Sistemas de Computação em Cloud

Relatório do Trabalho Prático 2

André Lopes n.º 45617
Nelson Coquenim n.º 45694
João Marques n.º 48500

7, Dezembro, 2018

Índice

1 Resumo

Este projecto tem como objectivo servir de introdução ao ecossistema Hadoop na *cloud*. Obteve-se dados de um *crawler* em formato WARC/WET, os quais foram processados usando MapReduce de forma a obter a *performance* da rede durante a extracção e as palavras mais frequentes nos diversos *sites*. Para além do desenvolvimento da solução, analisou-se a *performance* deste processamento variando o número e especificação dos *workers*. Por fim, os resultados permitiram demonstrar a escalabilidade horizontal desta solução.

2 Introdução

Este relatório incide sobre o segundo trabalho prático da cadeira de Sistemas de Computação em *Cloud*. O presente trabalho prático tem como objectivo servir de introdução ao ecossistema Hadoop na *cloud*, neste caso no Azure. Neste serviço de *cloud* está disponível o HDInsight. Este produto fornece a possibiliade de criar um *cluster* Hadoop de dimensão parametrizável, removendo a necessidade de instalação do *cluster*. Da introdução ao Hadoop surge naturalmente a necessidade de entender o paradigma de programação MapReduce, o que será acompanhado de uma análise experimental do mesmo.

O Hadoop é uma *framework* para armazenamento distribuído e processamento de grandes volumes de dados recorrendo ao modelo de programação MapReduce. Este modelo consiste, como o nome indica, fundamentalmente em duas funções: *map* e *reduce*. Abstrai-se assim a arquitectura do sistema, o que permite um maior foco na lógica da aplicação.

Em relação à estrutura do presente relatório, começar-se-á por descrever o problema. Após esta apresentação será descrito cada um dos processamentos a fazer. Dentro deste contexto, será apresentada a solução através do pseudo-código e uma breve explicação do mesmo. Por fim, serão apresentadas as condições experimentais, os resultados e a discussão dos mesmos.

3 Descrição do problema

O problema consiste fundamentalmente em processar informação obtida através de um *crawler*. Este produz ficheiros WARC/WET que representam o texto das páginas Web visitadas pelo mesmo, removendo, por exemplo, as *tags* do HTML. Esta informação e o seu processamento são úteis em diversas áreas. A título de exemplo, uma plataforma de comércio online pode ajustar preços de

acordo com os disponíveis no *site* da concorrência [1]. Neste projecto não se é obviamente tão ambicioso, pois o mais importante não são os dados a processar mas sim analisar a forma como é feito o processamento. O processamento destes ficheiros WARC/WET terá dois fins: analisar a *performance* da rede durante a extracção e encontrar o *top* 10 das palavras mais usadas. Quanto à implementação, esta foi feita em Java recorrendo à API do Hadoop. Outro detalhe de implementação é o facto do *parser* do ficheiro WARC/WET ter sido fornecido à partida.

3.1 *Performance* da rede

No final do presente processamento, ter-se-á informação como o número de bytes(NB) extraídos, a duração dessa extracção(T) e a *bandwith*(NB/T) para cada *site*.

3.1.1 Solução

Este *job* MapReduce é bastante simples, sendo que é importante realçar a forma como é calculado o tempo de extracção. Assume-se que, caso existam várias ocorrências do mesmo *site* no ficheiro, o tempo de extracção é igual à diferença entre as datas mais distantes. Caso apenas exista uma ocorrência, não é possível calcular nem o tempo de extracção nem a *bandwith*, marcando-se como não-definido.

Ao executar este *job* com o método anteriormente referido para calcular o tempo de extracção, notou-se que muitos dos *sites* ficavam com a *bandwith* não definida ou com valores absurdamente baixos. Dado que o *dataset* não contém o tempo de extracção, esta foi a solução encontrada. À partida esta não permite tirar conclusões fidedignas quanto à *performance* do *crawler* para cada um dos *sites*.

```
class MAPPER
        Map(recordID, warcRecord)
            bytes ← warcRecord.contentLength
            extractionDate ← warcRecord.extractionDate
            site ← warcRecord.url
            EMIT(site, (bytes,extractionDate) )
```

```
class REDUCER
    Reduce(site, collection [(bytes, extractionDate), ...] )
        extractionDates ← OrderedSet
        sumBytes ← 0
        for all (bytes, extractionDate) ∈ collection do
            extractionDates ← extractionDates U {(bytes, extractionDate)}
            sumBytes ← sumBytes + bytes
        if extractionDates > 1
            extractionTime ← extractionDates.last -
                                extractionDates.first
            bandwith ← extractionTime / sumBytes
        else
            extractionTime ← ⊥
            bandwith ← ⊥
        EMIT(site, (sumBytes, extractionTime, bandwith) )
```

3.2 *Top* 10 palavras

No final deste processamento, ter-se-á as *top* 10 palavras mais frequentes com mais de 5 caracteres acompanhadas da frequência absoluta. Na verdade só se fará este processamento para os *sites* que apenas contenham conteúdos no alfabeto Latino e que estejam incluídos nos 10 *sites* mais volumosos, ou seja, cujo o *crawler* extraiu maior número de *bytes*.

3.2.1 Solução

Para este processamento é necessário executar uma cadeia de *jobs*: LatinSites-NetPerformance → TopHeaviestSites → WordCount → TopPopularWords

LatinSitesNetPerformance Este *job* executa dois processamentos semanticamente distintos. Por um lado, irá filtrar todos os *sites* que contenham conteúdo em que o alfabeto não seja o Latino. Por outro lado, irá computar o número total de *bytes* extraídos de cada um dos *sites*. A razão desta solução é o facto de evitar que se execute dois *jobs* distintos, o que iria envolver escrever mais resultados intermédios em disco.

Ao observar os resultados deste *job*, todos os caracteres eram do alfabeto Latino. Funcionando este facto como um argumento informal de correcção.

```
class MAPPER
    Map(recordID, warcRecord)
        bytes ← warcRecord.contentLength
        site ← warcRecord.url
        content ← warcRecord.content
        content ← content.removePunctuation
        if isLatinAlphabet(content)
            EMIT(site, (bytes, content) )
```

```
class COMBINER // equivalent to REDUCER

class REDUCER
        Reduce(site, collection [(bytes, content), ...] )
        contentAccumulator ← ""
        sumBytes ← 0
        for all (bytes, content) ∈ collection do
            contentAccumulator ← contentAccumulator + content
            sumBytes ← sumBytes + bytes
        EMIT(site, (sumBytes, contentAccumulator) )
```

TopHeaviestSites Este *job* tem como objectivo o processamento do *top* 10 dos *sites* em que foram extraídos um maior número de *bytes*. Em relação ao método Map, este para cada *site* irá adicionar ao mapa a entrada (bytes, (site, bytes, content)). É importante referir que o mapa é ordenado por chave, ou seja, por quantidade de *bytes*. Após todos os *maps* é chamada a função CleanUp() que emitirá os 10 *sites* mais volumosos processados por aquele *mapper* com a chave a $\perp$. A intenção desta chave é que apenas exista um *reducer*. Este receberá os vários *top* 10 dos diferentes *mappers* e irá ordená-los e emitir os 10 *sites* mais volumosos.

```
class MAPPER

    topBytesMap ← OrderedMap(bytes, (site, bytes, content)) //orders by bytes
    topSize ← 10

    Map(site, (bytes, content) )
        topBytesMap ← topBytesMap + (bytes, (site, bytes, content))
        if topBytesMap > topSize
            //removes elements to have only 10 elements
            cutToTopSize(topBytesMap)

    Cleanup() //runs after all maps
        for all (bytes, (site, bytes, content)) ∈ topBytesMap do
            EMIT(⊥, (site, bytes, content))
```

```
//only one reducer
class REDUCER

    topBytesMap ← OrderedMap(bytes, (site, bytes, content)) //orders by bytes
    topSize ← 10

    Reduce(⊥, collection [(site, bytes, content),...])
        for all (site, bytes, content) ∈ collection do
            topBytesMap ← topBytesMap + (bytes, (site, bytes, content))
        cutToTopSize(topBytesMap)
        for all (bytes, (site, bytes, content)) ∈ topBytesMap do
            EMIT(site, (bytes, content))
```

WordCount Em relação a este *job*, é o clássico WordCount, com o pequeno detalhe de que irá filtrar as palavras cujo tamanho seja menor ou igual a 5. Por outro lado, irá emitir as palavras com todas as letras minúsculas.

```
class MAPPER
    Map(site, (bytes, content))
        for all word ∈ content do
            if word.length > 5
                word ← word.toLowerCase()
                EMIT(word, 1)
```

```
class COMBINER //equal to REDUCER

class REDUCER
    Reduce(word, collection [count, ...] )
        sum ← 0
        for all count ∈ collection do
            sum ← sum + count
        EMIT(word, sum)
```

TopPopularWords Por fim, o *job* final que de forma muito semelhante ao TopHeaviestSites calculará o *top* 10 das palavras mais frequentes.

```
class MAPPER

    //orders by frequency of word
    topWordsMap ← OrderedMap(count, (word, count))
    topSize ← 10

    Map(word, count)
        topWordsMap ← topWordsMap + (count, (word, count))
        if topWordsMap > topSize
            cutToTopSize(topWordsMap)

    Cleanup() //runs after all maps
        for all (count, (word, count)) ∈ topWordsMap do
            EMIT(⊥, (word, count))
```

```
class REDUCER

    //orders by frequency of word
    topWordsMap ← OrderedMap(count, (word, count))
    topSize ← 10

    Reduce(⊥, collection [(word, count),...])
        for all (word, count) ∈ collection do
            topWordsMap ← topWordsMap + (count, (word, count))
        cutToTopSize(topWordsMap)
        for all (count, (word, count) ∈ topWordsMap do
            EMIT(word, count)
```

4 Resultados

4.1 Condições Experimentais

A avaliação experimental foi realizada em 6 *clusters* diferentes, as especificações técnicas de cada um estão listadas na tabela 1. Todos os *clusters* possuem 2 *head nodes* com as mesmas especificações técnicas, 4 vCPUs e 14 GB de RAM para cada *head node*.

Todos os *clusters* foram criados ao mesmo tempo e os testes executados em paralelo, optámos por esta metodologia pois os testes demoravam um tempo significativo a executar.

De forma a automatizar parte do processo de teste foram criados dois *scripts* em bash, `setupTestEnv.sh` e `testQuery.sh`. O script `setupTestEnv.sh` tem a função de configurar o ambiente de teste, cria uma pasta onde serão guardados

Cluster	Workers	vCPUs	RAM
C_0	1	4	14 GB
C_1	2	8	28 GB
C_2	4	16	56 GB
C_3	8	32	112 GB
C_4	16	64	224 GB
C_5	1	32	448 GB

Tabela 1: Especificações técnicas dos *clusters* utilizados.

os resultados, faz download de um ficheiro WARC, upload do ficheiro para o HDFS e, finalmente, copia o ficheiro trinta e uma vezes através do comando `hdfs dfs -cp`. No total ficam a existir trinta e duas cópias do ficheiro WARC para os testes de *MapReduce*. Não existe a opção de efectuar *symbolic links* no hdfs, obrigando assim a realizar cópias do ficheiro original. Isto tornou a operação de configurar o ambiente de testes muito demorada, demorando cada cópia em média dois minutos, perfazendo um total de mais de uma hora para configurar cada *cluster*.

O segundo *script*, `testQuery.sh`, tem a função de executar a *query performance* da rede com 1, 2, 4, 8, 16 e 32 ficheiros de *input*. Para cada número de ficheiros de *input* a *query* é executada três vezes, sendo que no final é feita a média dos três tempos e esse é o valor registado.

4.2 Resultados

Inicialmente procedeu-se à avaliação da primeira *query*, *performance* da rede, sob o conjunto de ficheiros composto por: um ficheiro (0.5GB); dois ficheiros (1GB); quatro ficheiros (2GB); oito ficheiros (4GB) e dezasseis ficheiros (8GB). Sendo que os resultados obtidos apresentam-se na Figura 1. Estes revelam tempos de execução semelhantes entre as várias configurações de *workers*, bem como a ausência de um melhoramento significativo do tempo de execução tendo em conta o aumento no número de *workers*. Estas duas observações podem sugerir que os ficheiros utilizados são demasiados pequenos para compensar o esforço de sincronização dos workers. Desta forma, efectuou-se outro conjunto de testes utilizando como base um ficheiro maior, de quatro *gigabytes*.

O segundo grupo de testes ao tempo de execução da *query performance* da rede foi efectuado de forma idêntica ao grupo anterior excepto que os ficheiros

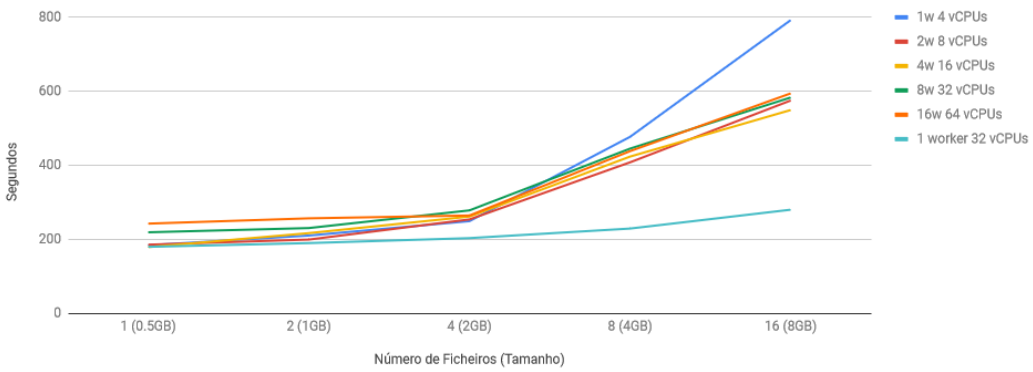

Figura 1: Tempo de execução da primeira *query*, *performance* da rede, sob o conjunto de ficheiros de tamanho entre quatro *gigabytes* e oito *gigabytes*. No eixo das abcissas varia-se o número de ficheiros e por consequência o tamanho total do *input* e no eixo das ordenadas observa-se o tempo de execução da *query*. Cada recta corresponde a um número de *workers* diferente.

.

utilizados possuem um tamanho maior. Os resultados obtidos dos mesmos encontram-se na Figura 2.

Contrariamente aos resultados obtidos nos testes anteriores, é possível observar uma redução do tempo de execução com o aumento do número de *workers*. Adicionalmente verifica-se que para um dado número de *workers* o tempo de execução mantém-se, aproximadamente, constante até o número de vCPUs corresponder ao número de ficheiros. Isto porque maximiza-se a utilização dos *cores* de um nó, através da atribuição de um ficheiro por *core*, promovendo o uso de todos os cores do mesmo. Após este ponto, o tempo de execução cresce proporcionalmente com o aumento da carga de trabalho uma vez que não se consegue paralelizar mais a execução do *map-reduce job*. De salientar que o facto de ser impossível proceder à divisão de um dado ficheiro de *input* durante a execução de um *map-reduce job* possibilita a inutilização de alguns dos *cores* de um nó em situações onde existem mais *cores* do que ficheiros.

No decorrer destes testes foi possível constatar que o número de *mappers* efectuados é directamente proporcional ao número de ficheiros de *input* após a divisão dos mesmo quando possível. Sendo assim e como neste não era possível dividir os ficheiros de *input* numa situação com 16 ficheiros o número de *mappers* criados é igual mesmo quando o número de CPUs é inferior. No entanto, o número de *mappers* executados de forma paralela é sempre limitado pelo número de CPUs.

Finalmente, tendo em conta os resultados obtidos no conjunto de testes anterior, é possível traçar um gráfico que demonstre a evolução do tempo de execução

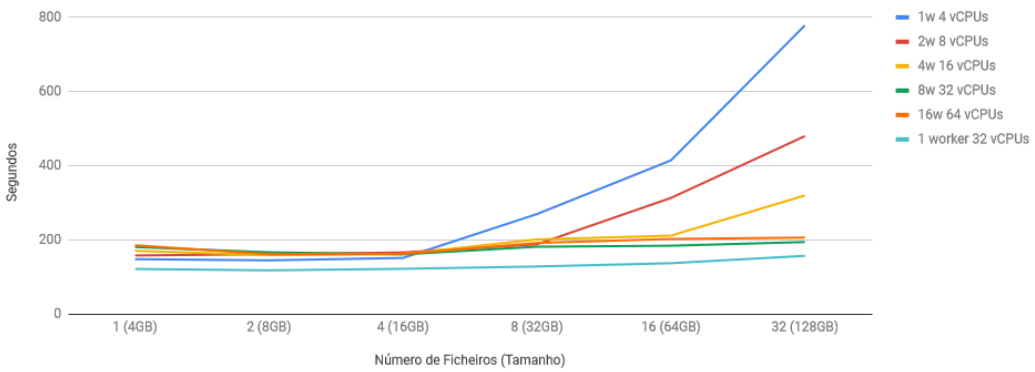

Figura 2: Tempo de execução da primeira *query*, *performance* da rede, sob o conjunto de ficheiros de tamanho entre quatro *gigabytes* e cento e vinte e oito *gigabytes*. No eixo das abcissas varia-se o número de ficheiros e por consequência o tamanho total do *input* e no eixo das ordenadas observa-se o tempo de execução da *query*. Cada recta corresponde a um número de *workers* diferente.

com aumento proporcional do número de *cores* com o número de ficheiros (Figura 3). Este gráfico permite concluir que o *map-reduce job* da primeira *query* escala de forma, aproximadamente, linear. Uma vez que o aumento progressivo do conjunto de dados bem como dos recursos de processamento origina um tempo de execução, relativamente, constante.

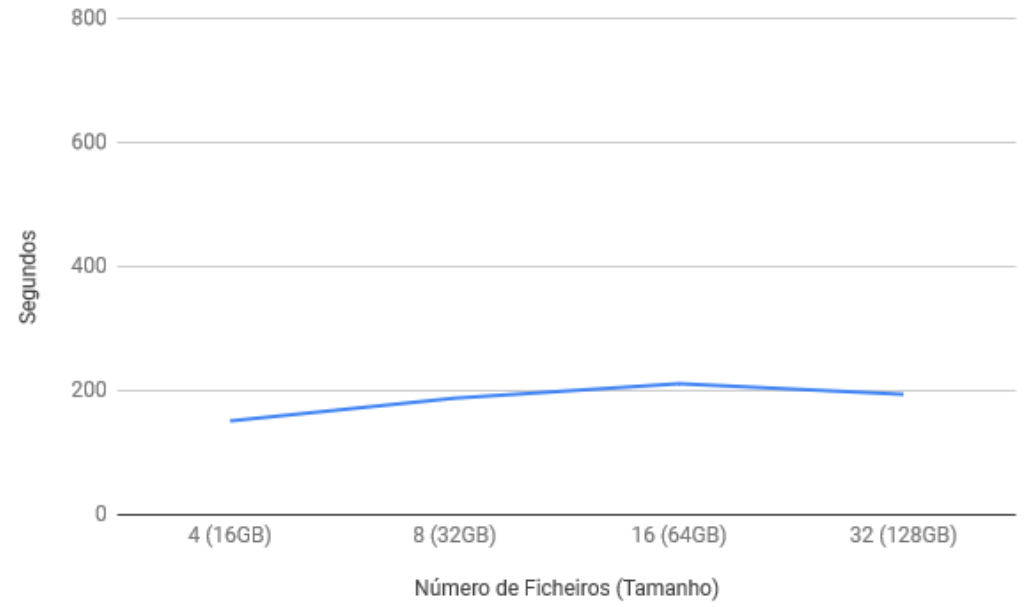

Figura 3: Tempo de execução, em segundos, da primeira *query* variando o número de ficheiros de forma directamente proporcional com o número de CPUs.

5 Conclusão

No final do presente relatório importa retirar conclusões, o facto de se ter recorrido a serviços de *cloud* facilitou as experiências a ser feitas. Usar um produto como o HDInsight permite criar um *cluster* Hadoop, com uma dada arquitectura, em poucos minutos. Evitando-se assim a instalação do *cluster*, o que é ideal para os testes que foram feitos, nos quais era necessário variar o número de *workers* e a especificação dos mesmos.

Na verdade, apesar do MapReduce abstrair o programador da arquitectura do sistema, o desenvolvimento torna-se complicado devido à existência de, basicamente, duas funções. Sendo que programar algo como o processamento de um *top* 10 não é trivial.

Efectivamente, consegue-se obter uma escalabilidade horizontal usando esta *framework*. E a chave para este feito é a localidade dos dados, ou seja, não são os dados que vão à computação, mas sim a computação aos dados. Resolvendo-se os problemas de *bandwith*.

Para trabalho futuro, seria interessante comparar a *performance* entre o MapReduce e o Spark. Uma das vantagens desta última *framework* é que o desenvolvimento da solução para processar as palavras mais frequentes seria mais simples. Outra das vantagens, para alguns tipos de aplicação, é o facto da computação ser feita em memória não sendo necessário ler e escrever do disco resultados intermédios.

Referências

[1] *Web Crawler Use Cases*. URL: `https://www.promptcloud.com/web-crawl-use-cases` (acedido em 07/12/2018).