Java Streams

Characteristics, Phases, Executions and Types of Stream Operations

By:

N.V. Hari Kishore (282790)

Sr Software Engineer B3, Full Stack Engineering Community

Stream is a pipeline of aggregate operations that can be applied to process a sequence of elements.

An aggregate operation is a higher-order function that receives a behaviour in the form of a function or lambda, and that behaviour is what gets applied to our sequence. Java Streams are built around its main interface, the Stream interface, which was released in JDK 8.

Characteristics of a Stream:

The Declarative Paradigm Streams are written specifying what must be done, but not how.

The Lazy Evaluation This basically means that until we call a terminal operation, our stream won't do anything, we will just have declared what our pipeline will be doing.

Single Traversal Once we call a terminal operation, a new stream would have to be generated to apply the same series of aggregate operations.

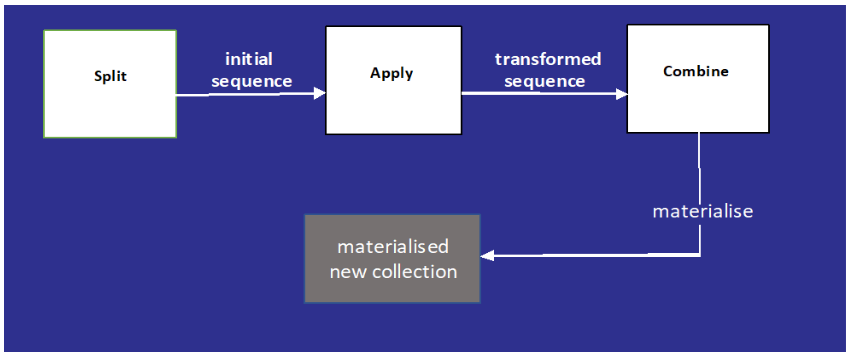
Parallelization Java Streams are sequential by default, but they can be very easily parallelized

Phases of a Stream:

Split Data is collected from a collection, a channel, or a generator function. For example, in this step we convert a data source to a Stream to process our data, which we usually call it a stream source.

Apply Every operation in the pipeline is applied to each element in the sequence. Operations in this phase are called intermediate operations.

Combine Completion with a terminal operation where the stream gets materialized.

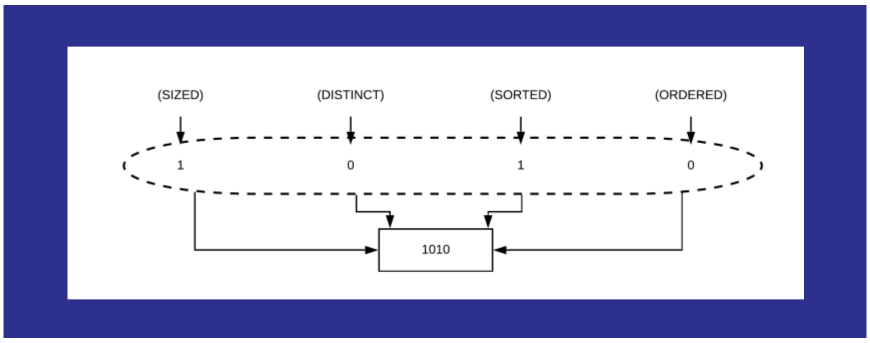


When defining a stream, we define the above steps in our pipeline of operations. They won't get executed until the terminal operation is called.

Stream Internals

Java stream operations are stored internally using a LinkedList structure and in its internal storage structure, each stage gets assigned a bitmap that follows this structure:

FLAG	MEANING
SIZED	The size of the stream in known
DISTINCT	The elements in the stream are unique, no duplicates.
SORTED	Elements are sorted in a natural order.
ORDERED	If the stream has a meaningful encounter order (the order in which elements are streamed should be preserved)



Each operation will clear, set, or preserve different flags; this is important because this means that each stage knows what effects cause itself in these flags and this will be used to make the optimizations.

Example 1

map will **clear SORTED** and **DISTINCT** bits because data may have changed; however, it will always **preserve SIZED** flag, as the size of the stream will never be modified using map.

Example 2

filter will clear **SIZED** flag because size of the stream may have changed, but it'll always preserve **SORTED** and **DISTINCT** flags because filter will never modify the structure of the data.

Example 3

```
final Set<String> set = Set.of("a", "b", "c", "d");
final List<String> list = set.stream().filter(element -> element.length() >
0).distinct().collect(Collectors.toList());
list.forEach(System.out::println);
```

In the above example, Set already guarantees unique elements, So our Stream will be able to cleverly skip distinct stage making use of the flags.

Stream Execution:

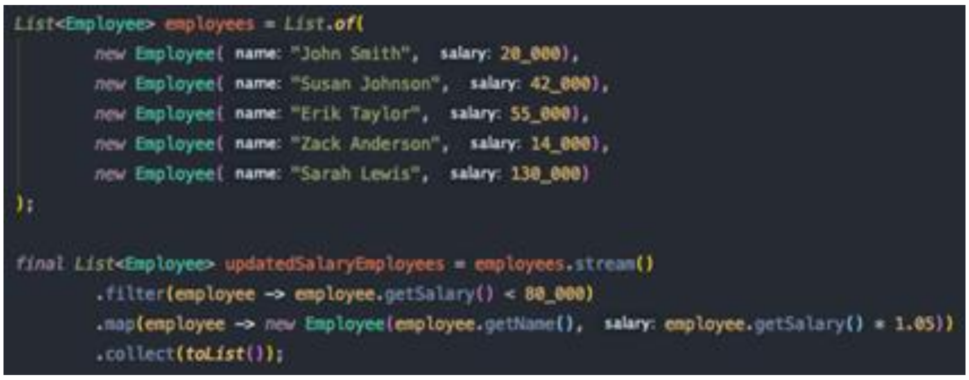
We already know that a Stream is lazily executed, so when a terminal operation gets executed what happens is that the Stream selects an execution plan.

There are two main scenarios in the execution of a Java Stream: when all stages are stateless and when NOT all stages are stateless. To be able to understand this we need to know what stateless and stateful operations are.

Stateless Operations:

A stateless operation doesn't need to know about any other element, to be able to emit a result. Examples of stateless operations are, filter, map or flatMap.

Execution of stateless pipelines:

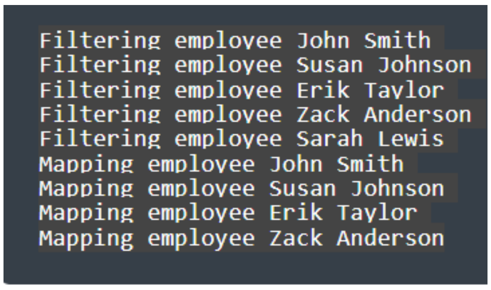


```
List<Employee> employees = List.of(
        new Employee( name: "John Smith",  salary: 20_000),
        new Employee( name: "Susan Johnson",  salary: 42_000),
        new Employee( name: "Erik Taylor",  salary: 55_000),
        new Employee( name: "Zack Anderson",  salary: 14_000),
        new Employee( name: "Sarah Lewis",  salary: 130_000)
);

final List<Employee> updatedSalaryEmployees = employees.stream()
        .filter(employee -> employee.getSalary() < 80_000)
        .map(employee -> new Employee(employee.getName(),  salary: employee.getSalary() * 1.05))
        .collect(toList());
```

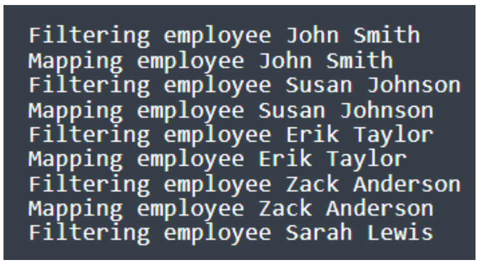
We'd normally think that the collection gets filtered first, then we create a new collection including the employees with their updated salaries and finally we'd collect the result, right? Something like this:

If our initial reasoning was correct, we should be seeing the following:



```
Filtering employee John Smith
Filtering employee Susan Johnson
Filtering employee Erik Taylor
Filtering employee Zack Anderson
Filtering employee Sarah Lewis
Mapping employee John Smith
Mapping employee Susan Johnson
Mapping employee Erik Taylor
Mapping employee Zack Anderson
```

But the above execution and thought process was in-correct, below is the actual execution of stream processing.



```
Filtering employee John Smith
Mapping employee John Smith
Filtering employee Susan Johnson
Mapping employee Susan Johnson
Filtering employee Erik Taylor
Mapping employee Erik Taylor
Filtering employee Zack Anderson
Mapping employee Zack Anderson
Filtering employee Sarah Lewis
```

In reality the elements of a Stream get processed individually and then they finally get collected.

Stateful Operations:

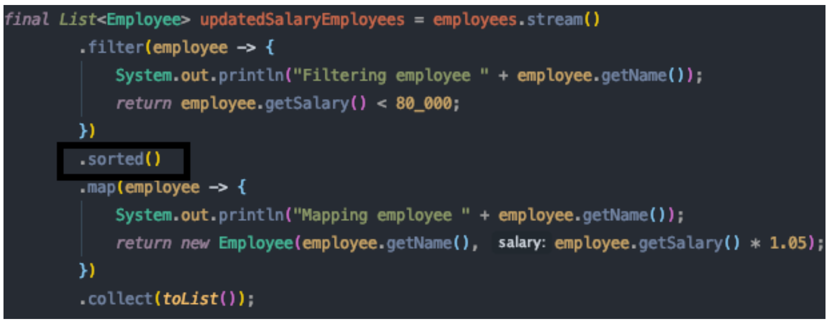
On the contrary, stateful operations need to know about all the elements before emitting a result. Examples of stateful operations are, sorted, limit or distinct.

Execution of pipelines with stateful operations:

Main difference when we have stateful operations is that, a stateful operation needs to know about all the elements before emitting a result. So, what happens is that a stateful operation buffers all the elements until it reaches the last element and then it emits a result.

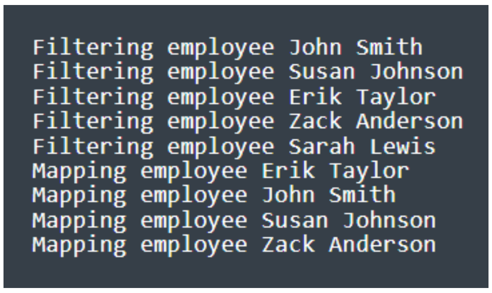
That means that our pipeline gets divided into two sections!

Let’s modify our previous example by adding **sorted()** method with no arguments.



```
final List<Employee> updatedSalaryEmployees = employees.stream()
        .filter(employee -> {
            System.out.println("Filtering employee " + employee.getName());
            return employee.getSalary() < 80_000;
        })
        .sorted()
        .map(employee -> {
            System.out.println("Mapping employee " + employee.getName());
            return new Employee(employee.getName(),  salary: employee.getSalary() * 1.05);
        })
        .collect(toList());
```

Order of Execution:



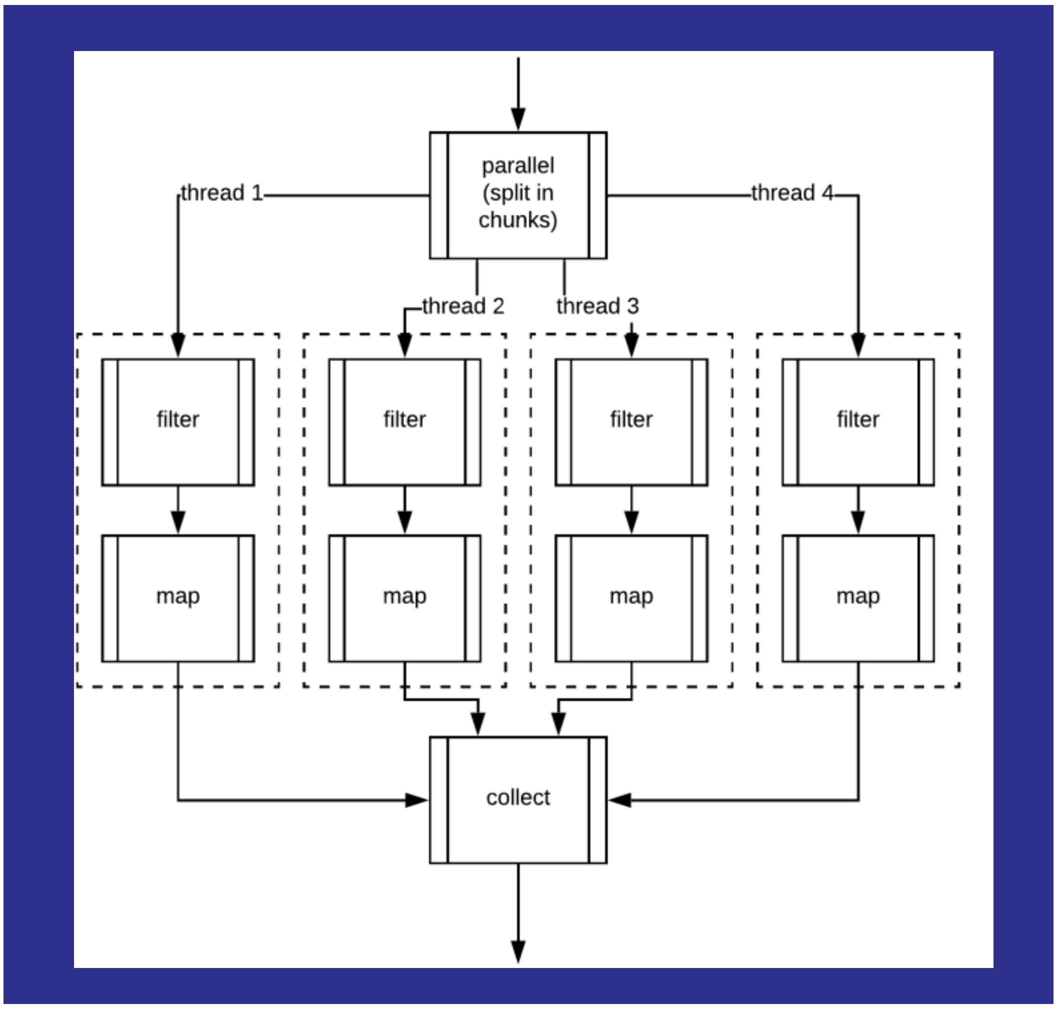
```
Filtering employee John Smith
Filtering employee Susan Johnson
Filtering employee Erik Taylor
Filtering employee Zack Anderson
Filtering employee Sarah Lewis
Mapping employee Erik Taylor
Mapping employee John Smith
Mapping employee Susan Johnson
Mapping employee Zack Anderson
```

Sorted method cannot emit a result until all the elements have been filtered, so it buffers them before emitting any result to the next stage (map).This is a clear example of how the execution plan changes completely, depending on the type of operation.

Execution of parallel streams:
We can execute parallel streams very easily by using **parallelStream** or **parallel**.

Java uses **trySplit** method to try splitting the collection in chunks that could be processed by different threads. In terms of the execution plan, it works very similarly, with one main difference. Instead of having one single set of linked operations, we have multiple copies of it and each thread applies these operations to the chunk of elements that it's responsible for.

Once completed all the results produced by each thread get merged to produce one single and final result!



One last thing to know about parallel streams is that Java assigns each chunk of work to a thread in the common **ForkJoinPool**, in the same way as **CompletableFuture** does.

Short-circuiting terminal operations:

Short-circuiting terminal operations are some kind of operations where we can “short-circuit” the stream as soon as we’ve found what we were looking for, even if it’s a parallel stream and multiple threads are doing some work.

Example limit, findFirst, findAny, anyMatch, allMatch or noneMatch

For example, if we are processing a **noneMatch** terminal operation, we’ll finish the processing as soon as one element matches the criteria.

One interesting fact to mention in terms of execution is that for short-circuiting operations the **tryAdvance** method in **Split Iterator** is called; however, for non-short-circuiting operations the method called would be **forEachRemaining**.

Conclusion

Finally, the decision whether to use Streams or not should not be driven by performance consideration, but rather by readability.

Ex: anyMatch() is a shortcut for filter(Predicate p).findFirst().isPresent()

When it comes to performance parallelStream (although platform dependent) is preferred because it engages multiple cores of the computer and hence can execute multiple iterations simultaneously. The sequential one, in this regard, is generally slower.

Streams are widely used as a replacement for MapReduce operations (projects that are built using Hive, Pig, MapReduce api’s)and they are used in all popular Intranet/Internet Web Applications

Sharing a blog here for more deeper understanding of Streams:

https://blog.hackajob.co/why-you-should-be-using-stream-api-in-java-8/