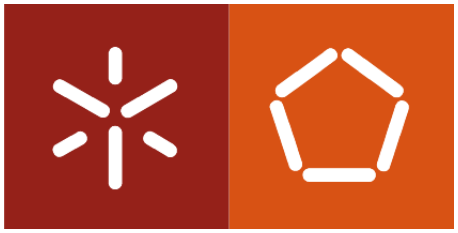

UNIVERSIDADE DO MINHO
Mestrado Integrado em Engenharia Informática

Gestão de Grandes Conjuntos de Dados

Trabalho Prático - Relatório de Desenvolvimento

Carlos Gonçalves (A77278)
José Ferreira (A78452)
Ricardo Peixoto (A78587)

25 de Maio de 2019

Conteúdo

Lista de Figuras

Lista de Tabelas

Resumo

O objetivo deste trabalho era a conceção, estruturação e implementação de uma solução para um problema especifico na área de *data analytics/bigdata*. Foi desenvolvida uma API web à qual se pode efetuar diversas perguntas sobre um *dataset* escolhido na área do tráfego aéreo, de maneira a encontrar soluções para problemas como o congestionamento,atrasos,entre outros, utilizando tecnologias como o *Spark, Hbase e HDFS.*

1 Introdução

O objetivo deste trabalho passava por definir um problema relacionado com a temática de *Big Data* e implementar uma solução do mesmo utilizando as tecnologias apropriadas para o efeito. Decidimos escolher a área do tráfego aéreo por ser de uma dimensão enorme e por gerar uma quantidade massiva de dados diariamente. Mais especificamente escolhemos um *dataset* que contem várias informações sobre voos realizados em diferentes aeroportos no ano de 2008. Em seguida apresentamos as questões formuladas que tentamos responder com a nossa solução e que nos permitem retirar informação útil para a área do tráfego aéreo a partir da análise dos dados presentes no *dataset* escolhido.

1. Quais os aviões que fizeram mais kms num ano?
2. Qual o avião com mais voos?
3. Quais os aeroportos com um maior numero de voos/cancelados/desviados?
4. Dias do mês com mais afluência?
5. Dias da semana com mais afluência?
6. Quais os aviões que partem/chegam com mais atraso?
7. Por aeroporto, qual a hora do dia mais movimentada?
8. Qual o tipo de atraso mais significativo, em termos totais e de número de atrasos?
9. Quais as informações de um determinado voo, por identificador/data?
10. Quais as informações de um voo em específico?
11. Quais as informações totais de um voo em específico?

Estas perguntas vão-nos permitir identificar alguns problemas existentes neste domınio e poderá ser importante para perceber que medidas deverão ser tomadas de maneira a otimizar o tráfego aéreo, diminuindo atrasos, tempos de espera, cancelamentos, desvios, entre outros. Uma aplicação concreta possıvel poderá passar por tentar distribuir melhor o numero de voos num aeroporto ao longo de todo o dia, de maneira a evitar horas de confusão excessiva ou colocar mais voos a uma certa hora.

2 Análise e tratamento dos dados

O Dataset original possui uma dimensão de 7009728 linhas por 29 colunas. De seguida é possível observar as colunas e o significado de cada uma delas:

	Name	Description
1	Year	1987-2008
2	Month	1-12
3	DayofMonth	1-31
4	DayOfWeek	1 (Monday) - 7 (Sunday)
5	DepTime	actual departure time (local, hhmm)
6	CRSDepTime	scheduled departure time (local, hhmm)
7	ArrTime	actual arrival time (local, hhmm)
8	CRSArrTime	scheduled arrival time (local, hhmm)
9	UniqueCarrier	unique carrier code
10	FlightNum	flight number
11	TailNum	plane tail number
12	ActualElapsedTime	in minutes
13	CRSElapsedTime	in minutes
14	AirTime	in minutes
15	ArrDelay	arrival delay, in minutes
16	DepDelay	departure delay, in minutes
17	Origin	origin IATA airport code
18	Dest	destination IATA airport code
19	Distance	in miles
20	TaxiIn	taxi in time, in minutes
21	TaxiOut	taxi out time in minutes
22	Cancelled	was the flight cancelled?
23	CancellationCode	reason for cancellation (A = carrier, B = weather, C = NAS, D = security)
24	Diverted	1 = yes, 0 = no
25	CarrierDelay	in minutes
26	WeatherDelay	in minutes
27	NASDelay	in minutes
28	SecurityDelay	in minutes
29	LateAircraftDelay	in minutes

Figura 1: Dimensão do dataset

Inicialmente começamos por analisar o significado de cada variável individualmente e verificar se davam informações importantes às perguntas que pretendíamos responder. Com esta análise acabamos por excluir três variáveis: *CancellationCode*, *TaxiIn* e *TaxiOut*.

De seguida verificamos as percentagens de valores nulos em cada uma das restantes variáveis e obtivemos os seguintes resultados:

LateAircraftDelay	0.782483
NASDelay	0.782483
WeatherDelay	0.782483
CarrierDelay	0.782483
SecurityDelay	0.782483
AirTime	0.022069
ActualElapsedTime	0.022069
ArrDelay	0.022069
ArrTime	0.021634
DepTime	0.019437
DepDelay	0.019437
TailNum	0.011893
CRSElapsedTime	0.000120
Origin	0.000000
Dest	0.000000
Distance	0.000000
FlightNum	0.000000
UniqueCarrier	0.000000
CRSArrTime	0.000000
Cancelled	0.000000
CRSDepTime	0.000000
Diverted	0.000000
DayOfWeek	0.000000
DayofMonth	0.000000
Month	0.000000
Year	0.000000

Figura 2: Percentagens de valores nulos

Como podemos observar na tabela acima, todas as variáveis apresentam percentagens de valores nulos muito reduzidas com a exceção das primeiras cinco: *LateAircraftDelay*, *NASDelay*, *WeatherDelay*, *CarrierDelay* e *SecurityDelay*. Tendo em conta que estas variáveis retratam atrasos, em minutos, decidimos considerar todos os casos de valores nulos como situações em que não houveram atrasos de modo que substituímos todos estes valores nulos pelo valor "0".

Por último, como o dataset não possuía um identificador que identificasse unicamente cada linha do dataset. Originalmente pensamos em usar o *FlightNum*, contudo este identifica uma linha aérea particular entre uma origem e um destino pelo que existem ou podem existir inúmeros voos com o mesmo *FlightNum*. Deste modo, a solução encontrada pelo grupo foi criar uma nova coluna (key), recorrendo à agregação de quatro colunas distintas do dataset (*FlightNum*, *Year*, *Month* e *DayofMonth*) com o seguinte formato: **Key = FlightNum::Year/Month/DayofMonth**. Após adicionarmos esta noca coluna ao dataset, as quatro colunas que lhe deram origem deixaram de ser úteis pelo que foram também excluídas.

3 Solução Proposta

Nesta secção vamos falar um pouco das arquitecturas que utilizamos ao longo do desenvolvimento da aplicação, apresentando depois a arquitectura final.

3.1 1º Versão

A arquitectura da primeira versão está descrita na figura 3, onde podemos verificar que utilizamos **HBase** para armazenar os dados, e **Apache Spark** para realizar o processamento dos mesmos.

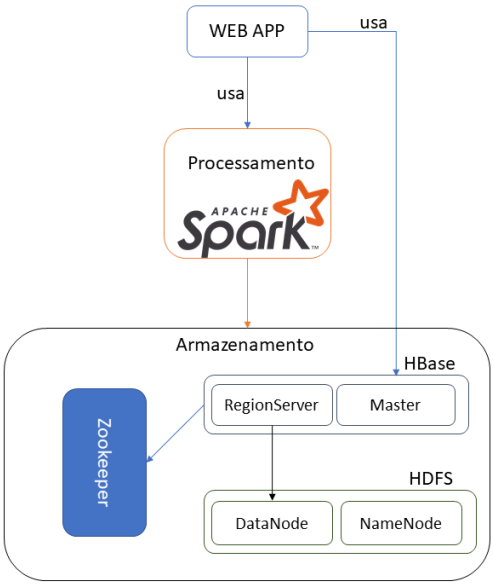

Figura 3: Arquitetura da solução

Podemos também verificar que no processo de processamento existe uma ligação entre o **Spark** e o **HBase**, sendo que os dados são "diretamente"passados do **HBase** para um *JavaRDD* sendo possível realizar o processamento de uma forma mais eficiente. Existe também uma ligação direta da aplicação ao **HBase** no caso de interrogações objectivas, especificamente um *GET*. É ainda possível verificar que o **HBase** armazena os dados no **HDFS** (como era de esperar). Nesta arquitetura temos apenas 1 *regionserver* pelo que apenas temos 1 local diferente onde os dados são armazenados.

Nesta primeira versão a tabela foi criada apenas com uma família de colunas, que contém todas as colunas anteriormente referidas. A tabela foi criada com o seguinte comando, directamente no *regionserver*:

```
1 create 'trafego', 'data'
```

O processo de transferência dos dados realizado foi um *Bulk import*[1], com o auxílio da funcionalidade fornecida pelo **HBase**, sendo o seguinte comando:

```
1 hbase org.apache.hadoop.hbase.mapreduce.ImportTsv -Dimporttsv.separator=, -Dimporttsv.columns=HBASE_ROW_KEY,
    data:DayOfWeek,data:DepTime,data:CRSDepTime,data:ArrTime,data:CRSArrTime,data:UniqueCarrier,data:TailNum
    ,data:ActualElapsedTime,data:CRSElapsedTime,data:AirTime,data:ArrDelay,data:DepDelay,data:Origin,data:
    Dest,data:Distance,data:Cancelled,data:Diverted,data:CarrierDelay,data:WeatherDelay,data:NASDelay,data:
    SecurityDelay,data:LateAircraftDelay,data:Dest,data:Distance,data:TailNum trafego hdfs://namenode-1.vnet
    :8020/tmp/final3.csv
```

3.2 2º Versão

A arquitectura da segunda versão é bastante parecida à da primeira versão, utilizando exactamente os mesmos componentes. A única diferença relativamente à versão anterior está na organização das colunas na tabela da Base de Dados.

Como o **HBase** guarda todas as colunas da mesma família de colunas no mesmo ficheiro a leitura é menos demorada se as colunas estiverem organizadas por famílias, pelo facto de conter menos colunas para ler. No entanto quando se quer ler colunas de mais do que uma família, existe uma penalização por se ler de vários ficheiros(possivelmente).

Nesta versão dividimos a tabela em diferentes famílias de colunas, para poder responder mais rapidamente às interrogações. Decidimos dividir a tabela nas seguintes famílias de colunas:

- Aeroportos - informações relativas aos aerportos, importante para responder às interrogações 3 e 7
- InfoAviao - informações relativas aos aviões, importante para responder às questões 1,2 e 6

- TipoAtrasos - informações relativas aos atrasos, importante para responder às questões 8
- InfosGerais - informações gerais de um deterinado voo, importante para responder à questão 5,9 e 10

Para criar a tabela utilizamos o seguinte comando:

```
1 create 'trafego', 'aeroportos', 'infoaviao', 'tipoatrasos', 'infogerais'
```

Para cada uma das famílias de colunas definimos as seguintes colunas como sendo essenciais:

- Aeroportos
  - Dest
  - Cancelled
  - Diverted
- InfoAviao
  - TailNum
  - ArrDelay
  - DepDelay
  - Distance
- TipoAtrasos
  - WeatherDelay
  - NASDelay
  - CarrierDelay
  - SecuritYDelay
  - LateAircraftDelay
- InfosGerais
  - DayOfWeek
  - DepTime
  - ArrTime
  - UniqueCarrier
  - TailNum
  - AirTime
  - Orig
  - Dest
  - Distance

Decidimos não criar uma família de colunas para cada uma das interrogações, visto que isto traria bastante replicação e bastantes problemas em actualizações e tratamento de dados. O que tentamos foi agrupar algumas colunas necessárias para certas interrogações, como por exemplo para os aviões que percorreram maior distância e tiveram mais atrasos mantemos as colunas todas na mesma família de colunas, sendo mais rápido ler esta família com poucas colunas do que com as colunas todas que existiam. No entanto não conseguimos escapar a alguma replicação, neste caso no **Número do avião** e **Destino do voo** visto que estas são importantes em ambas as famílias em que foram colocados, pelo facto de ser normal verificar qual é um determinado avião e o aeroporto de destino de um determinado voo quando vemos as informações gerais do mesmo. Isto tem um custo na actualização de uma determinada linha, no entanto consideramos que o nosso modelo não sofrerá grandes actualizações, podendo isto ser uma limitação.

Com esta divisão esperamos que as perguntas mais frequentes como a 1,7 e 9 sejam respondidas em um tempo menor, enquanto que a pergunta que obtém as informações totais de um determinado voo deve ser respondida com um tempo maior devido às questões referidas inicialmente.

3.3 3ª Versão

A ideia na terceira versão era colocar 2 *regionservers* e tirar partido disto dividindo a tabela em 2 regiões o mais uniformemente possível. Para isto tentamos verificar a opção de divisão uniforme do **HBase** no entanto não conseguimos criar a tabela, pelo que avançamos para outra solução.

A ideia foi verificar qual era a chave que se encontrava aproximadamente no meio e decidimos criar uma divisão explícita nessa chave. Obviamente esta abordagem trazia um problema de que com a chegada de outras chaves esta distribuição deixasse de ser uniforme. Uma forma de realizar esta análise de uma melhor forma era tentar perceber quais os identificadores de voo existentes, visto que as nossas chaves começam por esses, e tentar perceber qual é o que se encontra mais ou menos no meio (obviamente que se os identificadores mudarem muito no futuro continua a existir um problema).

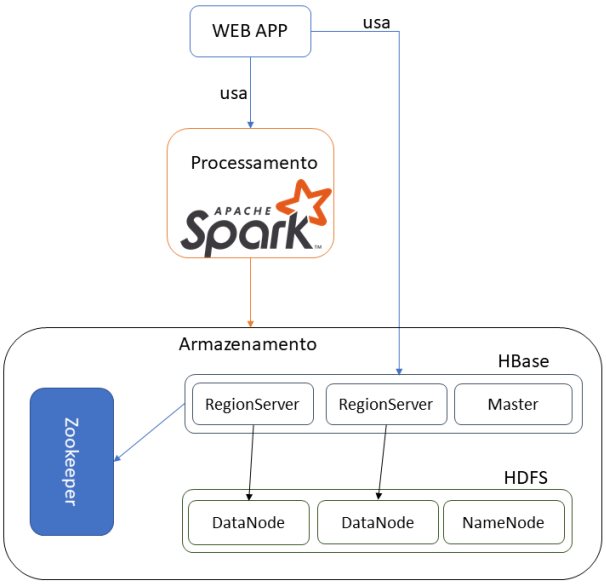

Figura 4: Arquitetura pretendida da solução

O esperado para esta solução era que melhorasse os tempos das interrogações que tratassem muitos dados pelo facto de poder distribuir os *SCAN's* e também pelo facto de tratarem menos linhas. No entanto não conseguimos implementar a solução pelo facto de não conseguirmos importar os dados para o **HBase**, visto que um dos *regionservers* estava constantemente a falhar.

4 Avaliação de Desempenho

Para a avaliação da nossa aplicação e como realizamos uma *WEB API* decidimos utilizar o tempo de resposta como medida de desempenho. Como estamos a medir o tempo de resposta, o importante não são tanto as médias, mas os percentis, que nos indicam a proporção de utilizadores que experenciaram um determinado *delay*. Decidimos então usar os seguintes 3 percentis:

- mediana - que nos indica que 50% dos utilizadores experenciaram um tempo igual ou inferior a este
- percentil 95 - indica o mesmo, mas para 95% dos utilizadores
- percentil 99 - o mesmo que a mediana para 99% dos utilizadores

Apenas utilizamos os primeiros 1500000 registos presentes no *dataset* pelo facto de não termos tido memória suficiente para colocar tudo no **HBase**.

A estratégia de avaliação realizada segue uma abordagem bastante simples, sendo a seguinte:

- Começamos com 1 cliente concorrente a realizar 100 pedidos
- Depois vamos aumentando o número de clientes concorrentes em 1 unidade até 25
- Guardamos o *array* de tempos de resposta num *JSON* da seguinte forma:
  - número de clientes concorrentes: *array* com os tempos de resposta

Apenas comparamos os tempos de resposta para 3 interrogações (1,9,11) pelo facto de estas já conterem basicamente todos os testes que queremos.

- A 1 faz um *SCAN* total à tabela e agrupa os resultados.
- A 9 faz um *SCAN* mas com um prefixo de filtro.
- A 11 faz um *GET* direto na qual tem de ir buscar todas as informações.

4.0.1 Interrogação 1

Relativamente à interrogação 1 e visto que esta demorava bastante tempo não conseguimos realizar os testes com até 25 clientes concorrentes e não conseguimos realizar 100 testes por cliente. Neste caso apenas conseguimos realizar 10 testes por clientes, com até 10 clientes concorrentes.

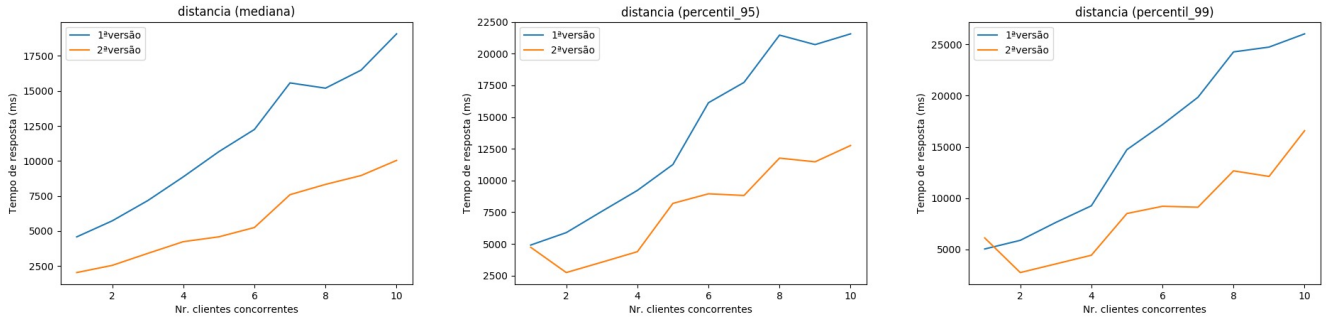

Figura 5: Resultados da avaliação da interrogação 1

Como podemos verificar nas figuras existe uma grande diferença nos 3 casos testados, diferença esta que aumenta à medida que o número de clientes concorrentes aumenta. Isto era algo que já esperávamos visto que esta questão apenas trabalha com algumas colunas presentes em uma das famílias de colunas criadas, sendo menos dispendioso no caso em que as colunas estão divididas por família.
De referir apenas que o objectivo era terem sidos realizados mais testes para ser mais fiável.

4.0.2 Interrogação 9

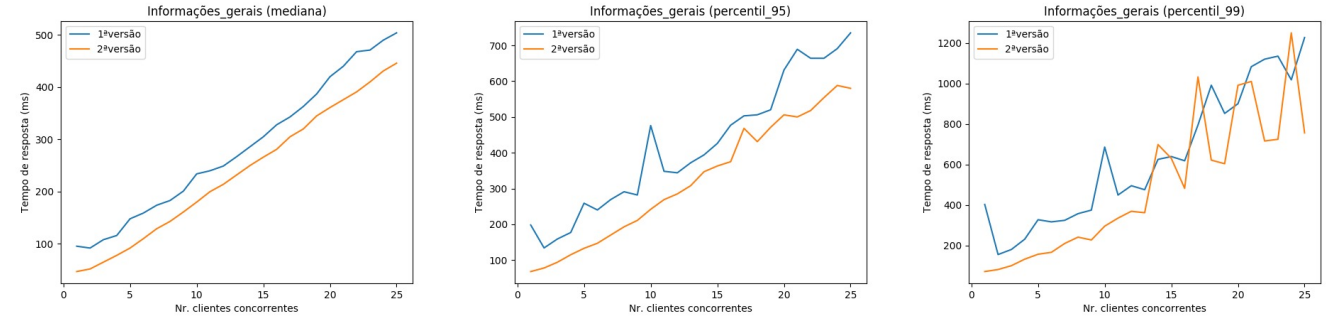

Figura 6: Resultados da avaliação da interrogação 9

Aqui verificamos também que existe uma melhoria relativamente à mediana e ao percentil 95, melhoria esta que se mantém aproximadamente constante ao longo do tempo. A melhoria é, no entanto, menor do que no caso anterior possivelmente porque aqui já estamos a considerar um número de colunas maior, apesar de não serem todas e estarem todas na mesma família. Relativamente ao percentil 99 verificamos também que é melhor na segunda versão, no entanto a partir de um certo ponto torna-se bastante inconstante nesta versão.

4.0.3 Interrogação 11

A interrogação 11 é melhor em todos os casos na primeira versão. A melhoria é reduzida, mas isto prende-se no facto de apenas estarmos a obter um registo. Esta melhoria era algo que já antecipávamos pelo facto de na primeira versão as informações estarem todas no mesmo ficheiro, enquanto que na segunda estão em ficheiros distintos e existe uma penalização para as ir buscar todas.

4.0.4 Conclusões da análise

Como foi referido anteriormente consideramos que as interrogações 1 e 9 são mais usuais do que a 11 pelo que podemos verificar que a solução proposta na segunda versão é relativamente mais eficiente do que a primeira. Isto comprova que a versão final é melhor do que a inicial.

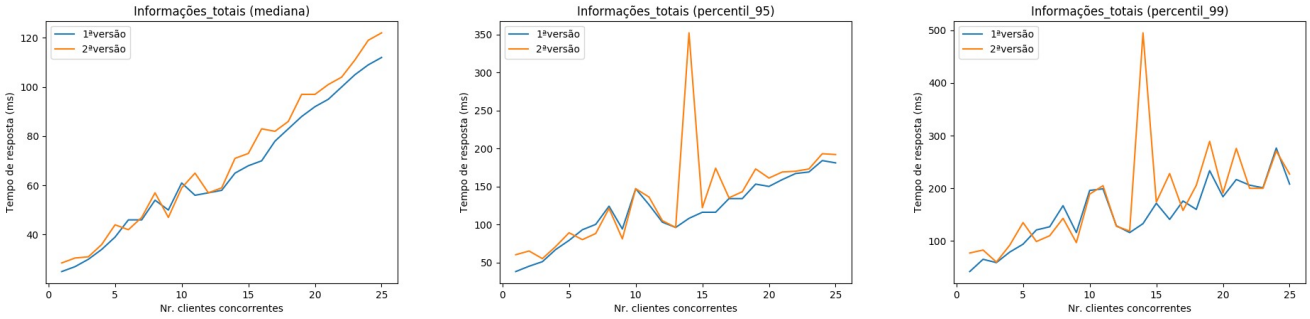

Figura 7: Resultados da avaliação da interrogação 1

Como podemos verificar nas figuras existe uma grande diferença nos 3 casos testados, diferença esta que aumenta à medida que o número de clientes concorrentes aumenta. Isto era algo que já esperávamos visto que esta questão apenas trabalha com algumas colunas presentes em uma das famílias de colunas criadas, sendo menos dispendioso no caso em que as colunas estão divididas por família.

De referir apenas que o objectivo era terem sidos realizados mais testes para ser mais fiável.

5 Conclusão

Como foi referido anteriormente a arquitectura actual da aplicação adequa-se mais para a nossa aplicação e para os seus objetivos comparativamente à arquitectura inicial. No entanto, como mencionamos, a ideia ainda era testar uma outra arquitectura com a finalidade de ainda se adequar mais, sendo mais eficiente. Apesar desse contratempo consideramos que a solução apresentado está consideravelmente melhor do que a primeira solução e que contém alguma complexidade. Não é apenas uma solução na qual ligamos a uma base de dados (**HBase**), mas onde tiramos proveito de ferramentas de processamento dos mesmos já existentes, e bastante utilizadas na indústria, para computar as interrogações mais complexas.

Existe, contudo, uma variedade de melhorias que podem ser feitas à nossa solução, passando a explicar alguamas dessas melhorias de seguida. A primeira melhoria, que foi alvo de tentativa por parte do grupo, seria conseguir criar um *cluster* do **Spark**[2] com um *Master* e vários *Workers*, em vez de usar apenas o **Spark** de uma forma local, na qual não existe grande controlo sobre o número de *Workers* ativos. Isto potencializaria, possivelmente, a execução de tarefas paralelas e distribuídas, permitindo um tempo de resposta mais baixo para clientes concorrentes.

Uma outra optimização que poderia melhorar o desempenho da aplicação criada é o facto de que podíamos introduzir uma arquitectura na qual existe uma *cache* onde os resultados das interrogações vão sendo mantidos na mesma (possivelmente em memória). Isto poderia melhorar o desempenho no caso de existirem muitas interrogações a estarem guardadas em *cache*, sendo mais rápida a sua execução. Porém é necessário ter noção de que sempre que uma interrogação não estivesse em *cache* sofria uma penalização do tempo perdido à procura na *cache* pelo que esta procura teria de ser eficiente. Para além disso as escritas eventualmente invalidariam algumas interrogações em cache pelo que era necessário ter cuidado com as mesmas e ter noção que ia aumentar o seu tempo.

Algo que poderá ser melhorado no futuro e que talvez seja a parte mais pobre da nossa aplicação é a falta de existência de um *workflow* para o processo de inserção de dados. Como foi descrito anteriormente, é necessário que se corra um programa que trata os dados e depois esses dados têm de ser manualmente inseridos no **HBase**. Obviamente que o objetivo não é este e que no futuro a ideia passa por realizar um fluxo no qual tenhamos o programa que trata os dados e os transfere para um outro programa, possivelmente através de um evento, para que esse programa automaticamente transfira os dados para a camada de armazenamento, sendo assim apenas necessário passar os dados ao "programa"que realiza o tratamento.

Uma evolução futura poderá também incluir a criação de uma interface gráfica para que os dados possam ser apresentados de um modo mais apelativo tanto ao nível da visualização como da compreensão.

Referências

[1] https://community.hortonworks.com/articles/4942/import-csv-data-into-hbase-using-importtsv.html

[2] https://medium.com/@marcovillarreal_40011/creating-a-spark-standalone-cluster-with-docker-and-docker-compose-ba9d743a157f

[3] https://stackoverflow.com/questions/25040709/how-to-read-from-hbase-using-spark

[4] https://spark.apache.org/docs/latest/api/java/index.html

[5] https://hbase.apache.org/apidocs/index.html