão citado abaixo, e baixaria o conteúdo dos sites encontrados por meio dessa pesquisa.

Além disso, ele deprezaria as tags HTML dos sites a fim de gerar um conteúdo válido a ser avaliado posteriormente.

Parâmetros enviados via URL utilizados:

- q= -> termo de busca.

- num= -> número de links por página.
- start= -> qual o número do link em que se inicia a busca.

Exemplo de busca:

```
http://www.google.com/search?q=apple&num=100&start=100
```

- Problemas encontrados

Erro 503

Após rodar o parser algumas vezes o google começou a bloquear nossas buscas gerando um erro 503, possivelmente, estava exibindo um CAPTCHA para verificar se era algum robô que estava realizando a pesquisa.

Resolvemos mostrar na tela o resultado obtido quando o erro 503 ocorria, e o conteúdo consistia basicamente do que está abaixo:

```
Our systems have detected unusual traffic from your computer network.  This page checks to see if it's really you sen

This page appears when Google automatically detects requests coming from your computer network which appear to be in

The block will expire shortly after those requests stop.  In the meantime, solving the above CAPTCHA will let you con

This traffic may have been sent by malicious software, a browser plug-in, or a script that sends automated requests.

Sometimes you may be asked to solve the CAPTCHA if you are using advanced terms that robots are known to use, or send
```

API do Google

Após identificar o problema do erro 503, pensamos em utilizar a API de busca do Google, porém a versão gratuita dela permite que poucas consultas sejam feitas diariamente. Já a versão paga é bastante cara e o pagamento depende do número de buscas, dessa forma achamos inconveniente baixar sites do Google para utilizar como input do contador de palavras, visto que o volume de dados seria pequeno.

WordCount

Visto que o contexto do parser acima explicado não gerou bons resultados, e uma vez que o foco era testar o comportamento do Hadoop, foi retomado o contexto contador de palavras utilizando um arquivo de aproximadamente 2,0 GB de nomes aleatórios.

É esperado que essa quantidade de dados seja suficiente para testar a vantagem de se utilizar o Hadoop. Para analisar isso, nos experimentos abaixo será relatado tanto os testes desse contador em Hadoop, quanto na sua forma tradicional, sem a plataforma, para que no fim seja possível obter uma conclusão acerca das vantagens/desvantagens do uso do Hadoop.

Testes no Hadoop

- Fizemos um teste com o contador usando apenas um arquivo de 1.9 GB de nomes aleatórios, afim de testar o comportamento do hadoop trabalhando com essa ordem de tamanho.
- Vale lembrar que essa etapa foi realizada na configuração comum, usando 1 namenode e 3 datanodes, o qual 1 estava junto ao namenode e os outros dois em máquinas distintas.
- Para tal, foi utilizado o comando abaixo no hadoop fs, para transferir o arquivo do disco rígido do master para o HDFS.

```
$ hadoop fs -put ~/130718954_words.input hdfs://master:9000/input/
```

- O comando mostra uma série de logs, o qual é possível visualizar erros ou se a operação foi bem sucedida.
- Depois da cópia para o hdfs, o WordCount foi executado:
- Observação: O log de todas execuções foi cortado na parte de MapReduce onde mostra a porcentagem da tarefa, afim de reduzir a quantidade de linhas descritas no relatório.

```
$ hadoop jar wc.jar WordCount hdfs://master:9000/input hdfs://master:9000/output
17/07/06 18:19:47 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
17/07/06 18:19:48 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the
17/07/06 18:19:48 INFO input.FileInputFormat: Total input files to process : 1
17/07/06 18:19:51 INFO hdfs.DataStreamer: Exception in createBlockOutputStream
java.io.IOException: Got error, status=ERROR, status message , ack with firstBadLink as 192.168.133.193:50010
        at org.apache.hadoop.hdfs.protocol.datatransfer.DataTransferProtoUtil.checkBlockOpStatus(DataTransferProtoUti
        at org.apache.hadoop.hdfs.DataStreamer.createBlockOutputStream(DataStreamer.java:1643)
        at org.apache.hadoop.hdfs.DataStreamer.nextBlockOutputStream(DataStreamer.java:1547)
        at org.apache.hadoop.hdfs.DataStreamer.run(DataStreamer.java:658)
17/07/06 18:19:51 WARN hdfs.DataStreamer: Abandoning BP-55179728-127.0.0.1-1499196379193:blk_1073741858_1034
17/07/06 18:19:51 WARN hdfs.DataStreamer: Excluding datanode DatanodeInfoWithStorage[192.168.133.193:50010,DS-3f4cc3d
17/07/06 18:19:51 INFO mapreduce.JobSubmitter: number of splits:15
17/07/06 18:19:51 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1499374686704_0001
17/07/06 18:19:52 INFO impl.YarnClientImpl: Submitted application application_1499374686704_0001
17/07/06 18:19:52 INFO mapreduce.Job: The url to track the job: http://HPHost02:8088/proxy/application_1499374686704_
17/07/06 18:19:52 INFO mapreduce.Job: Running job: job_1499374686704_0001
17/07/06 18:19:59 INFO mapreduce.Job: Job job_1499374686704_0001 running in uber mode : false
17/07/06 18:19:59 INFO mapreduce.Job:  map 0% reduce 0%
17/07/06 18:20:17 INFO mapreduce.Job:  map 1% reduce 0%
17/07/06 18:20:18 INFO mapreduce.Job:  map 3% reduce 0%
17/07/06 18:20:21 INFO mapreduce.Job:  map 4% reduce 0%
17/07/06 18:20:23 INFO mapreduce.Job:  map 6% reduce 0%
17/07/06 18:20:24 INFO mapreduce.Job:  map 8% reduce 0%
...
17/07/06 18:33:34 INFO mapreduce.Job:  map 100% reduce 97%
17/07/06 18:33:46 INFO mapreduce.Job:  map 100% reduce 98%
17/07/06 18:34:04 INFO mapreduce.Job:  map 100% reduce 99%
17/07/06 18:34:16 INFO mapreduce.Job:  map 100% reduce 100%
17/07/06 18:34:23 INFO mapreduce.Job: Job job_1499374686704_0001 completed successfully
17/07/06 18:34:24 INFO mapreduce.Job: Counters: 50
        File System Counters
                FILE: Number of bytes read=6698777882
                FILE: Number of bytes written=9488445302
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=2026116230
                HDFS: Number of bytes written=2238785729
                HDFS: Number of read operations=48
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
        Job Counters
                Killed map tasks=1
                Launched map tasks=16
                Launched reduce tasks=1
                Rack-local map tasks=16
                Total time spent by all maps in occupied slots (ms)=1099789
                Total time spent by all reduces in occupied slots (ms)=769374
                Total time spent by all map tasks (ms)=1099789
                Total time spent by all reduce tasks (ms)=769374
                Total vcore-milliseconds taken by all map tasks=1099789
                Total vcore-milliseconds taken by all reduce tasks=769374
                Total megabyte-milliseconds taken by all map tasks=1126183936
                Total megabyte-milliseconds taken by all reduce tasks=787838976
        Map-Reduce Framework
                Map input records=130718954
                Map output records=130718954
                Map output bytes=2548933037
                Map output materialized bytes=2787487486
                Input split bytes=1665
                Combine input records=260799855
                Combine output records=258729171
                Reduce input groups=123904370
                Reduce shuffle bytes=2787487486
                Reduce input records=128648270
                Reduce output records=123904370
                Spilled Records=438509601
                Shuffled Maps =15
                Failed Shuffles=0
                Merged Map outputs=15
                GC time elapsed (ms)=13454
                CPU time spent (ms)=756920
                Physical memory (bytes) snapshot=4443439104
                Virtual memory (bytes) snapshot=30708047872
                Total committed heap usage (bytes)=3250061312
        Shuffle Errors
```

```
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=2026114565
        File Output Format Counters
                Bytes Written=2238785729
```

- Então desconectado 1 node, foi rodado de novo o job após excluir output com `$ hadoop fs -rmr hdfs://master:9000/output` . Isto é necessário porque o programa cria sempre a pasta output vazia.
- Para checar os status dos nodes, é possivel executar o seguinte comando:

```
$ hadoop dfsadmin -report hdfs://master:9000/
DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.

Configured Capacity: 555386372096 (517.24 GB)
Present Capacity: 478432574673 (445.58 GB)
DFS Remaining: 470042180642 (437.76 GB)
DFS Used: 8390394031 (7.81 GB)
DFS Used%: 1.75%
Under replicated blocks: 2
Blocks with corrupt replicas: 0
Missing blocks: 0
Missing blocks (with replication factor 1): 0
Pending deletion blocks: 0

-------------------------------------------------
Live datanodes (2):

Name: 192.168.133.129:50010 (slave-1)
Hostname: wenerianus
Decommission Status : Normal
Configured Capacity: 72203280384 (67.24 GB)
DFS Used: 4200034374 (3.91 GB)
Non DFS Used: 15562559418 (14.49 GB)
DFS Remaining: 48244320273 (44.93 GB)
DFS Used%: 5.82%
DFS Remaining%: 66.82%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 5
Last contact: Thu Jul 06 19:03:05 BRT 2017


Name: 192.168.133.209:50010 (localhost)
Hostname: ip6-localhost
Decommission Status : Normal
Configured Capacity: 483183091712 (450.00 GB)
DFS Used: 4190359657 (3.90 GB)
Non DFS Used: 31854854039 (29.67 GB)
DFS Remaining: 421797860369 (392.83 GB)
DFS Used%: 0.87%
DFS Remaining%: 87.30%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 6
Last contact: Thu Jul 06 19:03:05 BRT 2017


Dead datanodes (1):

Name: 192.168.133.193:50010 (slave-2)
Hostname: vagrant-ubuntu-trusty-64
```

```
Decommission Status : Normal
Configured Capacity: 0 (0 B)
DFS Used: 0 (0 B)
Non DFS Used: 4548063495 (4.24 GB)
DFS Remaining: 0 (0 B)
DFS Used%: 100.00%
DFS Remaining%: 0.00%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 0
Last contact: Thu Jul 06 18:39:50 BRT 2017
```

- É possível notar que um DataNode está morto, conforme desejado.
- Ao rodar então o `wc.jar` sem 1 datanode, obtivemos:

```
$ hadoop jar wc.jar WordCount hdfs://master:9000/input hdfs://master:9000/output
17/07/06 18:51:34 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
17/07/06 18:51:34 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the
17/07/06 18:51:53 INFO hdfs.DataStreamer: Exception in createBlockOutputStream
java.io.IOException: Got error, status=ERROR, status message , ack with firstBadLink as 192.168.133.193:50010
        at org.apache.hadoop.hdfs.protocol.datatransfer.DataTransferProtoUtil.checkBlockOpStatus(DataTransferProtoUti
        at org.apache.hadoop.hdfs.DataStreamer.createBlockOutputStream(DataStreamer.java:1643)
        at org.apache.hadoop.hdfs.DataStreamer.nextBlockOutputStream(DataStreamer.java:1547)
        at org.apache.hadoop.hdfs.DataStreamer.run(DataStreamer.java:658)
17/07/06 18:51:53 WARN hdfs.DataStreamer: Abandoning BP-55179728-127.0.0.1-1499196379193:blk_1073741885_1061
17/07/06 18:51:53 WARN hdfs.DataStreamer: Excluding datanode DatanodeInfoWithStorage[192.168.133.193:50010,DS-3f4cc3d
17/07/06 18:51:53 INFO input.FileInputFormat: Total input files to process : 1
17/07/06 18:51:56 INFO hdfs.DataStreamer: Exception in createBlockOutputStream
java.io.IOException: Got error, status=ERROR, status message , ack with firstBadLink as 192.168.133.193:50010
        at org.apache.hadoop.hdfs.protocol.datatransfer.DataTransferProtoUtil.checkBlockOpStatus(DataTransferProtoUti
        at org.apache.hadoop.hdfs.DataStreamer.createBlockOutputStream(DataStreamer.java:1643)
        at org.apache.hadoop.hdfs.DataStreamer.nextBlockOutputStream(DataStreamer.java:1547)
        at org.apache.hadoop.hdfs.DataStreamer.run(DataStreamer.java:658)
17/07/06 18:51:56 WARN hdfs.DataStreamer: Abandoning BP-55179728-127.0.0.1-1499196379193:blk_1073741887_1063
17/07/06 18:51:56 WARN hdfs.DataStreamer: Excluding datanode DatanodeInfoWithStorage[192.168.133.193:50010,DS-3f4cc3d
17/07/06 18:51:59 INFO hdfs.DataStreamer: Exception in createBlockOutputStream
java.io.IOException: Got error, status=ERROR, status message , ack with firstBadLink as 192.168.133.193:50010
        at org.apache.hadoop.hdfs.protocol.datatransfer.DataTransferProtoUtil.checkBlockOpStatus(DataTransferProtoUti
        at org.apache.hadoop.hdfs.DataStreamer.createBlockOutputStream(DataStreamer.java:1643)
        at org.apache.hadoop.hdfs.DataStreamer.nextBlockOutputStream(DataStreamer.java:1547)
        at org.apache.hadoop.hdfs.DataStreamer.run(DataStreamer.java:658)
17/07/06 18:51:59 WARN hdfs.DataStreamer: Abandoning BP-55179728-127.0.0.1-1499196379193:blk_1073741889_1065
17/07/06 18:51:59 WARN hdfs.DataStreamer: Excluding datanode DatanodeInfoWithStorage[192.168.133.193:50010,DS-3f4cc3d
17/07/06 18:51:59 INFO mapreduce.JobSubmitter: number of splits:15
17/07/06 18:52:02 INFO hdfs.DataStreamer: Exception in createBlockOutputStream
java.io.IOException: Got error, status=ERROR, status message , ack with firstBadLink as 192.168.133.193:50010
        at org.apache.hadoop.hdfs.protocol.datatransfer.DataTransferProtoUtil.checkBlockOpStatus(DataTransferProtoUti
        at org.apache.hadoop.hdfs.DataStreamer.createBlockOutputStream(DataStreamer.java:1643)
        at org.apache.hadoop.hdfs.DataStreamer.nextBlockOutputStream(DataStreamer.java:1547)
        at org.apache.hadoop.hdfs.DataStreamer.run(DataStreamer.java:658)
17/07/06 18:52:02 WARN hdfs.DataStreamer: Abandoning BP-55179728-127.0.0.1-1499196379193:blk_1073741891_1067
17/07/06 18:52:02 WARN hdfs.DataStreamer: Excluding datanode DatanodeInfoWithStorage[192.168.133.193:50010,DS-3f4cc3d
17/07/06 18:52:02 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1499374686704_0002
17/07/06 18:52:03 INFO impl.YarnClientImpl: Submitted application application_1499374686704_0002
17/07/06 18:52:03 INFO mapreduce.Job: The url to track the job: http://HPHost02:8088/proxy/application_1499374686704_
17/07/06 18:52:03 INFO mapreduce.Job: Running job: job_1499374686704_0002
17/07/06 18:52:09 INFO mapreduce.Job: Job job_1499374686704_0002 running in uber mode : false
17/07/06 18:52:09 INFO mapreduce.Job:  map 0% reduce 0%
17/07/06 18:52:26 INFO mapreduce.Job:  map 1% reduce 0%
17/07/06 18:52:28 INFO mapreduce.Job:  map 3% reduce 0%
17/07/06 18:52:30 INFO mapreduce.Job:  map 4% reduce 0%
17/07/06 18:52:32 INFO mapreduce.Job:  map 6% reduce 0%
17/07/06 18:52:33 INFO mapreduce.Job:  map 7% reduce 0%
...
17/07/06 19:03:03 INFO mapreduce.Job:  map 100% reduce 97%
17/07/06 19:03:15 INFO mapreduce.Job:  map 100% reduce 98%
17/07/06 19:03:27 INFO mapreduce.Job:  map 100% reduce 99%
17/07/06 19:03:39 INFO mapreduce.Job:  map 100% reduce 100%
17/07/06 19:03:47 INFO mapreduce.Job: Job job_1499374686704_0002 completed successfully
17/07/06 19:03:47 INFO mapreduce.Job: Counters: 50
        File System Counters
```

```
                FILE: Number of bytes read=6698777882
                FILE: Number of bytes written=9488445302
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=2026116230
                HDFS: Number of bytes written=2238785729
                HDFS: Number of read operations=48
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
        Job Counters
                Killed map tasks=1
                Launched map tasks=16
                Launched reduce tasks=1
                Rack-local map tasks=16
                Total time spent by all maps in occupied slots (ms)=916813
                Total time spent by all reduces in occupied slots (ms)=627806
                Total time spent by all map tasks (ms)=916813
                Total time spent by all reduce tasks (ms)=627806
                Total vcore-milliseconds taken by all map tasks=916813
                Total vcore-milliseconds taken by all reduce tasks=627806
                Total megabyte-milliseconds taken by all map tasks=938816512
                Total megabyte-milliseconds taken by all reduce tasks=642873344
        Map-Reduce Framework
                Map input records=130718954
                Map output records=130718954
                Map output bytes=2548933037
                Map output materialized bytes=2787487486
                Input split bytes=1665
                Combine input records=260799855
                Combine output records=258729171
                Reduce input groups=123904370
                Reduce shuffle bytes=2787487486
                Reduce input records=128648270
                Reduce output records=123904370
                Spilled Records=438509601
                Shuffled Maps =15
                Failed Shuffles=0
                Merged Map outputs=15
                GC time elapsed (ms)=13411
                CPU time spent (ms)=745140
                Physical memory (bytes) snapshot=4528783360
                Virtual memory (bytes) snapshot=30679724032
                Total committed heap usage (bytes)=3268935680
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=2026114565
        File Output Format Counters
                Bytes Written=2238785729
```

- Agora com apenas 2 nodes, executamos outra vez o job derrubando um DataNode no meio do reduce, afim de testar a redundância e tolerância a falhas:

```
$ hadoop jar wc.jar WordCount hdfs://master:9000/input hdfs://master:9000/output
17/07/06 19:09:34 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
17/07/06 19:09:34 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the
17/07/06 19:09:35 INFO input.FileInputFormat: Total input files to process : 1
17/07/06 19:09:35 INFO mapreduce.JobSubmitter: number of splits:15
17/07/06 19:09:35 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1499374686704_0003
17/07/06 19:09:35 INFO impl.YarnClientImpl: Submitted application application_1499374686704_0003
17/07/06 19:09:35 INFO mapreduce.Job: The url to track the job: http://HPHost02:8088/proxy/application_1499374686704_
17/07/06 19:09:35 INFO mapreduce.Job: Running job: job_1499374686704_0003
17/07/06 19:09:41 INFO mapreduce.Job: Job job_1499374686704_0003 running in uber mode : false
17/07/06 19:09:41 INFO mapreduce.Job:  map 0% reduce 0%
17/07/06 19:09:59 INFO mapreduce.Job:  map 2% reduce 0%
17/07/06 19:10:01 INFO mapreduce.Job:  map 3% reduce 0%
17/07/06 19:10:03 INFO mapreduce.Job:  map 5% reduce 0%
17/07/06 19:10:05 INFO mapreduce.Job:  map 7% reduce 0%
```

```
....
17/07/06 19:15:35 INFO mapreduce.Job:  map 100% reduce 97%
17/07/06 19:15:41 INFO mapreduce.Job:  map 100% reduce 99%
17/07/06 19:15:49 INFO mapreduce.Job:  map 100% reduce 100%
17/07/06 19:15:59 INFO mapreduce.Job: Job job_1499374686704_0003 completed successfully
17/07/06 19:15:59 INFO mapreduce.Job: Counters: 50
        File System Counters
                FILE: Number of bytes read=6698777882
                FILE: Number of bytes written=9488445302
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=2026116230
                HDFS: Number of bytes written=2238785729
                HDFS: Number of read operations=48
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
        Job Counters
                Killed map tasks=1
                Launched map tasks=16
                Launched reduce tasks=1
                Rack-local map tasks=16
                Total time spent by all maps in occupied slots (ms)=897757
                Total time spent by all reduces in occupied slots (ms)=298911
                Total time spent by all map tasks (ms)=897757
                Total time spent by all reduce tasks (ms)=298911
                Total vcore-milliseconds taken by all map tasks=897757
                Total vcore-milliseconds taken by all reduce tasks=298911
                Total megabyte-milliseconds taken by all map tasks=919303168
                Total megabyte-milliseconds taken by all reduce tasks=306084864
        Map-Reduce Framework
                Map input records=130718954
                Map output records=130718954
                Map output bytes=2548933037
                Map output materialized bytes=2787487486
                Input split bytes=1665
                Combine input records=260799855
                Combine output records=258729171
                Reduce input groups=123904370
                Reduce shuffle bytes=2787487486
                Reduce input records=128648270
                Reduce output records=123904370
                Spilled Records=438509601
                Shuffled Maps =15
                Failed Shuffles=0
                Merged Map outputs=15
                GC time elapsed (ms)=11911
                CPU time spent (ms)=687010
                Physical memory (bytes) snapshot=4446457856
                Virtual memory (bytes) snapshot=30679662592
                Total committed heap usage (bytes)=3232759808
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=2026114565
        File Output Format Counters
                Bytes Written=2238785729
```

- A tarefa foi executada com sucesso, a tolerância a falhas e a redundância foram atestadas.
- Foi possível ver que com 1 DataNode caindo no meio do mapreduce(foi desconectado o slave-1 em mais ou menos 60% do progresso) e o tempo de execução nas tarefas de map e reduce cairam. Isso não era esperado.
- Algumas possíveis explicações para isso:
  - A conexão Wi-fi utilizada estava sendo gargalo para comunicação master-slaves.
  - A redundância passou a não ser feita com apenas 1 Node, acelerando mais o processo de MapReduce.

    - A redudância é um gargalo para 3 datanodes processando a quantidade usada de dados(aproximadamente 2 GB). Quanto maior a massa de dados mais rentável se torna o processamento Hadoop.

- Afim de tentar esclarecer algumas dessas hipóteses mais experimentos foram realizados.

Testes Hadoop com cabo de rede

- Uma vez que os testes passados foram feitos com os data nodes conectados via wi-fi, percebeu-se que talvez a conectividade de rede estaria impactando nos experimentos. Para tirar essas dúvidas foram realizados experimentos com um node conectado pelo cabo de rede.

- A execução com um master e um cabo de rede teve os seguintes resultados:

```
17/07/07 17:02:11 WARN hdfs.DataStreamer: Abandoning BP-55179728-127.0.0.1-1499196379193:blk_1073742071_1248
17/07/07 17:02:11 WARN hdfs.DataStreamer: Excluding datanode DatanodeInfoWithStorage[192.168.133.193:50010,DS-3f4cc3d
17/07/07 17:02:11 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1499457518771_0001
17/07/07 17:02:11 INFO impl.YarnClientImpl: Submitted application application_1499457518771_0001
17/07/07 17:02:11 INFO mapreduce.Job: The url to track the job: http://HPHost02:8088/proxy/application_1499457518771_
17/07/07 17:02:11 INFO mapreduce.Job: Running job: job_1499457518771_0001
17/07/07 17:02:17 INFO mapreduce.Job: Job job_1499457518771_0001 running in uber mode : false
17/07/07 17:02:17 INFO mapreduce.Job:  map 0% reduce 0%
17/07/07 17:02:34 INFO mapreduce.Job:  map 1% reduce 0%
17/07/07 17:02:36 INFO mapreduce.Job:  map 3% reduce 0%
17/07/07 17:02:40 INFO mapreduce.Job:  map 6% reduce 0%
17/07/07 17:02:43 INFO mapreduce.Job:  map 7% reduce 0%
17/07/07 17:02:44 INFO mapreduce.Job:  map 10% reduce 0%
...
17/07/07 17:10:49 INFO mapreduce.Job:  map 100% reduce 97%
17/07/07 17:10:55 INFO mapreduce.Job:  map 100% reduce 98%
17/07/07 17:11:01 INFO mapreduce.Job:  map 100% reduce 99%
17/07/07 17:11:07 INFO mapreduce.Job:  map 100% reduce 100%
17/07/07 17:11:10 INFO mapreduce.Job: Job job_1499457518771_0001 completed successfully
17/07/07 17:11:10 INFO mapreduce.Job: Counters: 50
        File System Counters
                FILE: Number of bytes read=6698777882
                FILE: Number of bytes written=9488445302
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=2026116230
                HDFS: Number of bytes written=2238785729
                HDFS: Number of read operations=48
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
        Job Counters
                Killed map tasks=2
                Launched map tasks=17
                Launched reduce tasks=1
                Rack-local map tasks=17
                Total time spent by all maps in occupied slots (ms)=990826
                Total time spent by all reduces in occupied slots (ms)=460028
                Total time spent by all map tasks (ms)=990826
                Total time spent by all reduce tasks (ms)=460028
                Total vcore-milliseconds taken by all map tasks=990826
                Total vcore-milliseconds taken by all reduce tasks=460028
                Total megabyte-milliseconds taken by all map tasks=1014605824
                Total megabyte-milliseconds taken by all reduce tasks=471068672
        Map-Reduce Framework
                Map input records=130718954
                Map output records=130718954
                Map output bytes=2548933037
                Map output materialized bytes=2787487486
                Input split bytes=1665
                Combine input records=260799855
                Combine output records=258729171
                Reduce input groups=123904370
                Reduce shuffle bytes=2787487486
                Reduce input records=128648270
                Reduce output records=123904370
                Spilled Records=438509601
                Shuffled Maps =15
                Failed Shuffles=0
                Merged Map outputs=15
                GC time elapsed (ms)=13514
```

```
                CPU time spent (ms)=705660
                Physical memory (bytes) snapshot=4549906432
                Virtual memory (bytes) snapshot=30695657472
                Total committed heap usage (bytes)=3287810048
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=2026114565
        File Output Format Counters
                Bytes Written=2238785729

real    9m2.565s
user    0m6.636s
sys     0m0.360s
```

- Já desconectando um nó, obtivemos os seguintes resultados:

```
$ time hadoop jar wc.jar WordCount hdfs://master:9000/input hdfs://master:9000/output
17/07/07 17:25:40 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
17/07/07 17:25:40 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the
17/07/07 17:25:40 INFO input.FileInputFormat: Total input files to process : 1
17/07/07 17:25:41 INFO mapreduce.JobSubmitter: number of splits:15
17/07/07 17:25:41 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1499457518771_0003
17/07/07 17:25:41 INFO impl.YarnClientImpl: Submitted application application_1499457518771_0003
17/07/07 17:25:41 INFO mapreduce.Job: The url to track the job: http://HPHost02:8088/proxy/application_1499457518771_
17/07/07 17:25:41 INFO mapreduce.Job: Running job: job_1499457518771_0003
17/07/07 17:25:46 INFO mapreduce.Job: Job job_1499457518771_0003 running in uber mode : false
17/07/07 17:25:46 INFO mapreduce.Job:  map 0% reduce 0%
17/07/07 17:26:04 INFO mapreduce.Job:  map 1% reduce 0%
17/07/07 17:26:05 INFO mapreduce.Job:  map 3% reduce 0%
17/07/07 17:26:09 INFO mapreduce.Job:  map 4% reduce 0%
17/07/07 17:26:10 INFO mapreduce.Job:  map 6% reduce 0%
17/07/07 17:26:11 INFO mapreduce.Job:  map 7% reduce 0%
...
17/07/07 17:31:46 INFO mapreduce.Job:  map 100% reduce 98%
17/07/07 17:31:52 INFO mapreduce.Job:  map 100% reduce 99%
17/07/07 17:31:54 INFO mapreduce.Job:  map 100% reduce 100%
17/07/07 17:32:05 INFO mapreduce.Job: Job job_1499457518771_0003 completed successfully
17/07/07 17:32:05 INFO mapreduce.Job: Counters: 50
        File System Counters
                FILE: Number of bytes read=6698777882
                FILE: Number of bytes written=9488445302
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=2026116230
                HDFS: Number of bytes written=2238785729
                HDFS: Number of read operations=48
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
        Job Counters
                Killed map tasks=1
                Launched map tasks=16
                Launched reduce tasks=1
                Rack-local map tasks=16
                Total time spent by all maps in occupied slots (ms)=895788
                Total time spent by all reduces in occupied slots (ms)=296822
                Total time spent by all map tasks (ms)=895788
                Total time spent by all reduce tasks (ms)=296822
                Total vcore-milliseconds taken by all map tasks=895788
                Total vcore-milliseconds taken by all reduce tasks=296822
                Total megabyte-milliseconds taken by all map tasks=917286912
                Total megabyte-milliseconds taken by all reduce tasks=303945728
        Map-Reduce Framework
                Map input records=130718954
                Map output records=130718954
                Map output bytes=2548933037
                Map output materialized bytes=2787487486
                Input split bytes=1665
```

```
                Combine input records=260799855
                Combine output records=258729171
                Reduce input groups=123904370
                Reduce shuffle bytes=2787487486
                Reduce input records=128648270
                Reduce output records=123904370
                Spilled Records=438509601
                Shuffled Maps =15
                Failed Shuffles=0
                Merged Map outputs=15
                GC time elapsed (ms)=11834
                CPU time spent (ms)=687830
                Physical memory (bytes) snapshot=4460859392
                Virtual memory (bytes) snapshot=30680518656
                Total committed heap usage (bytes)=3253731328
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=2026114565
        File Output Format Counters
                Bytes Written=2238785729

real    6m26.887s
user    0m6.208s
sys     0m0.304s
```

- Percebe-se novamente um tempo de execução menor com a queda de um DataNode. Portanto a conectividade não foi o fator que impactou nos experimentos passados.
- Portanto, há duas hipoteses restantes:
    - A redundância passou a não ser feita com apenas 1 Node, acelerando mais o processo de MapReduce.
    - A redudância é um gargalo para 3 datanodes processando a quantidade usada de dados(aproximadamente 2 GB). Ou seja, quanto maior a massa de dados mais rentável se torna o processamento Hadoop.
- Bem provavelmente as duas estão certas, pois o hadoop foi criado para trabalhar com grandes quantidades de dado em um sistema altamente clusterizado. Logo, quanto mais dados, maior será a diferença testada, e mais eficiente será o esquema de clusters.
- Por questões de estrutura bem como de quantidade de dados, não foi possível realizar outros testes possíveis.
- Alguns experimentos que revelariam reforçariam nossas hipóteses:
    - Testar com uma ordem de 100 GB de dados.
    - Testar com pelo menos 10 nós com 100 GB de dados.
    - Testar numa mesma infra-estrutura com redundância de nível 3, 4 ou mais, para ver qual gargalo disso.
    - Testar com blocos de dados menores e maiores e monitorar como isso impacta no desempenho.

Versão simples do contador de palavra.

Esse experimento é simplesmente para mostrar que na visão tradicional de programação é inviável operar com tantos volumes de dados, seja por suporte de memória ou por questões de tempo. Nos subtópicos abaixo é perpassado essa visão.

Versão que armazena os resultados em memória

Contador para arquivos pequenos

Para arquivos contendo pouca palavras, o contador funciona bem, e não apresenta erro algum. Para processar entradas pequenas ele seria mais indicado do que utilizar o Hadoop.

Processando arquivo grande

Para processar grandes arquivos, esse programa não funciona conforme o esperado. Foi lançada uma excessão de falta de memória ao testarmos um input de 2GB. Dessa forma, percebe-se a necessidade de salvar os dados que estão sendo processados em disco. Portanto, resolvemos utilizar o MongoDB para armazenar os dados. O resultado desse teste será descrito mais abaixo.

A excessão gerada foi a seguinte:

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.util.Arrays.copyOfRange(Arrays.java:3664)
        at java.lang.String.<init>(String.java:207)
        at java.io.BufferedReader.readLine(BufferedReader.java:356)
        at java.io.BufferedReader.readLine(BufferedReader.java:389)
        at SimpleWordCounter.main(SimpleWordCounter.java:13)
```

Código em Java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.*;


class SimpleWordCounter {

  public static void main(String args[]) throws IOException {
      Hashtable<String, Integer> words = new Hashtable<String, Integer>();
      BufferedReader br = new BufferedReader(new FileReader("test.txt"));

      // For each line in the file
      for (String line; (line = br.readLine()) != null;) {

          // Check if word already exists,
          Integer n = words.get(line);
          if (n == null) {
              // If word does not exists, save 1.
              words.put(line, 1);
          } else {
              // If word does not exists, update the value.
              words.put(line, n + 1);
          }
      }

      FileWriter fw = new FileWriter("output.txt");
      BufferedWriter bw = new BufferedWriter(fw);
      // Save each word to file
      for (String key : words.keySet()) {
          bw.write(key + "  " + words.get(key) + "\n");
      }
      bw.close();

  }
}
```

Versão salvando o conteúdo em disco (MongoDB)

Contador para arquivos pequenos

Para arquivos contendo pouca palavras, o contador funciona bem, e não apresenta erro algum. Porém é mais lento do que a versão simples que salva todos os resultados em memória.

Processando arquivo grande

Ele processou os dados durante muito tempo sem gerar erro algum. Quando resolvi medir o tempo de execução pela segunda vez, desisti pois estava demorando muito. Porém, aparentemente, ele estava funcionando corretamente. Aos 25 minutos de execução, o banco de dados continha 0.836GB de dados

```
> show dbs;
admin  0.000GB
local  0.000GB
test   0.836GB
```

Aos 30 minutos, passou a ter 0.965GB de dados:

```
> show dbs;
admin  0.000GB
local  0.000GB
test   0.965GB
```

Dessa forma, percebe-se que demoraria mais de uma hora para processar 2GB de dados. Portanto, resolvi interromper o experimento.

Código em Java

Observação: foi utilizado o seguinte driver para utilizar MongoDB com java:

- mongo-java-driver-2.13.3.jar

```
import com.mongodb.BasicDBObject;
import com.mongodb.BulkWriteOperation;
import com.mongodb.BulkWriteResult;
import com.mongodb.Cursor;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.DBCursor;
import com.mongodb.DBObject;
import com.mongodb.MongoClient;
import com.mongodb.ParallelScanOptions;
import com.mongodb.ServerAddress;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.*;


public class MongoWordCounter {
        public static void main(String args[]) {

                long startTime = System.currentTimeMillis();

            try {
                final MongoClient mongo = new MongoClient( "localhost" , 27017 );

                final DBCollection col =  mongo.getDB("test").getCollection("testKeywordCounter");
                // Clean Collection
                col.drop();


                BufferedReader br = new BufferedReader(new FileReader("/Users/alexandretk/Desktop/projeto_Final_FSD/1
                //BufferedReader br = new BufferedReader(new FileReader("test.txt"));

                for (String line; (line = br.readLine()) != null;) {

                        // Create a new object with (_id = word, value = qnt)
                        BasicDBObject data = new BasicDBObject();
                        data.append("_id", line);
                        data.append("value", 1);

                        // Query the database to check if there is already a record.
                        BasicDBObject whereQuery = new BasicDBObject();
                        whereQuery.put("_id", line);
                        DBObject objectFound = col.findOne(whereQuery);
                        // If there is a record update it's value
                        if(objectFound != null) {
                                DBObject update = new BasicDBObject(
                                        "$inc", new BasicDBObject("value", 1)
                                    );
```

```
                        col.update(objectFound, update);
                    // Else insert to the collection
                    } else {
                        col.insert(data);
                    }
                }

                // Print all entries in the db.
                DBCursor cursor = col.find();
                while(cursor.hasNext()) {
                    System.out.println(cursor.next());
                }

            }
            catch (Exception e) {
              e.printStackTrace();
            }

        // Prints the time it took to run.
        long endTime = System.currentTimeMillis();
        System.out.println("It took " + (endTime - startTime) + " milliseconds");
    }
}
```

Observação

As duas versões se demostraram ineficientes. Seria necessário paralelizar mais o processamento para que o resultado fosse satisfatório. Para solucionar este problema o Hadoop é altamente recomendado, percebe-se até mesmo pelos resultados obtidos no capitulo anterior(Testando Hadoop), que foram extremamente melhores, até mesmo no caso de um único Node em execução.

Conclusão do Relatório

Ao longo do desenvolvimento deste projeto, foi possível perceber o poder do Hadoop para resolver uma série problemas recorrentes no contexto de Big Data, principalmente, a partir da versão 2.0 com a introdução do YARN. Mesmo antes da introdução do YARN, o conjunto de MapReduce com HDFS era algo bastante poderoso, e que podia resolver uma série de problemas comuns nesse contexto. Com a implementação do YARN, essa grande quantidade de aplicações se tornou algo muito maior, visto que essa mudança permitiu o framework do Hadoop utilizar outros frameworks de processamento, além de programas que não são baseados no paradigma MapReduce.

Entender o Hadoop traz um suporte poderoso para lidar com grandes quantidades de dados em qualquer contexto computacional. As possibilidades de experimentação e uso ainda são muito grandes, seria praticamente impossível abordar todo potencial da ferramenta, ainda que este relatório foi contemplado o entendimento básico, o que é expansível para uma variedade de soluções que se encaixam nos problemas de Map Reduce.

Portanto, esse projeto foi bastante proveitoso e permitiu com que os alunos tivessem um contato inicial com Big Data e Data Mining, facilitando que futuros projetos sejam desenvolvidos nesta área. Portanto, além de interessante, essa experiência será bastante útil, visto que esta área está crescendo rapidamente na atualidade.