Laboratórios de Informática III

Trabalho Prático nº 2

Relatório de desenvolvimento

José Resende	Miguel Lobo	Patrícia Barreira
(a77486)	(a78225)	(a79007)

11 de Junho de 2017

Resumo

A `Wikipédia` é uma enciclopédia web global de livre acesso. Um dos aspetos mais interessantes desta plataforma é que os artigos são escritos pelas próprias pessoas que utilizam o serviço para procurar de informação. Este serviço é hoje em dia usado massivamente por milhões de utilizadores todos os dias.Existem mais de 43 milhões de artigos (970 272 em português, até 06 de junho de 2017) encontrados hoje na `Wikipédia`.

Conteúdo

Capítulo 1

Introdução

O enunciado deste trabalho prático previa uma análise da quantidade gigantesca de dados uma vez que estes contêm informações muito útil. Não só ao nível do texto dos artigos mas também dos próprios metadados dos mesmos. Perceber as características da `Wikipédia`,tais como os colaboradores mais ativos ou os artigos que mais são revisto é fundamental para melhor compreender o funcionamento desta plataforma da `Wikipédia`.Assim o objetivo deste trabalho prático é desenhar cuidadosamente um sistema de forma a não só conseguir carregar esta quantidade enorme de dados mas também utilizar as melhores estruturas de dados e algoritmos para depois conseguir responder a interrogações como, número total de artigos ou o número de revisões por exemplo, em tempo útil.

Capítulo 2

Concepção do problema

Pela análise do problema, foi possível dividir as *queries* em 3 grupos distintos:

- Artigos
- Colaboradores
- Texto

Para as *queries* de artigos são necessários os respetivos id's, logo torna-se imperativo ter uma `collection` onde a procura de um artigo seja o mais rápido possível. De notar que cada artigo podia ter mais do que uma revisão, assim, é necessário implementar uma `collection` associada à `collection` dos artigos capaz de armazenar as suas revisões únicas.

No caso das *queries* precisa-se de uma `collection` que guarde os nomes dos contribuidores, ids e o número de contribuições dos mesmos.

Já para os textos, as *queries* focam-se no número de palavras e tamanho do texto. Por isso, será necessário guardar numa `collection`, o número de palavras, número de carateres e por fim, id e título de artigos.

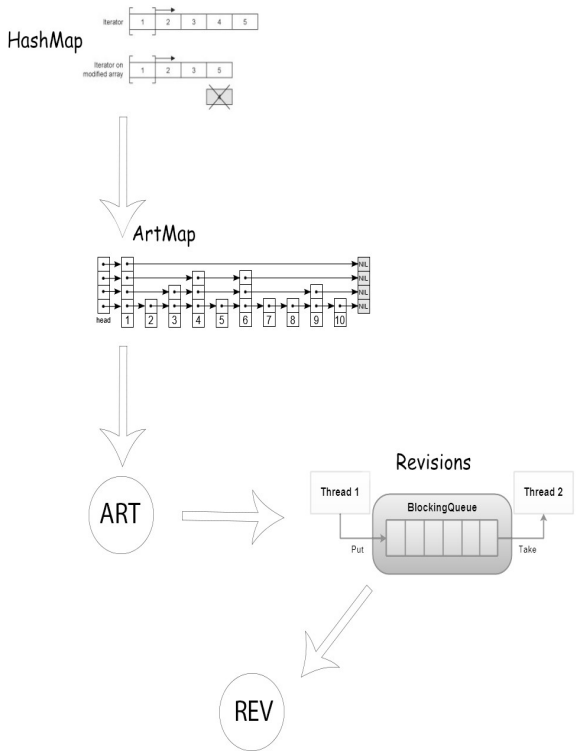
Capítulo 3

Concepção da solução

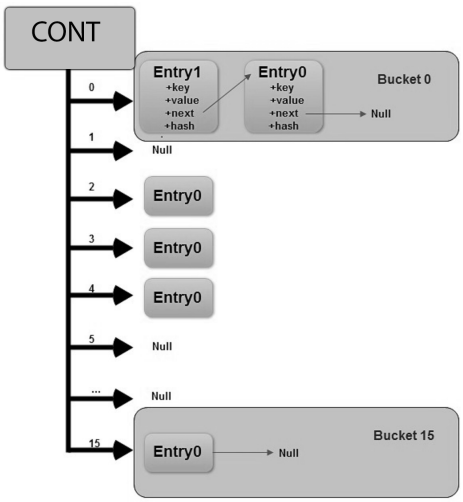
Partindo do principio que se queria distribuir os artigos por diferentes *Maps*, decidiu-se implementar um *ArrayList*, em que cada posição do *ArrayList* é constituída por um *Map*, sendo que cada posição deste, contém uma *Queue*.
Mais concretamente, foram utilizadas as seguintes `collections` para a implementação do descrito em a cima:

- *CopyOnWriteArrayList* para o *ArrayList*
- *ConcurrentSkipListMap* para o *Map*
- *LinkedBlockingQueue* para a *queue*

A escolha da *collection CopyOnWriteArrayList* deve-se ao facto de ser uma variante do*ArrayList* com `thread-safe`. Operações como `add` estão implementadas fazendo uma cópia do array subjacente. Tal como descrito na `API` um objeto desta classe é, por norma, muito dispendioso a nível de memória, no entanto poderá ser mais eficiente em casos que requeiram mais operações transversais do que mutações. Portanto, com está *collection* podemos então utilizar *parallelStream* de forma a aumentar a performance.
Já em relação à *ConcurrentSkipListMap*, está foi escolhida uma vez que para além dos requisitos fundamentais de um *TreeMap* permite também que as operações sejam concorrentes.É a versão concorrente de *TreeMap* que o `Java` fornece.É na verdade uma *SkipList* o que permite uma pesquisa rápida dentro de uma sequência ordenada de elementos. Cada chave será o id do Artigo.E tal como em cima, permitir a utilização de *parallelStream* para uma melhor performance.
Por ultimo, a *LinkedBlockingQueue* surgiu na medida que é a `queue` mais simples que permite concorrência.
Em suma, para o caso do grupo de *queries* referentes aos artigos e texto utilizou-se a estrutura descrita a cima. Já para as *queries* dos contribuidores foi criada um *HashMap* em que a chave é o id do contribuidor.Esta estrutura, tal como as outras,garante o sincronismo das *threads* para as travessias por isso também foi utilizado *parallelStream* para este caso.
Assim o conjunto das *collection* descritas a cima encaixam da seguinte forma:



e para os os Contribuidores:



- HashMap

```
private CopyOnWriteArrayList<ArtTree> hash;
private Integer N;
```

- ArtMap

```
private ConcurrentSkipListMap<Long,Article> tree;
```

- Articles

```
private Long id;
private String title;
private LinkedBlockingQueue<Revision> revisions;
```

- Revisions

```
private Long id;
private String timestamp;
private Long words;
private Long tamanho;
```

- ContMap

```
private HashMap <Long,Contributors> ContMap;
```

- Contributors

```
private Long id;
private String nome;
private Integer contN;
```

Capítulo 4

Testes de Performance

- O tamanho do *CopyOnWriteArrayList* é importante uma vez que para adicionar uma elemento é calculado o seu *hashcode*.

```
  public void addArticle(Article a){
    int pos = a.hashCode() % N;

    hash.get(pos).addArticle(a);
}
```

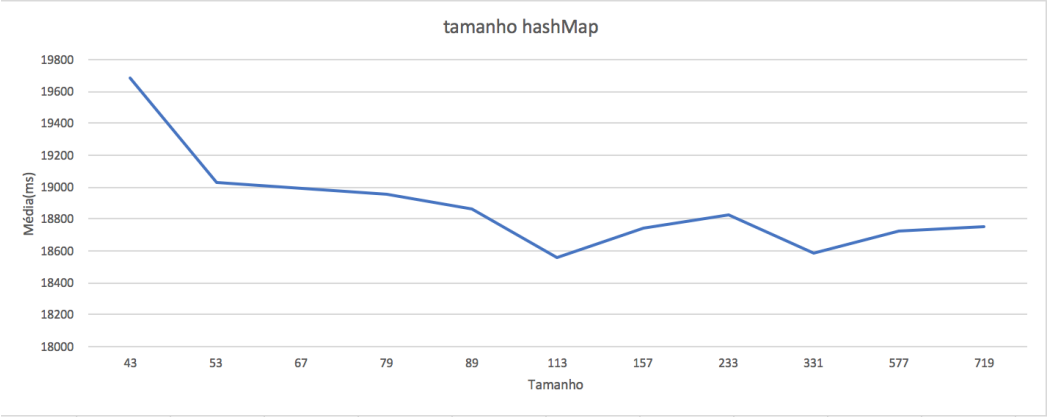
onde:

```
public int hashCode(){
    return id.hashCode();
}
```

e

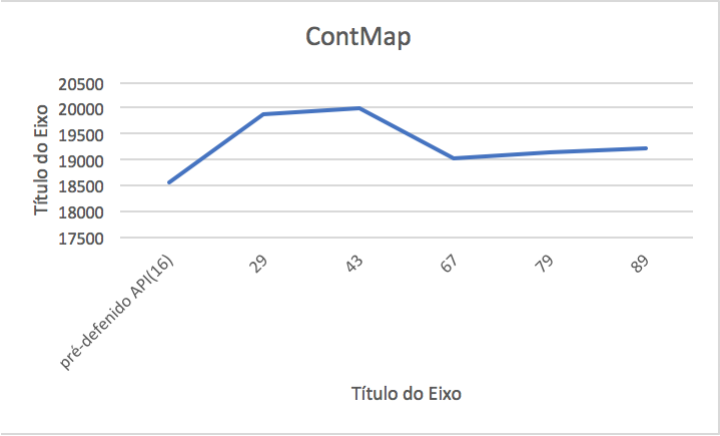
```
    public void addArticle(Article a){
    Long id = a.getId();
    LinkedBlockingQueue<Revision> revTo, revFrom;

    if(map.containsKey(id)){
        Article a2 = map.get(id);

        revTo = a2.getRevisions();
        revFrom = a.getRevisions();

        revFrom
        .parallelStream()
        .filter(r -> !revTo.contains(r))
        .forEach(r -> revTo.offer(r.clone()));

    }
    else
        map.put(id, a.clone());
}
```

Portanto foi feito um estudo sobre o tamanho que haveria de ser escolhido:



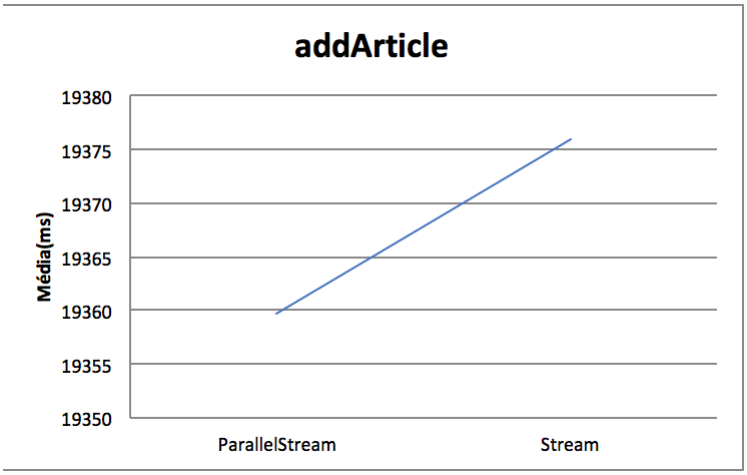
Posto isto, a capacidade inicial escolhida foi 116.

- Para o *HashMap* dos contribuidores também foi feito o estudo do tamanho inicial passado ao construtor:



Portanto utilizou-se o construtor vazio da *API* que irá criar um *HashMap* com capacidade inicial 16.

- Para o método `addArticle` foi feito o estudo se *Parallelstream* seria mais eficiente do que *stream*



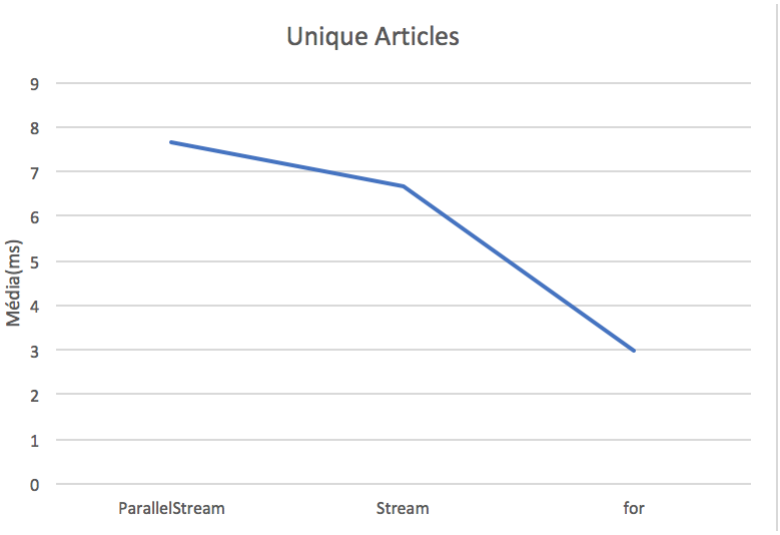
Então utilizou-se *Parallelstream*.

- Para a `query unique_articles` verificou-se se seria mais eficiente com iteradores externos do que internos: Iterador Interno:

```
 hashmap
.getHash()
.parallelStream()
.flatMap(artree -> artree.getMap().values().parallelStream())
.count();
```

Iterador Externo:

```
long soma = 0;
for(ArtMap aux : hashmap.getHash()){
    soma += aux.getMap().size();
}
```

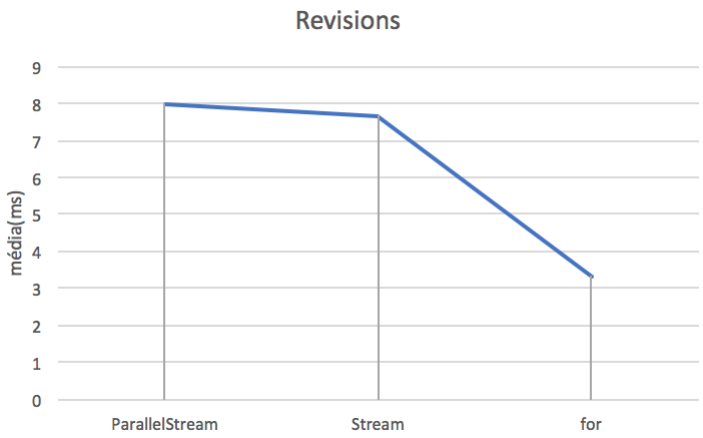


- Para a `query all_revisions` verificou-se se seria mais eficiente com iteradores externos do que internos: Iterador Interno:

```
 hashmap
.getHash()
.parallelStream()
.flatMap(artree -> artree.getMap().values().parallelStream())
.mapToLong(Article::numeroDeRevisoes)
.sum();
```

Iterador Externo:

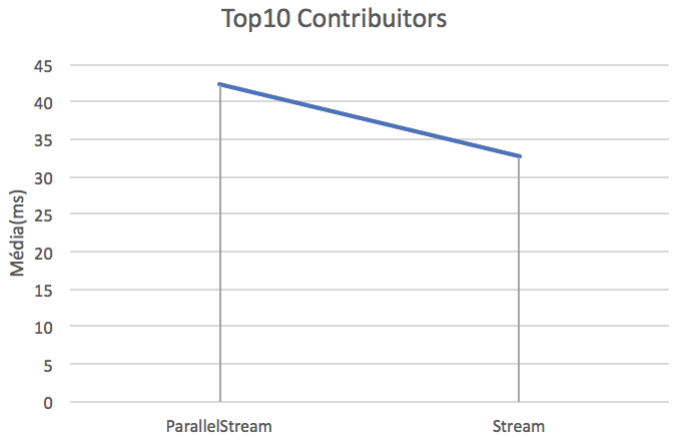
```
long soma = 0;
for(ArtMap aux : hashmap.getHash()){
    for(Article ar : aux.getMap().values()){
        soma += ar.numeroDeRevisoes();
    }
}
```



- Para a `query top10 Contributors` fez-se uma análise da utilização de *parallelStream* e *Stream*

```
 contMap
.getContMap()
.values()
.stream()
.sorted(Comparator.comparing(Contributors::getContN).reversed())
.limit(10)
.mapToLong(Contributors::getId)
.collect(ArrayList<Long>::new, ArrayList<Long>::add, ArrayList<Long>::addAll);
```

Não há qualquer problema de encapsulamento ao utilizador o método `addAll` porque o objeto `Long` é imutável.



Então utilizou-se apenas *Stream*.

- Para a `query top20 largest Articles` testou-se a diferença entre a utilização entre *parallelStream* e *Stream*

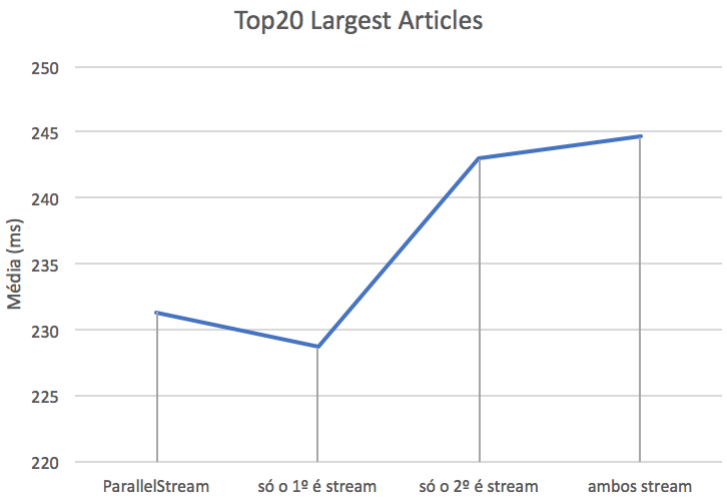
```
 hashmap
.getHash()
.stream()
.flatMap(artree -> artree.getMap().values().parallelStream())
.sorted(Comparator.comparing(Article::LargestRevLength).reversed().thenComparing(Article
.limit(20)
```

```
                .mapToLong(Article::getId)
                .collect(ArrayList<Long>::new, ArrayList<Long>::add, ArrayList<Long>::addAll);
```

onde:

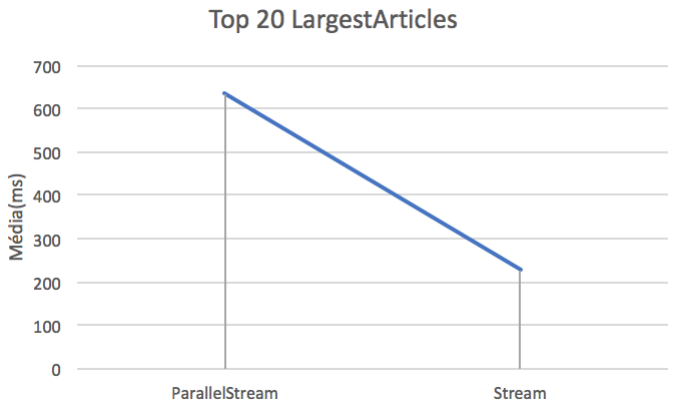
```
        public Long LargestRevLength(){
        return  revisions
                .stream()
                .mapToLong(rev -> rev.getTamanho())
                .max()
                .getAsLong();

    }
```

Não há qualquer problema de encapsulamento ao utilizador o método `addAll` porque o objeto `Long` é imutável.
Teste para o primeiro método apresentado em cima:



Portanto utilizou-se *Stream* no primeiro e *ParallelStream* no segundo.

Teste para o segundo método em cima indicado:



Utilizou-se sem dúvida *Stream*.

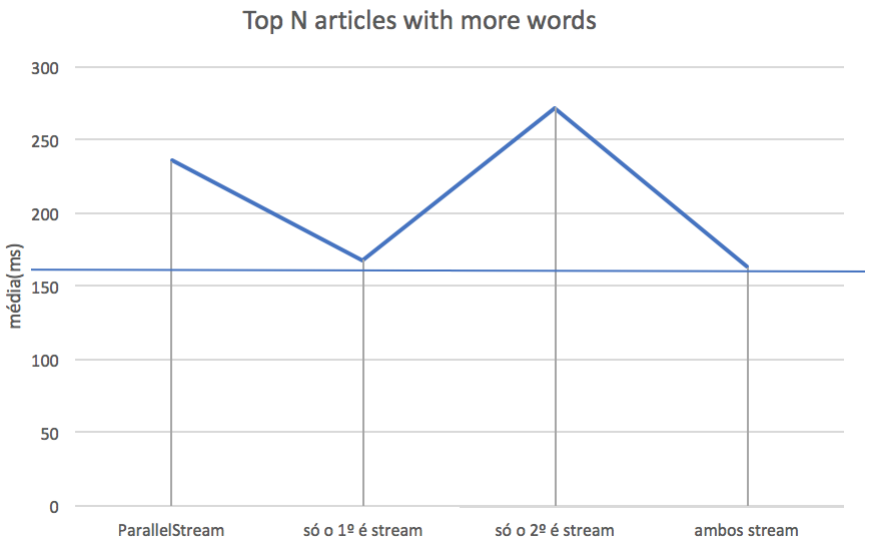
- Para a `query topN articles_with_more_words` testou-se a diferença entre a utilização entre *parallelStream* e *Stream*

```
                .getHash()
                .stream()
                .flatMap(artree -> artree.getMap().values().stream())
                .sorted(Comparator.comparing(Article::largestRevWords).reversed().thenComparing(Article:
                .limit(n)
                .mapToLong(Article::getId)
                .collect(ArrayList<Long>::new, ArrayList<Long>::add, ArrayList<Long>::addAll);
```

Não há qualquer problema de encapsulamento ao utilizador o método `addAll` porque o objeto `Long` é imutável. onde:

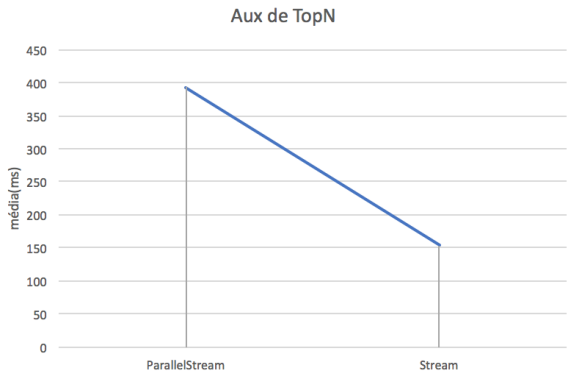
```
        public long largestRevWords() {
        return revisions
                .stream()
                .mapToLong(rev -> rev.getWords())
                .max()
                .getAsLong();
    }
```

Teste para o primeiro método apresentado em cima:



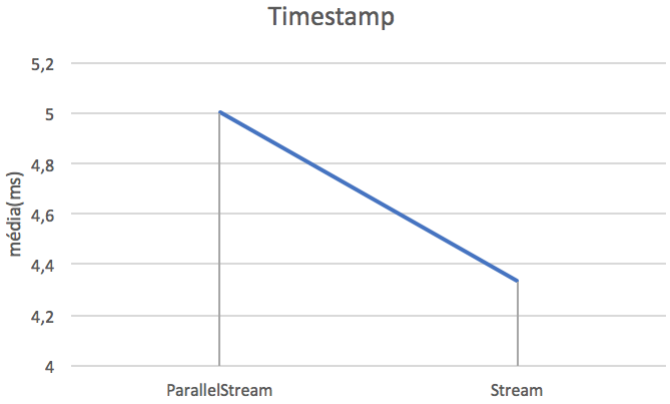
Portanto utilizou-se em ambos *Stream*.

Teste para o segundo método em cima indicado:



Utilizou-se sem dúvida *Stream*.

- Para a `query timestamp` os resultados foram os seguintes:



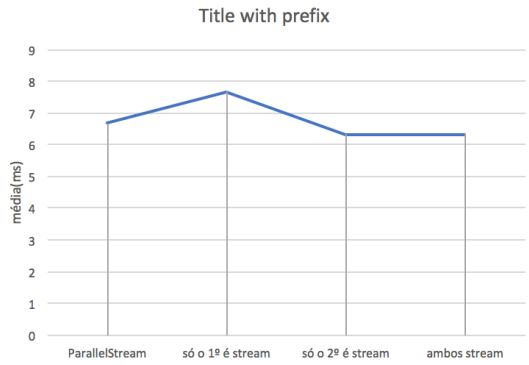
Portanto utilizou-se *stream*.

```
Article a = hashmap.getArticle(article_id);
if (a != null){
        Revision r = a.getRevision(revision_id);
        if(r!=null)
            return a.getRevisions().parallelStream()
                    .filter(rev -> rev.getId() == revision_id)
                    .findFirst()
                    .get()
                    .getTimestamp();
        else
            return null;

}
else
    return null;
```

Neste método teve-se a atenção que o *filter* termina quando encontrar o primeiro elemento que verifique a condição (*findFirst*).

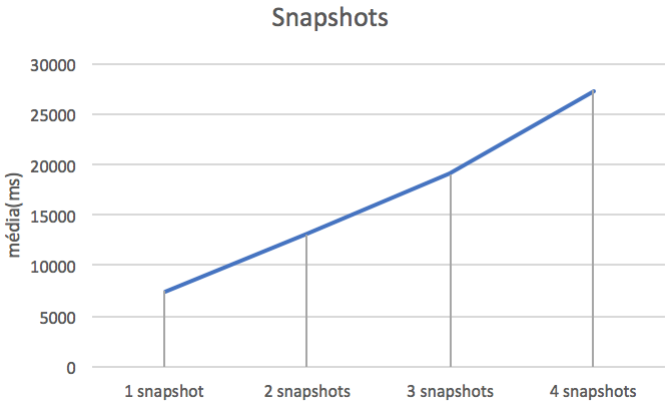
- Para a `query title_with_prefix` os resultados foram os seguintes:

```
hashmap
.getHash()
.stream()
.flatMap(artree -> artree.getMap().values().stream())
.filter(art -> art.getTitle().startsWith(prefix))
.map(Article::getTitle)
.sorted()
.collect(ArrayList<String>::new, ArrayList<String>::add, ArrayList<String>::addAll);
```



Portanto utilizou-se *streams* em ambos.

Em suma,foi feito o teste de como o programa varia com o aumento da carga de trabalho. As conclusões foram as seguintes:



Mostra que há um aumento linear conforme o tamanho do input.

Capítulo 5

Conclusão

As *queries* implementadas respondem aos requisitos impostos pelo enunciado, pelo que este grupo de trabalho se considera sobremaneira satisfeito. É no entanto evidente que muitas outras otimizações poderiam ser acrescentadas, tanto no método que conta as palavras como na *query load*.