

Benchmarking Java Streams

Processamento de Dados com Streams de JAVA

Afonso Silva

Octávio Maia

10 de Janeiro de 2018

Conteúdo

I. Testes	2
1. Cálculo dos valores de transações registadas	3
2. Extração dos primeiros e últimos 20% de transações realizadas	6
3. Esforço de eliminação de duplicados	9
4. Comparação entre a aplicação de método estático, BiFunction e Lambda	11
5. Comparação da ordenação através de um TreeSet e do método sorted	13
6. Comparação do cálculo da soma com ou sem partições	15
7. Determinação da maior transação entre uma determinada hora	18
8. Cálculo do total facturado numa determinada semana do ano	20
9. Cálculo do total de IVA associado a cada mês	22
10. Comparação de performance entre JDK8 e JDK9	24

Parte I.

Testes

1. Cálculo dos valores de transações registradas

Observações

Métodos a testar

```
public double sumArray() {  
    double[] values = new double[this.transactions.size()];  
    int i = 0;  
  
    for (TransCaixa transaction : this.transactions) {  
        values[i++] = transaction.getValor();  
    }  
  
    double sum = 0.0;  
  
    for (i = 0; i < values.length; i++) {  
        sum += values[i];  
    }  
  
    return sum;  
}
```

Listing 1.1: Cálculo da soma dos valores das transações através de um array do tipo double

```
public double sumDoubleStream() {  
    DoubleStream values = this.transactions.stream()  
        .mapToDouble(TransCaixa::getValor);  
    return values.sum();  
}  
  
public double sumDoubleStreamP() {  
    DoubleStream values = this.transactions.parallelStream()  
        .mapToDouble(TransCaixa::getValor);  
    return values.sum();  
}
```

Listing 1.2: Cálculo da soma dos valores das transações através de uma DoubleStream

```

public double sumStream() {
    Stream<Double> values = this.transactions.stream()
        .map(TransCaixa::getValor);
    return values.reduce(0.0, (a,b) -> a + b);
}

public double sumStreamP() {
    Stream<Double> values = this.transactions.parallelStream()
        .map(TransCaixa::getValor);
    return values.reduce(0.0, (a,b) -> a + b);
}

```

Listing 1.3: Cálculo da soma dos valores das transações através de Stream<Double>

Resultados

Input	(1) sumArray	(2) sumDoubleStream	(3) sumDoubleStreamP	(4) sumStream	(5) sumStreamP
1000000 transactions	0,010215	0,010794	0,009364	0,026481	0,015645
2000000 transactions	0,021934	0,021800	0,009912	0,053940	0,026226
4000000 transactions	0,044380	0,047799	0,026682	0,113063	0,051981
8000000 transactions	0,078939	0,089719	0,046436	0,214286	0,126348

Análise e conclusões

Após uma breve observação dos gráficos, podemos afirmar que a estrutura de dados mais adequada é de facto a *DoubleStreamP* que implementa streams paralelas. Em contraste, a pior estrutura em nível de performance é a *Stream*, sendo até 5 vezes mais lenta que a *DoubleStreamP*.

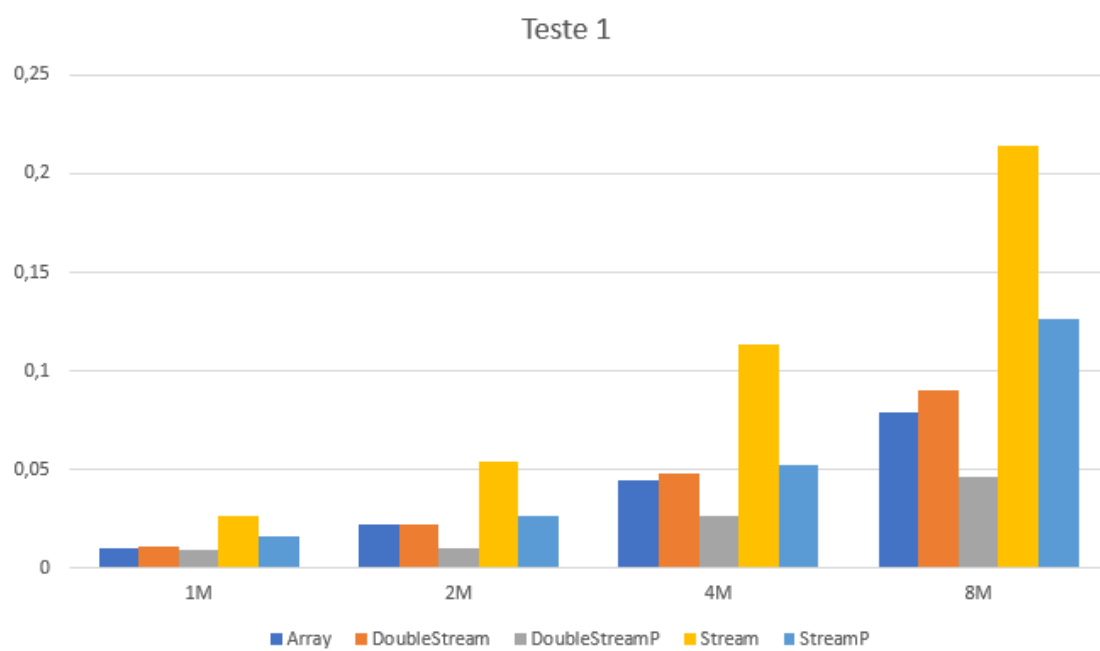


Figura 1.1.: Representação gráfica destes resultados

2. Extração dos primeiros e últimos 20% de transações realizadas

Observações

Métodos a testar

```
public SimpleEntry<List<TransCaixa>, List<TransCaixa>> byDateList() {
    int nelems = 20 * this.transactions.size() / 100;

    List<TransCaixa> sorted = new ArrayList<>(this.transactions);
    sorted.sort(Comparator.comparing(TransCaixa::getData));

    List<TransCaixa> first = sorted.subList(0, nelems);
    List<TransCaixa> last = sorted.subList(
        sorted.size() - 1 - nelems, sorted.size() - 1);

    return new SimpleEntry<>(first, last);
}
```

```
public SimpleEntry<List<TransCaixa>, List<TransCaixa>> byDateSet() {
    int nelems = 20 * this.transactions.size() / 100;

    TreeSet<TransCaixa> sorted = new TreeSet<>(
        // Com este comparador garante-se que a lista fica ordenada
        // e nao se removem os elementos iguais
        (t1, t2) -> t1.getData().isBefore(t2.getData()) ? -1 : 1
    );
    sorted.addAll(this.transactions);

    List<TransCaixa> first = new ArrayList<>(sorted)
        .subList(0, nelems);
    List<TransCaixa> last = new ArrayList<>(sorted.descendingSet())
        .subList(0, nelems);

    return new SimpleEntry<>(first, last);
}
```

```
public SimpleEntry<List<TransCaixa>, List<TransCaixa>> byDateStream() {
    int nelems = 20 * this.transactions.size() / 100;

    List<TransCaixa> first = this.transactions.stream()
```

```

        .sorted(Comparator.comparing(TransCaixa::getData))
        .limit(nelems)
        .collect(Collectors.toList());
List<TransCaixa> last = this.transactions.stream()
        .sorted((t1, t2) -> t2.getData().compareTo(t1.getData()))
        .limit(nelems)
        .collect(Collectors.toList());

return new SimpleEntry<>(first, last);
}

```

```

public SimpleEntry<List<TransCaixa>, List<TransCaixa>> byDateStreamP() {
    int nelems = 20 * this.transactions.size() / 100;

    List<TransCaixa> first = this.transactions.stream()
        .sorted(Comparator.comparing(TransCaixa::getData))
        .limit(nelems)
        .collect(Collectors.toList());
    List<TransCaixa> last = this.transactions.stream()
        .sorted((t1, t2) -> t2.getData().compareTo(t1.getData()))
        .limit(nelems)
        .collect(Collectors.toList());

    return new SimpleEntry<>(first, last);
}

```

Resultados

Input	(1) byDateList	(2) byDateSet	(3) byDateStream	(4) byDateStreamP
1000000 transactions	1,029137	1,939311	1,861916	2,239986
2000000 transactions	2,331822	3,706627	3,858658	4,378531
4000000 transactions	5,745454	9,411794	9,281572	10,556521
8000000 transactions	12,635500	21,085741	20,579760	22,002980

Análise e conclusões

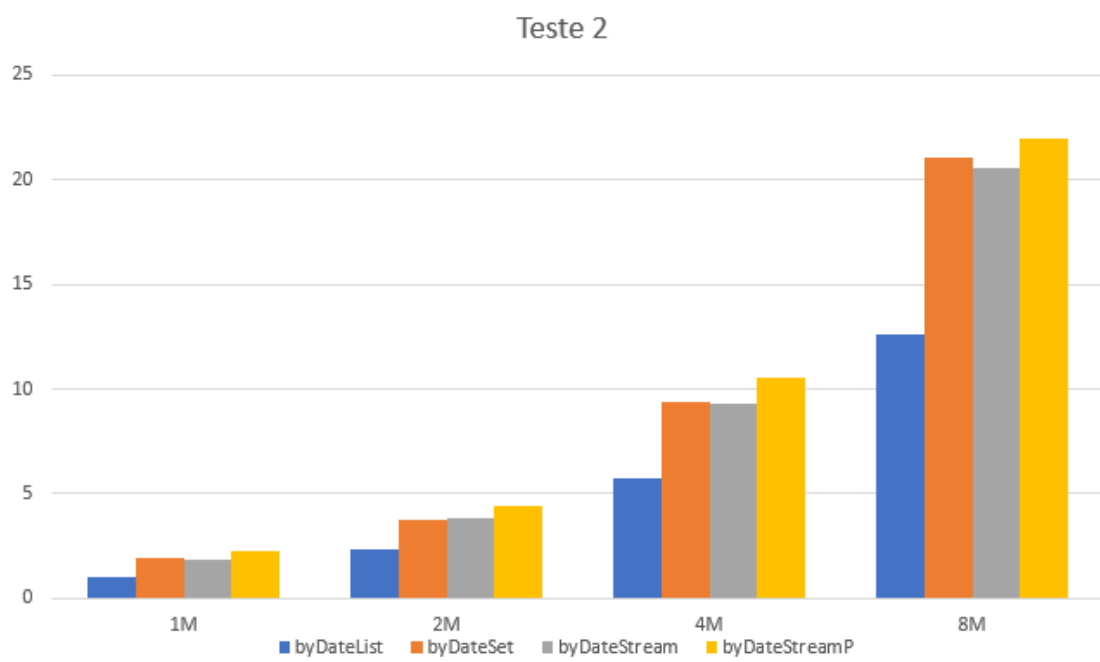


Figura 2.1.: Representação gráfica destes resultados

3. Esforço de eliminação de duplicados

Observações

Métodos a testar

```
public Integer [] uniqueArray () {  
    Set<Integer> nodups = new TreeSet<>();  
    for (int value : this.values) {  
        nodups.add(value);  
    }  
  
    return nodups.toArray(new Integer[nodups.size()]);  
}
```

Listing 3.1: Eliminação dos duplicados através de um array de inteiros

```
public Integer [] uniqueList () {  
    List<Integer> aux = new ArrayList<>();  
    for (int value : this.values) {  
        aux.add(value);  
    }  
  
    List<Integer> nodups = new ArrayList<>(new HashSet<>(aux));  
    return nodups.toArray(new Integer[nodups.size()]);  
}
```

Listing 3.2: Eliminação dos duplicados através de uma lista de inteiros

```
public int [] uniqueIntStream () {  
    IntStream values = new Random().ints(this.values.length, 0, 9999);  
    return values.distinct().toArray();  
}
```

Listing 3.3: Eliminação dos duplicados através de uma stream de inteiros

Resultados

Input	(1) uniqueArray	(2) uniqueList	(3) uniqueIntStream
1000000 random numbers	0,130102	0,027098	0,025731
2000000 random numbers	0,253295	0,056908	0,048782
4000000 random numbers	0,483225	0,094416	0,087328
8000000 random numbers	1,033746	0,223069	0,203687

Análise e conclusões

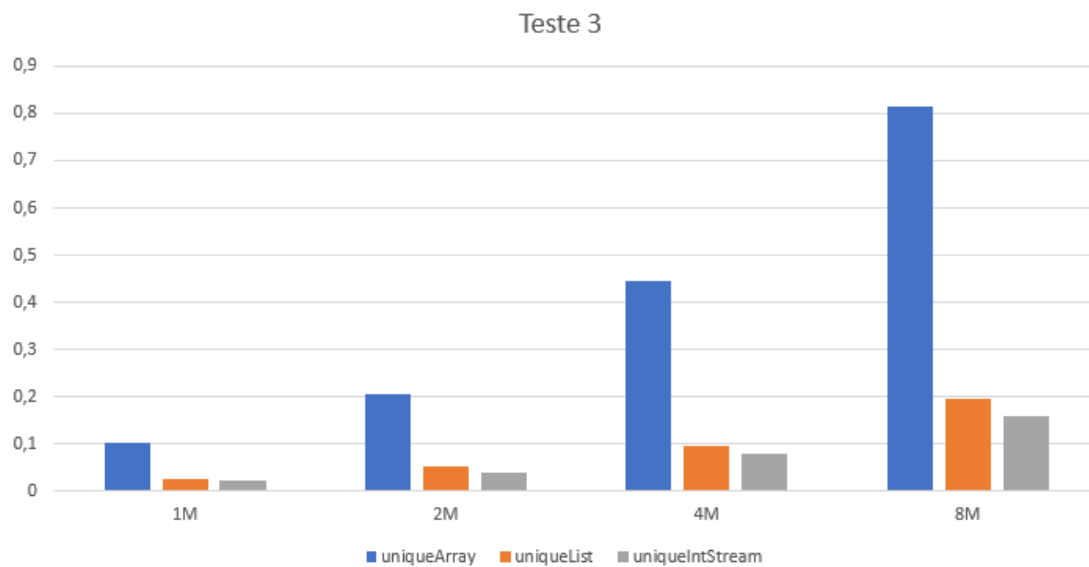


Figura 3.1.: Representação gráfica destes resultados

4. Comparação entre a aplicação de método estático, BiFunction e Lambda

Observações

Métodos a testar

```
public static int div(int x, int y) {  
    return x / y;  
}  
  
public int [] divMethodStream() {  
    return Arrays.stream(this.values)  
        .map(x -> div(x, 2)).toArray();  
}  
  
public int [] divMethodStreamP() {  
    return Arrays.stream(this.values).parallel()  
        .map(x -> div(x, 2)).toArray();  
}
```

Listing 4.1: Divisão de todos os números por 2 através de um método estático

```
public int [] divBiFunStream() {  
    BiFunction<Integer, Integer, Integer> f = (x, y) -> x / y;  
    return Arrays.stream(this.values)  
        .map(x -> f.apply(x, 2)).toArray();  
}  
  
public int [] divBiFunStreamP() {  
    BiFunction<Integer, Integer, Integer> f = (x, y) -> x / y;  
    return Arrays.stream(this.values).parallel()  
        .map(x -> f.apply(x, 2)).toArray();  
}
```

Listing 4.2: Divisão de todos os números por 2 através de uma BiFunction

```
public int [] divLambdaStream() {  
    return Arrays.stream(this.values).map(x -> x / 2).toArray();  
}  
  
public int [] divLambdaStreamP() {
```

```

    return Arrays.stream(this.values).parallel().map(x -> x / 2).toArray();
}

```

Listing 4.3: Divisão de todos os números por 2 através de um Lambda

Resultados

1000000 transactions	1,005284	1,529525	1,639273	1,830654
2000000 transactions	2,284556	3,531155	3,664016	4,204185
4000000 transactions	5,314449	8,259452	8,455954	9,671057
8000000 transactions	11,686906	20,374520	18,379551	21,186783

Análise e conclusões

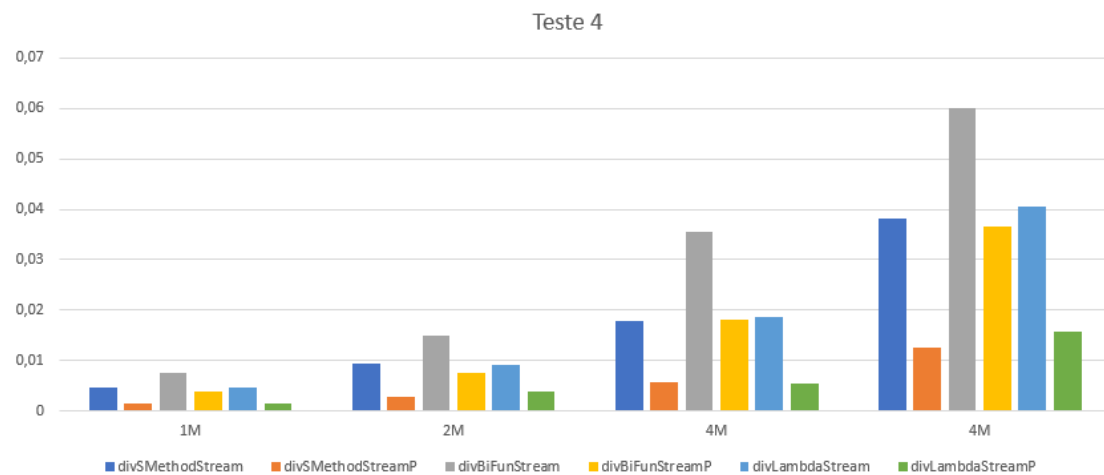


Figura 4.1.: Representação gráfica destes resultados

5. Comparação da ordenação através de um TreeSet e do método sorted

Observações

Métodos a testar

```
public TreeSet<TransCaixa> sortTreeSet() {
    Comparator<TransCaixa> byDate =
        // Com este comparador garante-se que a lista fica ordenada
        // e nao se removem os elementos iguais
        (t1, t2) -> t1.getData().isBefore(t2.getData()) ? -1 : 1;

    return this.transactions.stream()
        .collect(Collectors.toCollection(
            () -> new TreeSet<>(byDate)));
}
```

Listing 5.1: Ordenação através de um TreeSet

```
public List<TransCaixa> sortList() {
    Comparator<TransCaixa> byDate =
        Comparator.comparing(TransCaixa::getData);

    return this.transactions.stream()
        .sorted(byDate).collect(Collectors.toList());
}
```

Listing 5.2: Ordenação através do método sorted

Resultados

Input	(1) sortTreeSet	(2) sortList
1000000 transactions	1,510102	0,849679
2000000 transactions	3,449318	1,840600
4000000 transactions	8,242589	4,047586
8000000 transactions	19,584347	8,851000

Análise e conclusões

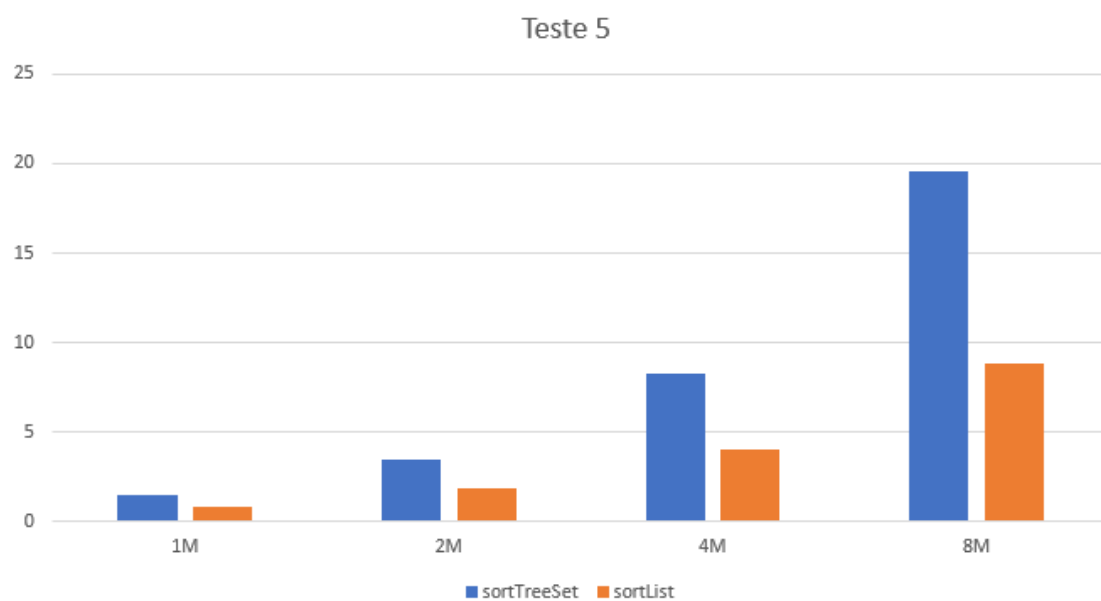


Figura 5.1.: Representação gráfica destes resultados

6. Comparação do cálculo da soma com ou sem partições

Observações

Métodos a testar

```
public double sum() {
    double res = 0.0;
    for (TransCaixa transCaixa : this.transactions) {
        res += transCaixa.getValor();
    }
    return 0.0;

    return res;
}

public double sumPartition() {
    // Make the partitions
    List<List<TransCaixa>> partitions = new ArrayList<>();
    int[] pos = {
        0,
        this.transactions.size() - 3 * transactions.size() / 4,
        this.transactions.size() - 2 * transactions.size() / 4,
        this.transactions.size() - 1 * transactions.size() / 4,
        this.transactions.size() - 0 * transactions.size() / 4,
    };

    partitions.add(this.transactions.subList(pos[0], pos[1]));
    partitions.add(this.transactions.subList(pos[1], pos[2]));
    partitions.add(this.transactions.subList(pos[2], pos[3]));
    partitions.add(this.transactions.subList(pos[3], pos[4]));

    // Compute the result
    double res = 0.0;
    for (List<TransCaixa> list : partitions) {
        for (TransCaixa transCaixa : list) {
            res += transCaixa.getValor();
        }
    }

    return res;
}
```

Listing 6.1: Cálculo iterativo sem e com partições

```
public double sumStream() {
    return this.transactions.stream()
        .mapToDouble(TransCaixa::getValor)
        .sum();
}

public double sumPartitionStream() {
    double res = 0.0;
    Spliterator<TransCaixa> split = this.transactions.stream().spliterator();

    for (int i = 0; i < 4; i++) {
        res += StreamSupport.stream(split.trySplit(), false)
            .mapToDouble(TransCaixa::getValor)
            .sum();
    }

    return res;
}
```

```
public double sumStreamP() {
    return this.transactions.parallelStream()
        .mapToDouble(TransCaixa::getValor)
        .sum();
}

public double sumPartitionStreamP() {
    double res = 0.0;
    Spliterator<TransCaixa> split = this.transactions.stream().spliterator();

    for (int i = 0; i < 4; i++) {
        res += StreamSupport.stream(split.trySplit(), true)
            .mapToDouble(TransCaixa::getValor)
            .sum();
    }

    return res;
}
```

Resultados

Input	(1) sum	(2) sumPartitionStream	(3) sumP	(4) sumPartitionStreamP	(5) sumStream	(6) sumPartitionStreamP
1000000 transactions	0,009782	0,010309	0,020080	0,019041	0,010039	0,010972

Input	(1) sum	(2) sumPartitions	(3) sumP	(4) sumPartitionStream	(5) sumStream	(6) sumPartitionStreamP
2000000 transactions	0,016814	0,020771	0,038469	0,038853	0,016470	0,016523
4000000 transactions	0,036431	0,033048	0,079529	0,073569	0,043543	0,030857
8000000 transactions	0,075378	0,064377	0,152512	0,144809	0,062982	0,058897

Análise e conclusões

7. Determinação da maior transação entre uma determinada hora

Observações

Métodos a testar

```
String biggestTransaction7() {
    List<TransCaixa> transactions = new ArrayList<>(this.transactions);

    transactions.sort(new Comparator<TransCaixa>() {
        @Override
        public int compare(TransCaixa t1, TransCaixa t2) {
            return Double.compare(t1.getValor(), t2.getValor());
        }
    });

    for (TransCaixa transaction : transactions) {
        int hour = transaction.getData().getHour();

        if (hour >= 16 && hour <= 20) {
            return transaction.getTrans();
        }
    }

    return null;
}
```

Listing 7.1: Determinação da maior transação entre uma determinada hora apenas com funcionalidades do JAVA7

```
Optional<String> biggestTransaction8() {
    t.getData().getHour() >= 16 && t.getData().getHour() <= 20;

    return this.transactions.stream()
        .filter(timeInRange)
        .max(Comparator.comparing(TransCaixa::getValor))
        .map(TransCaixa::getTrans);
}
```

Listing 7.2: Determinação da maior transação entre uma determinada hora com auxílio de Streams

Resultados

Input	(1) biggestTransaction7	(2) biggestTransaction8
1000000 transactions	0,255225	0,017511
2000000 transactions	0,598334	0,037735
4000000 transactions	1,174918	0,182393
8000000 transactions	2,133888	0,144549

Análise e conclusões

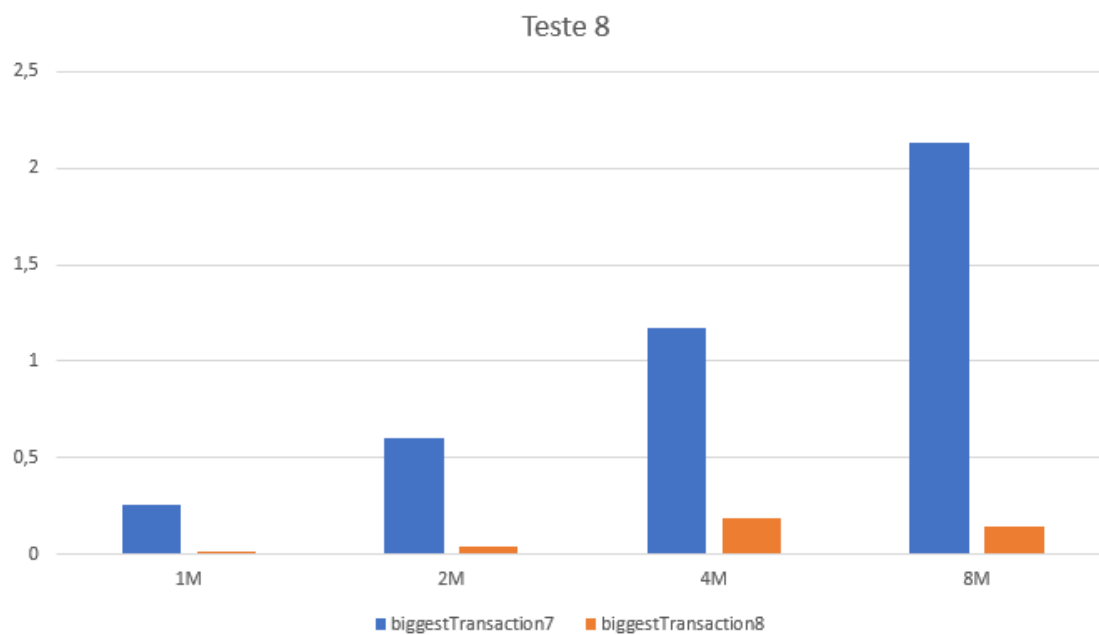


Figura 7.1.: Representação gráfica destes resultados

8. Cálculo do total facturado numa determinada semana do ano

Observações

Métodos a testar

```
public double totalInWeekList() {
    final int week = 12;
    List<List<TransCaixa>> byWeek = new ArrayList<>();

    // Inicializar cada uma das listas
    for (int i = 0; i < 54; i++) {
        byWeek.add(i, new ArrayList<>());
    }

    for (TransCaixa transaction : this.transactions) {
        byWeek.get(transaction.getData()
            .get(ChronoField.ALIGNED_WEEK_OF_YEAR))
            .add(transaction);
    }

    // Calcular total faturado
    double total = 0.0;

    for (TransCaixa transaction : byWeek.get(week)) {
        total += transaction.getValor();
    }

    return total;
}
```

Listing 8.1: Cálculo do total facturado na semana 12 do ano

```
public double totalInWeekStream() {
    final int week = 12;
    Map<Integer, List<TransCaixa>> byWeek = this.transactions.stream()
        .collect(Collectors.groupingBy(
            t -> t.getData().get(ChronoField.ALIGNED_WEEK_OF_YEAR)));
    return byWeek.entrySet().stream()
        .filter(e -> e.getKey() == week)
        .findFirst()
        .map(e -> e.getValue().stream())
```

```
        .mapToDouble(TransCaixa :: getValor).sum())  
        .orElse(0.0);  
}
```

Listing 8.2: Cálculo do total facturado na semana 12 do ano com recurso a streams

Resultados

Input	(1) totalInWeekList	(2) totalInWeekStream
1000000 transactions	0,033511	0,038533
2000000 transactions	0,070856	0,083489
4000000 transactions	0,144393	0,163294
8000000 transactions	0,284878	0,322149

Análise e conclusões

9. Cálculo do total de IVA associado a cada mês

Observações

Métodos a testar

```
public List<Double> iva() {
    List<Double> ivas = new ArrayList<>(13);

    // Initialize the IVAS with 0
    for (int i = 0; i < 13; i++) {
        ivas.add(0.0);
    }

    for (TransCaixa transaction : this.transactions) {
        Month month = transaction.getData().getMonth();
        Double value = transaction.getValor();
        Double iva = value < 20 ? 0.15 * value :
            (value > 20 && value < 20 ? 0.20 * value :
                0.23 * value);
        ivas.set(month.getValue(), ivas.get(month.getValue()) + iva);
    }

    return ivas;
}
```

Listing 9.1: Cálculo do total de IVA para cada mês

```
public Map<Month, Double> ivaStream() {
    return this.transactions.stream()
        .collect(Collectors.groupingBy(t -> t.getData().getMonth()))
        .entrySet()
        .stream()
        .collect(Collectors.toMap(
            Map.Entry::getKey,
            e -> e.getValue().stream()
                .mapToDouble(TransCaixa::getValor)
                .map(x -> x < 20 ? 0.15 * x :
                    (x > 20 && x < 20 ? 0.20 * x :
                        0.23 * x ))
                .sum()
        ));
}
```

```
}
```

Listing 9.2: Cálculo do total de IVA para cada mês, com recurso a streams

Resultados

Input	(1) iva	(2) ivaStream
1000000 transactions	0,027518	0,070057
2000000 transactions	0,058240	0,134494
4000000 transactions	0,107619	0,283225
8000000 transactions	0,210951	0,475746

Análise e conclusões

10. Comparação de performance entre JDK8 e JDK9

Observações

Métodos a testar

Dos testes anteriores, foram seleccionados os seguintes, que manipulam conjuntos enormes de registos TransCaixa:

- **byDateStream** do teste 4;
- **sortList** do teste 4;
- **biggestTransaction8** do teste 8;
- **ivaStream** do teste 10;

Resultados

Input	(1) byDateStream	(2) sortList	(3) biggestTransaction8	(4) ivaStream
1000000 transactions	2,487266	1,074664	0,021984	0,098529
2000000 transactions	4,158751	2,207813	0,048276	0,223570

Input	(1) byDateStream	(2) sortList	(3) biggestTransaction8	(4) ivaStream
1000000 transactions	2,487266	1,074664	0,021984	0,098529
2000000 transactions	4,158751	2,207813	0,048276	0,223570

Análise e conclusões