INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Optimizing sequences traversal and extensibility

DIOGO RAFAEL ESTEVES POEIRA

Licenciado em Engenharia Informática e de Computadores

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Doutor Fernando Miguel Gamboa de Carvalho

Júri:

Presidente: Doutor José Manuel De Campos Lages Garcia Simão

Vogal: Doutor António Rito Silva

January, 2021

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Optimizing sequences traversal and extensibility

DIOGO RAFAEL ESTEVES POEIRA

Licenciado em Engenharia Informática e de Computadores

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Doutor Fernando Miguel Gamboa de Carvalho

Júri:

Presidente: Doutor José Manuel De Campos Lages Garcia Simão

Vogal: Doutor António Rito Silva

January, 2021

Ao meu avô.

Acknowledgments

I would like to thank my supervisor, Fernando Miguel Gamboa de Carvalho, whose expertise was invaluable in the research and ideas achieved with this dissertation. Your insightful feedback taught me much and pushed me to achieve more than I thought I could throughout this period. Thank you for all the guidance and patience.

I would like to also thank the course coordinator, Nuno Datia, for the encouraging words back in 2019, that ultimately pushed me to keep pursuing the goal of finishing the dissertation.

In Addition, I would like to thank my friends who provided me with invaluable moments of happy distractions to rest my mind.

Finally, I would like to thank my close family and especially my parents for their sympathetic ear and wise counsel.

Abstract

Yield generators are a well-known programming feature available in most used programming environments such as JavaScript, Python and many others. They allow easy and compact extensibility on streams operations such as on iterators or enumerable types. Yet, two questions arise about their use: 1) are generators the most efficient choice to extend sequences with new user-defined operations? 2) What if the development programming languages does not provide the yield feature, such as in Java?

The research work that I describe in this dissertation aims to answer these two questions. To that end, I analyzed two different programming languages designs for a sequence type, Java and Javascript. Also, I studied the state-of-the-art alternatives to the out-of-the-box sequences included in each language, in a set of features, devising benchmarks to analyze their performance with real world use-cases, available for developers to use when choosing a sequence type according to their needs.

Not only that but, I also propose my own solution of a sequence type, based on a minimalist design that both allows for verboseless extension as well as fluent chaining of new operations. My proposal aims to be as simple and transparent as possible so the developer may clearly understand what he is using.

Finally, I answer the question "When should you use parallelism?" with a set of benchmarks that compare Java `Streams` sequential processing with its parallel counterpart.

Keywords: Framework, Yield, Generators, Lazy sequences, Iterators, Collections, Extensions

Resumo

Geradores *yield* são uma característica de programação bem conhecida, disponível na maioria dos ambientes de programação usados, como JavaScript, Python e muitos outros. Permitem uma extensibilidade fácil e compacta em operações de *streams*, como em iteradores ou tipos enumeráveis. Ainda assim, surgem duas questões sobre a sua utilização: 1) Os geradores são a melhor escolha para estender sequências com novas operações definidas pelo programador? 2) E se as linguagens de programação de desenvolvimento não fornecerem geradores *yield*, como em Java?

O trabalho de pesquisa que descrevo nesta dissertação visa responder a essas duas questões. Para tal, analisei dois desenhos de tipo de sequência de linguagens de programação diferentes, nomeadamente, Java e Javascript. Além disso, estudei as alternativas mais utilizadas às sequências incluídas em cada linguagem, num conjunto de características, criando benchmarks para analisar o desempenho de cada uma em casos de utilização baseados no mundo real, disponíveis para cada programador poder usar quando quiser escolher um tipo de sequência de acordo com suas necessidades.

Para além disto, proponho a minha própria solução para um tipo de sequência, baseado num desenho minimalista que permite não só a extensão concisa da sua *API* como o encadeamento fluente de operações definidas pelo utilizador. A minha proposta tem como objectivo ser quão simples e transparente quanto possível, para que qualquer programador consiga perceber claramente aquilo que está a usar.

Por fim, respondo à questão "Quando se deve usar paralelismo?"com um conjunto de *benchmarks* que comparam o processamento sequencial das `Streams` do Java com o seu processamento paralelo.

Palavras-chave: Framework, Yield, Geradores, Sequencias Lazy, Iteradores, Colecções, Extensões

Contents

List of Figures

List of Tables

List of Listings

1

Introduction

In this work, we propose an alternative design for sequences named `Tinyield`, which has the advantage of providing an extensible approach similar to *generators* [9]. The `Tinyield` approach can be implemented in any programming language with higher-order functions support. We tested it in C#, Java and JavaScript where it shows advantages in performance, extensibility and code conciseness in different use cases over the use of existing alternatives.

1.1 Motivation

Most sequence implementations, such as `java.util.stream.Stream` in Java or `System.Collections.IEnumerable` in .NET, support functional-style operations on sequences of elements. For example, given a variable `pastWeather` referring a sequence of `Weather` objects (i.e. `Stream<Weather>`), each one with day weather information, then we can filter sunny days, then produce a sequence of temperatures and finally select the first five elements, through a chain of operations of `filter`-`map`-`limit`, which results in the *pipeline* presented in Listing 1.1 or Listing 1.3. In this case, we must provide to some operations (i.e. `filter` and `map`) another function specifying (*telling*) what we want to do with each `Weather` object (i.e. `Weather::isSunny` and `Weather::getTempC`)[1]. On the

[1] `::` is the Java keyword for *method references*, also known as *function pointers* in other languages.

other hand, the imperative approaches presented in Listing 1.2 and Listing 1.4 *ask* for the items (*pull*) from the data source and invoke the functions to validate and transform those items (i.e. `w.isSunny()` and `w.getTempC()`). Note the difference between using a method reference with the `::` or the lambda function `w => ...`, in Java and in .NET, respectively, instead of invoking a method with the `()` suffix.

```
IntStream top5Temps = pastWeather
    .filter(Weather::isSunny)
    .mapToInt(Weather::getTempC)
    .limit(5)
```

Listing 1.1: Tell how to filter, how to map, and how many items to select through a stream pipeline.

```
List<Integer> top5Temps = new List<>();
Iterator<Integer> iter =
  pastWeather.iterator();
while(iter.hasNext()){
  WeatherInfo w = iter.next();
  if(w.isSunny()) {                // filter
    top5Temps.add(w.getTemp())     // map
    if(top5Temps.size() >= 5)      // limit
      break;
  }
}
```

Listing 1.2: Ask for items from a data source, check if they are valid, map and add to the resulting collection.

```
IEnumerable<int> top5Temps = pastWeather
    .Where(w => w.isSunny())
    .Select(w => w.getTempC())
    .Take(5);
```

Listing 1.3: Equivalent example to Listing 1.1 but in .NET using Linq

```
List<int> top5Temps = new List<int>();
IEnumerator<WeatherInfo> enumerator =
     pastWeather.AsEnumerable().
     GetEnumerator();
while(enumerator.MoveNext()) {
  WeatherInfo w = enumerator.Current;
  if(w.isSunny()) {                // Where
    top5Temps.Add(w.getTempC());   // Select
    if(top5Temps.Count >= 5) {     // Take
      break;
    }
  }
}
```

Listing 1.4: Equivalent to Listing 1.2 but in .NET

However, using just the Standard APIs, we soon come to the point where it is hard to express something, particularly when we need to build a new stream operation which does not exist in the current APIs. For example, consider now that we would like to include an additional operation in the pipeline that merges series of adjacent temperatures with the same value. We call this new operation

`collapse` and we want to interleave it between the `map` and `limit`, resulting in a chain of `filter-map-collapse-limit`. Yet, a `collapse` operation does not exist in either the `Stream` nor in the `Linq` APIs and we have to develop it. Note that we use the `collapse` just as an example of one of the operations that are still missing in `Stream` API, such as `zip`, `flatten`, `groupBy`, `intersection` and many others. Moreover, we cannot predict beforehand all the use cases a programmer needs, so there will always be missing stream operations.

But developing a stream operation is much harder than developing the same operation for a data-structure such as a Java `Collection`. For comparison, in Listing 1.5 we show a straightforward implementation of the corresponding version of `collapse` for collections (i.e. `colCollapse`).

```
1  public static <T> Collection<T> colCollapse(Iterable<T> src) {
2    Collection<T> res = new LinkedList<>();
3    Iterator<T> iter = src.iterator();
4    if (!iter.hasNext()) return res;
5    T curr, prev;
6    res.add(prev = iter.next());
7    while (iter.hasNext()) {
8      if (!prev.equals(curr = iter.next()))
9        res.add(prev = curr);
10   }
11   return res;
12 }
```

Listing 1.5: Implementation of a `colCollapse` method for collections.

A naive implementation of `collapse` for sequences could take advantage of the existing `colCollapse` method. However, such solution would incur in the overhead of creating an intermediate in-memory data structure (i.e. `res` of Listing 1.6). Note that the resulting collection is useless when we use the `collapse` method as an intermediate operation in a stream pipeline, which coalesces all operations into a single pass.

```
1  public static <T> Stream<T> collapse(Stream<T> source) {
2    Collection<T> res = colCollapse(source::iterator);
3    return res.stream();
4  }
```

Listing 1.6: A naive implementation of `collapse` that reuses the `colCollapse` method.

The implementation of Listing 1.6) returns a new sequence compatible with the `Stream` API, which provides the auxiliary stream operations (e.g. `filter`, `map`, etc), but does not satisfy one of the core features of a `Stream`: *laziness*. To implement a `collapse` method that preserves the main characteristics of a stream

pipeline we should create an auxiliary class, which implements the `Spliterator<T>` interface to specify the behavior of the resulting stream (e.g. class `Collapser` of Listing 1.7). In turn, the `collapse` method creates a new instance of this auxiliary `Spliterator`, which is wrapped into a new `Stream` object.

```
1  public static <T> Stream<T> collapse(Stream<T> source) {
2    return StreamSupport.stream(
3      new Collapser<T>(source.spliterator()), false);
4  }
5  class Collapser<T> extends AbstractSpliterator<T> implements Consumer<T> {
6    private final Spliterator<T> source;
7    private T curr = null;
8
9    Collapser(Spliterator<T> source) {
10     super(Long.MAX_VALUE, source.characteristics() & (~(SIZED | SUBSIZED)));
11     this.source = source;
12   }
13   @Override
14   public boolean tryAdvance(Consumer<? super T> cons) {
15     T prev = curr;
16     boolean hasNext;
17     while ((hasNext = source.tryAdvance(this)) && curr.equals(prev)) { }
18     if(hasNext) cons.accept(curr);
19     return hasNext;
20   }
21   @Override
22   public void accept(T item) {
23     curr = item;
24   }
25 }
```

Listing 1.7: Implementation of a `collapse` method, which merges series of adjacent elements of a `Stream`.

For now, we will not give further details about the implementation of Listing 1.7 and we just want to highlight the overheads incurred by this solution. Despite being one of the shortest implementations for a `collapse` stream operation, it is a verbose solution that reduces source code readability in comparison with the `colCollapse` version of Listing 1.5. Moreover, the reduced readability turns the source code maintainability more difficult too.

Second, the iteration logic is defined in the `tryAdvance` method, which according to Java documentation, *traverses elements individually*. This method is further wrapped into an auxiliary function that buffers every item that is passed to the `cons` consumer (line 18 of Listing 1.7). This wrapper is created from the use of `StreamSupport` utility class (line 2 in Listing 1.7) and the copying into the buffer is required because of the internal iteration approach followed by a stream pipeline.

Finally, despite being a verbose implementation, the solution of Listing 1.7 is still missing the *bulk traversal* method (i.e. `forEachRemaining`) which may optimize

traversal for aggregate operations, such as `reduce`. Once missing an implementation for the `forEachRemaining`, then the default implementation repeatedly invokes the `tryAdvance`, which in turn has an increased overhead due to internal buffering.

Concluding, this extension approach incurs in: 1) implementation verbosity, 2) additional buffering overhead and 3) wasted the fast-path iteration provided by the single bulk computation i.e. `forEachRemaining` of `Spliterator<T>`.

1.2 Alternatives

The limited extensibility of the `Stream` API is mitigated in other languages such as C#, Python or JavaScript, which provide the concept of *generators*. Generators first appeared in CLU [9] and are functions that may yield multiple values instead of returning one value a single time. Generators are lazily computed and thus, calling a generator function does not execute its body immediately. In C#, a generator returns an iterator object compatible with `IEnumerable<T>` interface. Just like Java `Streams`, the elements produced by a generator are computed on demand only when that sequence is iterated by a *for each* loop. For example, given a generator method `Foo` (e.g. `IEnumerable<T> Foo<T>(){...}`), the resulting elements are just computed only when we iterate over the sequence returned by `Foo` (e.g. `var seq = Foo(); foreach(var elem in seq){...}`).

C#, provides a `yield` keyword that allows generators to be implemented concisely with no verbosity. In C# the creation of a new `Collapse` query operation can be concisely implemented through the use of the `yield` statement. The C# implementation of the `Collapse` method in Listing 1.8 shares the advantages of the concise implementation of `colCollapse` for Java collections that we present side by side in Listing 1.9 (note that both algorithms are similar and every statement in one implementation has a corresponding one in the other). But the C# method `Collapse` returns a sequence (i.e. `IEnumerable`) instead of a data structure (i.e. `Collection`) and thus it suppresses the overhead of creating a in-memory collection (i.e. `res` of Listing 1.9). So, it shares the benefits of returning a sequence: *lazily computed*, without the additional verbosity incurred by the implementation of Listing 1.7.

```
IEnumerable<T> Collapse<T>(
   IEnumerable<T> src) {
 IEnumerator<T> iter = src.GetEnumerator()
 if (!iter.MoveNext()) yield break
 T prev
 yield return prev = iter.Current
 while (iter.MoveNext()) {
  if (!prev.Equals(iter.Current))
    yield return prev = iter.Current
 }

}
```

Listing 1.8: Implementation of a `Collapse` method for .NET generators.

```
<T> Collection<T> colCollapse(... src) {
 Collection<T> res = new LinkedList<>()
 Iterator<T> iter = src.iterator()
 if (!iter.hasNext()) return res
 T curr, prev;
 res.add(prev = iter.next())
 while (iter.hasNext()) {
  if (!prev.equals(curr = iter.next()))
    res.add(prev = curr)
 }
 return res
}
```

Listing 1.9: Implementation of a `colCollapse` method for Java collections.

1.3 Goals

The hypothesis is that we can achieve an effective type of sequence while maintaining an approach that is more simple and transparent to the developer than the ones provided by nowadays main programming languages. In opposition, Java `Streams` approach to add a user-defined operation is not only a verbose endeavour but the code written is also wrapped in further decorators behind the scenes.

Comparing to Java `Streams`, the `Tinyield` solution:

1. discards the partition capabilities for parallel processing;
2. gives primacy to the fast-path iteration protocol from the bulk traversal method (i.e. `forEachRemaining` of the `Spliterator`);

Given these requirements, we propose that a sequence can be expressed by a single function specified by the type `Traverser<T>`, which is an interface with only one abstract method `traverse(Yield<T> yield)`. Additionally a class `Query<T>` provides all auxiliary query operations (e.g. `filter`, `map`, etc). The `Yield<T>` type behaves just like a consumer equivalent to the java interface `Consumer<T>`. In Listings 1.10 and 1.11 we depicted the layout of the main `Tinyield` interfaces: `Traverser` and `Yield`, which are described in more detail in Chapter 3.

```java
public interface Traverser<T>
{
  void traverse(Yield<T> yield);
}
```

Listing 1.10: Interface `Traverser`.

```java
public interface Yield<T>
{
  public void ret(T item);
}
```

Listing 1.11: Interface `Yield`

The design of `Tinyield` is based on the idea proposed by Henry Baker [1] that iteration can be trivially achieved using an higher-order function with function closures. In this way, and according to `Tinyield` approach a new query operation (such as `collapse`) can be simply achieved with a single method implementation, just like an anonymous function provided by a lambda expression. The type of this function is `Traverser`. For example, a `collapse` operation can be expressed by the following method:

```java
class Collapse {
  private U prev = null;
  <U> Traverser<U> collapse(Query<U> src) {
    return yield -> {
      src.traverse(item -> {
        if (prev == null || !prev.equals(item))
        yield.ret(prev = item);
      });
    };
  }
}
```

Listing 1.12: Collapse implementation[2]with `Tinyield`

This solution shares most of the advantages of generators and the skeleton of the algorithm is similar to that one presented in Listing 1.8, which makes use of the C# `yield` keyword. Following such approach we achieve a solution that preserves the core characteristics of sequences and satisfies code conciseness in the implementation of new operations.

The source-code of `Tinyield` is publicly available in the Github repositories:

- github.com/tinyield/jayield for Java.
- github.com/tinyield/tinyield4ts for Typescript and Javascript.

We evaluate our proposal, both in Java and Javascript, with a realistic use cases based on data from *APIs* such as World Weather Online[3], REST Countries[4] and

[2]This implementation is not thread safe.
[3]https://www.worldweatheronline.com/developer/
[4]https://restcountries.eu/

Last.fm[5], where our approach is up to three fold faster than a `Spliterator<T>` implementation and has identical worst-case performance.

With regards to the loss of partition capabilities for parallel processing, we analyzed `Stream` *parallel()* behaviour through the use of benchmarks devised for the purpose, to better understand when it becomes an advantage and if it is a significant loss for all use-cases or just a particular niche.

The `Tinyield` solution can be applied to any other programming language with higher-order functions support. In this work we present two minimalistic implementations of `Tinyield` in Java and Typescript, which show the effective readability of the query operation definition based on a `Traverser` type function and also present better performance than the existing operations based on iterators or generators.

1.4 Outline

This document has a total of 5 chapters and it is organized as follows:

- Chapter 1 *Introduction*. This chapter establishes what is the subject of concern of this work. Specifically, it introduces the core concepts about sequences to clearly understand the following sections. Furthermore, it presents the main reasons of the overheads incurred by extending Java `Streams` with new operations. Additionally, It also describes the main contributions of this work and how this document is organized.
- Chapter 2 *Background and State of the Art*. In this chapter the main properties that characterize the design of sequences are addressed. After that, the related work and existing alternative libraries to Java `Streams` as well as Javascripts `Array` are presented.
- Chapter 3 *Minimalist Sequence traversal design*. In this chapter the basis for the `Tinyield` solution is explained.
- Chapter 4 *Benchmarks and Performance Analysis*. This chapter explains what tests were devised to analyse the sequence alternatives for Java and Javascript, including the `Tinyield` solution, as well as the ones devised to analyse Java `Stream` *parallel()*, how the tests were performed and analyse the results.

[5]https://www.last.fm/api/

- Chapter 5 *Conclusions*. This chapter summarizes the main contributions of this work and discusses some directions for future development.

2

Background and State of the Art

This chapter addresses the context of this document, it does so starting in Section 2.1, where it discusses related work to the one of this thesis. Following that it describes the main properties that characterize the design of sequences in Section 2.2. Further ahead, in Section 2.3, it describes the current State of the Art in Java, Javascript and Typescript regarding sequence type alternatives. Finally it goes over the Internals of Java's `Stream` illustrating how all the pieces work and how they come together.

2.1 Related work

Java's `Streams` are not a new idea. In fact, they are heavily inspired by the concept of *lazy lists*, also known as *streams*, first described in 1965 by Landin [7]. It was Landin who proposed the use of *delayed evaluation* to avoid "item-by-item" representation of collections. Friedman and Wise [5] introduced lazy lists in Lisp in 1976 and the idea was then adopted in other languages too, either as a fundamental data structure, as in Haskell, or as an added capability, as in Python's generators.

The idea of a single iteration method was already introduced in Python 2.2 [17], where iterators provide a single method `next` that returns the next element in a sequence, or raises an exception when no more elements are available. This

feature is described in the proposal PEP 234 (Python Enhancement Proposal 234) Iterators [19].

The advantages of a single bulk computation were highlighted in [1], where H. Baker shows how higher-order functions, taking as argument functions which are closed over their free lexical variables (*closures*) can be used to provide iteration capabilities.

The plethora of programming languages bring with them a plethora of sequence type implementations, each with its own nuances. Kotlins implementation of `Sequence` is a simpler approach to a sequence type than Java `Streams`. `Sequence` in of itself is the same as a Java `Iterable`, it provides a single method in its interface *iterator()* that specifies how the iteration of the `Sequence` is done. Each intermediate operation then, needs only provide a new implementation of the `Sequence` interface. In addition to that, `Sequence` operations are defined through extension methods, meaning the interface is open for extension, this approach is quite similar to the one provided by .NET with the `IEnumerable` and `IEnumerator` interfaces. Kotlins `Sequence` also made a few optimizations like in-lining most terminal operations at compile time, creating one less lambda and one less indirection when processing elements, and by not creating element wrappers like `Optionals` when handling missing values, leading to less overhead. `Sequence`, although simple, won't be enough for every developer, there will always be use-cases that require something that a sequence type does not provide, such as easy partitioning and parallelism capabilities in this particular case, which in turn are provided by Java `Streams`.

Javascript supports, since `EcmaScript5` in 2009, operation chaining over sequences, in other words, sequence pipelines. Javascripts sequence type is the `Array` type, distinguishing itself from other sequence type implementations by having an eager approach. Generators and the *yield* keyword were later introduced with `ES6` in 2015, yet, no lazy sequence type implementations were provided by this new standard either forcing developers to look for this feature in third-party libraries.

Evaluating sequence types performance through the use of benchmarks is not a new field, there have been many use-cases tested in this manner. One example is kotlin-benchmarks[13], which provide benchmarks over Kotlins features such as the use of `Sequence's` various methods with different data types, as well as the use of these features in Java. Another example are the benchmarks devised by Angelika Langer and Klaus Kreft for the JAX London Conference[8] of

2015 with the aim of better understanding how Java `Streams` perform and try to understand when `Stream` *parallel()* outperforms sequential processing of the same `Stream`. Nicolai Parlog also tackled this point in his benchmarks on Parallel `Stream` Vectorization [10], evaluating the performance gained using `Stream` *parallel()* when computing factorials.

2.2 Sequences design alternatives

Over the last 20 years, the intense research on managed runtime environments produced a number of implementations that explore a vast design space of choices for several key aspects of sequences design. In the following, we shall discuss sequences design alternatives, namely:

- *Eager* versus *lazy* processing
- *Deforestation* (also know as *fusion*)
- *Pull* versus *push* flow
- Internal versus external iteration
- *Bulk* versus individually traversal
- Once off versus multiple traversals
- *Synchronous* versus *Asynchronous*
- *Hot* versus *Cold*
- *Query* operations

  Type: *intermediate* and *terminal*

  Short-circuiting

  Pipeline: *Fluent* vs *Nested*

2.2.1 *Eager* versus *lazy* processing

The differences between the approaches of Listings 1.1 and 1.2 go much beyond the end usage API. The *query methods* used in the approach of Listing 1.1 (i.e. `filter`, `map` and `limit`) have the particularity of being *lazy*, which means that

they do not perform any processing on the source elements (i.e. `pastWeather`) when they are called. Instead, invoking such methods merely adds another operation to the end of the stream pipeline. Note that streams are immutable and thus, each query method always returns a new stream that results from the composition of the previous stream with an additional operation.

The resulting pipeline

```
pastWeather.filter(Weather::isSunny).mapToInt(Weather::getTempC).limit(5)
```

of Listing 1.1 can be decomposed in the stages depicted in Figure 2.1.

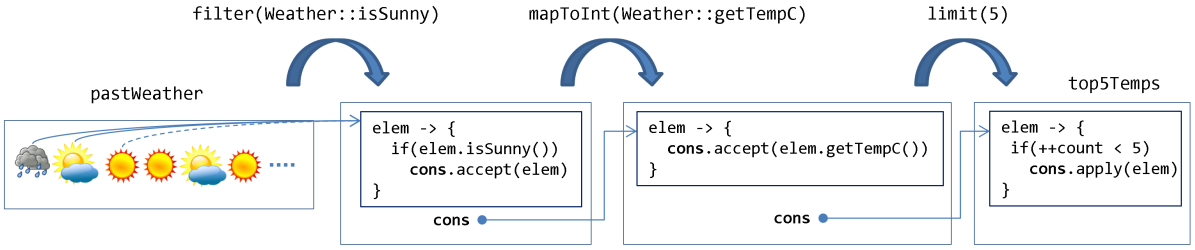

Figure 2.1: `Stream` pipeline resulting from a chain of `pastWeather-filter-map-limit`.

This contrasts with the behavior of collections which are in-memory data structures that hold all values contained by a collection. For example, considering that `colOfPastWeather` is a collection of `Weather` objects (i.e. `Collection<Weather>`) and `colFilter` and `colMap` are the equivalent versions of the `filter` and `map` methods for collections, then selecting the temperatures on sunny days through a chain of `colFilter-colMap` requires in-memory data structures to store the values produced *eagerly* by each operation, as depicted in Figure 2.2.

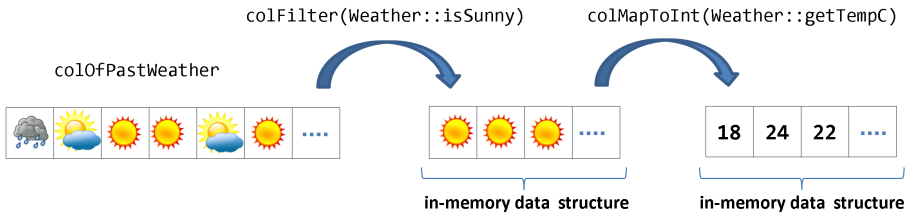

Figure 2.2: In-memory data structures produced by the eager execution of `colFilter` and `colMap`.

On the other hand, for a stream pipeline, the elements of the resulting sequence (i.e. `18`, `24`, `22`, etc.) are computed on demand–*lazily*–only when that sequence is iterated, for example by a `forEach` operation, such as: `top5Temps.forEach(System.out::println)`. Only when the `top5Temps` is iterated does the pipeline actually perform the work, apply the intermediate operations and compute the final sequence `18`, `24`, `22`, etc, depicted in Figure 2.3.

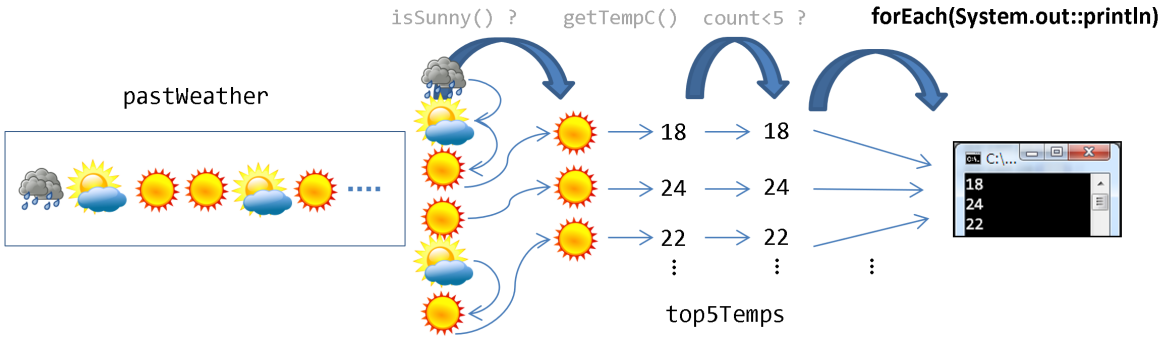

Figure 2.3: The elements of sequence `top5Temps` being computed when that sequence is iterated by a *forEach* operation.

2.2.2 *Deforestation* (also know as *fusion*)

`Streams` created from intermediate operations such as `filter`, `map` or `limit` can usually be coalesced and processed into a single pass. Each consumer resulting from an intermediate operation can be potentially inlined in the previous consumer, and so on, until the stream's data source is (e.g. `pastWeather`), as depicted in Figure 2.4. The whole pipeline can be fused into a single consumer which is connected directly to the source. This technique is known as *deforestation*[18], or *fusion*.

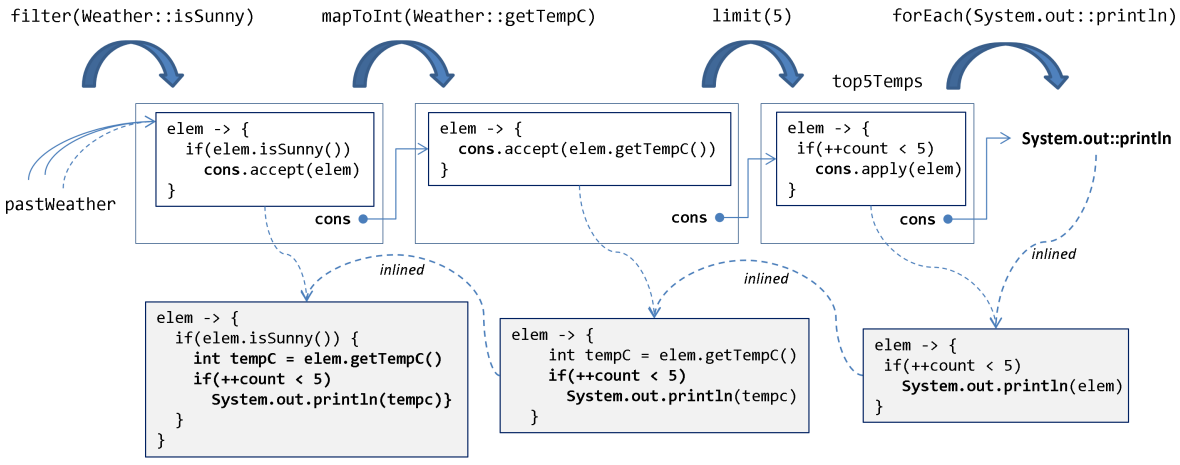

Figure 2.4: Each consumer in a stream pipeline can be fused with its predecessor, and so on, forming a single consumer connected directly to the stream data source.

2.2.3 *Pull* versus *push* flow

A sequence can be iterated through a *push* flow, where elements are pushed to a consumer function, or a *pull* flow, where elements are pulled by the algorithm that requires them.

These types of flows can be differentiated by who initiates the iteration. The *pull* flow is initiated by the consumer, which retrieves the element and then does any processing required. On the other hand, in a *push* flow the producer initiates the iteration of elements and the consumer merely states what should be done with the elements.

```
1 while(data.hasNext()){
2    Weather elem = data.next();
3    if(elem.isSunny() {
4       int tempC = elem.getTempC();
5       if(++count < 5) {
6          System.out.println(elem);
7       }
8    }
9 }
```

Listing 2.1: *Pull* flow

```
1 data.filter(Weather::isSunny)
2    .map(Weather::getTempC)
3    .limit(5)
4    .forEach(System.out::println);
```

Listing 2.2: *Push* flow

In Listing 2.1 elements are pulled from the data sequence and then processed, to do this the developer makes use of an iterator, first asking `data` if it has a next element by calling the `hasNext` method, and then retrieving the element by calling the `next` method in the case that it does have a next element. Finally, after the element is retrieved it is processed by whatever code the developer wrote. By contrast, in Listing 2.2, elements that the sequence produces are simply pushed into the consumer passed to the `forEach` method, in other words, the developer only has to define what should be done with each element, the process of iterating through the sequence's elements is done internally.

2.2.4 Internal versus external iteration

Iteration of a sequence can be either internal or external. We've already exposed examples of both these types of iteration in the previous example, but here are some more details.

Internal Iterators, also known as passive or implicit iterators, manage the iteration in the background leaving the programmer to specify what to do with each

element of the sequence, rather than managing the iteration. So if we wanted to read a collection of news pieces with an internal iterator, Listing 2.3 is what it would look like. This type of Iterator is strongly linked to a *push* flow, as we can match the same type of iteration to Listing 2.2.

```
1 void readTheNews(Collection<NewsPiece> news) {
2   news.forEach(newsPiece -> read(newsPiece));
3 }
```

Listing 2.3: Internal Iterator

External iterators, also known as active or explicit iterators, are the exact opposite of Internal iterators, passing to the programmer the responsibility of iterating over the elements, and making sure that this iteration takes into account the total number of records. Going back to our news reading example, Listing 2.4 shows how the same example as Listing 2.3 is done through external iteration. This type of iteration is strongly related with a *pull* flow, as we've seen the same type of iteration used in Listing 2.1.

```
1 void readTheNews(Collection<NewsPiece> news) {
2    Iterator<NewsPiece> iterator = news.iterator();
3    while(iterator.hasNext()){
4       read(iterator.next());
5    }
6 }
```

Listing 2.4: External Iterator

2.2.5 *Bulk* versus individually traversal

When traversing a sequence, one can traverse the whole sequence in a *bulk* or individually traverse element by element. While traversing a sequence in *bulk* is more performant, due to the fact that the context of the code running doesn't switch as much , traversing it individually grants the programmer extra control over when and if another element should be processed.

Both our *push* flow example in Listing 2.2 and our internal iteration example in Listing 2.3 showcase *bulk* traversal. Listings 2.5 and 2.6 showcase what an equivalent solution would be if we decided to individually traverse.

```
1 Spliterator<Weather> it = data
2     .filter(Weather::isSunny)
3     .map(Weather::getTempC)
4     .limit(5)
5     .spliterator();
6 while(it.tryAdvance(out::println));
```

Listing 2.5: Individually traversal equivalent to Listing 2.2

```
1 void readTheNews(Stream<NewsPiece> news) {
2     Spliterator<NewsPiece> it = news.
          spliterator();
3     while(it.tryAdvance(this::read));
4 }
```

Listing 2.6: Individually traversal equivalent to Listing 2.3

Both Listings 2.5 and 2.6 make use of Java's `Spliterator`, which, among other things, is used to traverse a sequence, it may traverse elements individually through *tryAdvance* method, or in *bulk* through the *forEachRemaining* method. Notice however, that the power to stop the iteration rests with the programmer, contrary to what happens when a *forEach* is called, such as the example given in Listing 2.2, where all elements are traversed with no method for interruption. In this particular case we're showcasing how to individually traverse elements, using the *tryAdvance* method, which is one of the ways to individually traverse a sequence, but not the only way, Listings 2.1 and 2.4 also showcase how to do it, using an iterator to do so.

2.2.6 Once off versus Multiple traversals

Once off sequences can only be traversed once, after that you can no longer use them, like Java `Streams`. On the other hand, the `Iterable` and `IEnumerable` protocols of Java and C# respectively, both allow for multiple traversals. At first this might seem like an odd decision to make, why wouldn't you want to be able to make multiple traversals on a sequence? Well, there are quite a few reasons actually. In the instance that the sequence's source isn't once off, like a Java's `Collection`, we could traverse it multiple times without it being a problem, each traversal would require a call to *iterator()*, which would produce an independent iterator for each traversal, making all active traversals independent of each other. But what if the source was once off, like reading from the file system or an making an HTTP request, what will traversing through the sequence again do? Maybe the first traversal should yield all the values but the second and subsequent traversals should be empty, or maybe all traversals should share the same values, but then what happens when one traversal gets ahead of another? It would require some sort of buffer to keep values ready for someone down the line, that may or may not ask for them. Not only that but if the sequence is lazy,

traversing it again would imply recomputing the whole sequence which could prove to be a very heavy toll on an algorithm. To exemplify, the cost of *size()* or *count()* on a sequence such as `ArrayList` is *O(1)*, but if you call the same method on a lazily-filtered sequence, it has to run the filter over source, which makes this operation *O(n)*, worse, it has to traverse the source every time it's called.

2.2.7 Synchronous versus Asynchronous

A sequence is labeled Synchronous or Asynchronous depending on how it behaves when a *terminal* operation is called. If we look at the *forEach(consumer)* operation, a Synchronous sequence returns only when the `Consumer` has been called for every element. In opposition an Asynchronous sequence will return immediately and the *consumer* will be called for each element once a `Thread` is available to do so. This means that an Asynchronous sequence will not block the execution of code in the `Thread` that calls the *terminal* operation. We can find an example of each type in Listings 2.7 and 2.8.

```
1 IntStream
2     .rangeClosed(1, 5)
3     .forEach(System.out::println);
4 System.out.println("Subscribed!");
5
6 // Output:
7 // 1
8 // 2
9 // 3
10 // 4
11 // 5
12 // Subscribed!
```

Listing 2.7: Example of a synchronous sequence.

```
1 Observable
2     .intervalRange(1, 5, 0, 200, TimeUnit.
          MILLISECONDS)
3     .subscribe(System.out::println);
4 System.out.println("Subscribed!");
5
6 // Output:
7 // Subscribed!
8 // 1
9 // 2
10 // 3
11 // 4
12 // 5
```

Listing 2.8: Example of a asynchronous sequence.

In Listing 2.7 we find an example of what a synchronous sequence would look like, in this case we have a sequence whose elements range from one to five, which are immediately traversed when *forEach* is called. On the other hand, Listing 2.8 shows an equivalent example to Listing 2.7, but asynchronous. This means that when *subscribe* is called, it returns immediately, hence "Subscribed!" being the first message outputted, after that, the values from one to five are emitted with a two hundred millisecond interval.

2.2.8 *Hot* versus *Cold*

Cold sequences are sequences that when a *terminal* operation is called start processing all the data and operations that were associated to the sequence's pipeline, the examples above all illustrate *cold* sequences. On the other hand, a *hot* sequence is one that starts producing results as soon as it is created, even if no one is making use of those results, this could be for instance an event emitter on a web page, events are being fired even if no one is listening.

```
1 ConnectableObservable<Long> nrs = Observable
2          .intervalRange(0, 20, 0, 1000, TimeUnit.MILLISECONDS)
3          .publish();
4 nrs.connect(); // Start emition
5 Thread.sleep(5000);
6 nrs.subscribe(System.out::println); // 5, 4, 6, 7, 9, ...
7 Thread.sleep(5000);
8 nrs.blockingSubscribe(System.out::println); // 10, 10, 11, 11, 12, 12 ....
```

Listing 2.9: Example of a *hot* sequence.

Listing 2.9 showcases what a *hot* sequence would look like. It starts emitting values when *connect* is called, even though there's no one *subscribed* to this sequence. This becomes evident when we make the first subscription on line 4 and the first value we output is 5. The same happens when a new subscription is made on line 8 and the first value outputted is 10, this time outputted twice since we have two subscriptions.

2.2.9 *Query* operations

Sequences often need to be transformed or queried to get to the data that we're actually after. These operations are what we're calling *query* operations, and they can be categorized in two types, *intermediate* or *terminal*, both of which can be also categorized as Short-Circuiting or not. The chain of query operations performed on a sequence are called a Pipeline, and depending on the way the code is written the pipeline can be either classified as Fluent or as Nested. This pattern is thoroughly described by Martin Fowler in his well-known article from 2015, "Collection Pipeline"[4].

2.2.9.1 Type: *intermediate* and *terminal*

A sequence's pipeline is composed of a source, one or more intermediate operations, and a terminal operation. Sources can be anything that provides access to a sequence of elements, like arrays or generator functions. *Intermediate* operations transform sequences into other sequences, such as filtering elements (e.g. *filter(Predicate<T>)*), transforming elements through a mapping function (*map(Function<T,R>)*), truncating the number of accessible elements (e.g. *limit(int)*), or using one of many other stream producing operations. *terminal* operations are also known as value or side-effect-producing operations that ask for elements to operate, consuming the sequence. These operations include aggregations such as *reduce()* or *collect()*, searching with operations like *findFirst()* or *anyMatch()*, and iteration with *forEach()*.

2.2.9.2 Short-Circuiting

What distinguishes Short-Circuiting operations from Non-Short-Circuiting operations is the fact that these operations, often times, don't process all the elements of the source, instead, when they reach a certain condition, they "short-circuit" the sequence of elements, terminating their processing. Some examples of these operations are:

- *limit(int **n**)*, which is an *intermediate* operation that will only process **n** elements before it short-circuits.
- *allMatch()* , which is a *terminal* operation that will short-circuit as soon as it finds an element that doesn't match the conditions.
- *findFirst()*, which is also a *terminal* operation that will short-circuit on the first element of the sequence.
- etc...

There are many ways to short-circuit a sequence, a few concepts are:

- Using an iterator to traverse a sequence instead of bulk traversal operation such as *forEach()*.
- Throwing a lightweight exception that interrupts the traversal process when the condition is met.

- Passing a boolean while traversing each element to determine if the traversal is done or not.

2.2.9.3 Pipeline: Fluent vs Nested

A Pipeline can be categorized as Fluent or as Nested depending on how it is written.

Fluent pipelines became common in Hybrid Programming languages, in other words, languages take features both from Object-oriented and Functional Programming, such as Java, C#, Scala or Python. In a Fluently written Pipeline, operations are chained to each other sequentially, for instance, looking back at Listing 2.2, we can easily understand that on the source data we will filter all sunny days, **then** map them to their temperature, **then** select the first five, and finally print each one of them, in other words, operations are processed in the same order as they are written.

A Nested Pipeline is another way of writing a pipeline. This way of writing a pipeline nests operations inside each other, making it somewhat counter-intuitive due to the fact that the deepest operation in the nest, is actually the first to run. Listing 2.10 gives an example of what Listing 2.2 would look like if it was written in a Nested form.

```
1 forEach(
2    limit(
3       map(
4          filter(data, Weather::isSunny),
5          Weather::getTempC
6          ),
7       5
8    ),
9    System.out::println
10 );
```

Listing 2.10: Nested Pipeline equivalent to Listing 2.2

2.3 State of the Art

When looking for an alternative to the `Stream` *api* we find that we have a number of options available to us, such as Kotlin's `Sequence`, `StreamEx`, `jOOλ` and `Vavr`. Table 2.1 shows the design features of each these sequences.

Sequence / Feature	Individually Traversal	Bulk Traversal	Internal Access	External Access	Traversals
Java's `Iterable`	✓	✓			Multiple
Java's `Stream`	✓	✓	✓		Once
`Eclipse Collections LazyIterable`	✓			✓	Multiple
Kotlin's `Sequence`	✓			✓	Multiple
`Tinyield's Query`	✓	✓	✓		Once
`StreamEx`	✓	✓	✓		Once
`jOOλ`	✓	✓	✓		Once
`Vavr`	✓			✓	Multiple

Table 2.1: Design choices for sequence alternatives in Java

2.3.1 Extensibility

When it comes to extensiblity each alternative provides a different approach. This section will go through each approach using our Collapse example. As stated previously, Java's approach to custom operations, shown in Listing 1.7, involves creating an auxiliary class that implements `Spliterator` and then making use of `StreamSupport`'s *stream* to wrap our newly created `Spliterator` in a new `Stream` object. This approach incurs in verbosity and loss of pipeline fluency, on the other hand it is available with the JDK. `JOOλ`'s approach is almost identical to Java's, this is due to the fact that `Seq` acts as a wrapper for Java `Streams`. This in turn makes this approach incur in the same drawbacks as the one provided by Java, with the addition of depending on a third-party library. `Eclipse Collections`' way of adding user defined operations, shown in Listing 2.11, requires the definition of a new `Iteratable` that takes the upstream as its source. This approach incurs in verbosity, as we have to implement a new `Iteratable`, as well as loss of pipeline fluency.

```
1  public static <U> LazyIterable<U> collapse(LazyIterable<U> src) {
2     return Lists.immutable.withAll(() -> {
3        Iterator<U> upstream = src.iterator();
4        return new Iterator<U>() {
5           U prev = null;
6           U curr = null;
7
8           @Override
9           public boolean hasNext() {
10             if(curr != null) {
11                return true;
12             }
13             while(upstream.hasNext() && curr == null){
14                U item = upstream.next();
15                if(prev == null || !prev.equals(item)) {
16                   curr = item;
17                }
18             }
19             return curr != null;
20          }
21
22          @Override
23          public U next() {
24             if (!hasNext()) {
25                throw new NoSuchElementException();
26             }
27             prev = curr;
28             curr = null;
29             return prev;
30          }
31       };
32    }).asLazy();
33 }
34
35 // Usage
36 collapse(Lists.immutable.of(1,1,2,3,1,3,3).asLazy())
37    .forEach(System.out::println)
```

Listing 2.11: Implementation of a `collapse` method, using `Eclipse Collections`.

Kotlin's approach, shown in Listing 2.12, could be done by creating an auxiliary class that implements the `Sequence` interface orm as we show in this case, by simply defining a new `Sequence` by implementing the iterator method using the *yield* keyword, accessing the upstream's elements with the instruction `this@collapse.iterator()`. The difference here is that Kotlin allows for extension methods, as we can see in line 1 where we say that calling the *collapse()* method on a `Sequence` will instantiate our newly defined `Sequence`. This approach is verboseless and if used within Kotlin it does not lose the pipeline's

fluency, however it does require a third-party and knowledge of the Kotlin programming language.

```
fun <T> Sequence<T>.collapse() = sequence {
   val iterator = this@collapse.iterator()
   var prev: T? = null
   while (iterator.hasNext()) {
      val aux = iterator.next()
      if (aux != null && aux != prev) {
         prev = aux
         yield(aux)
      }
   }
}
//Usage in Kotlin
sequenceOf(1,1,2,3,1,3,3)
   .collapse()
   .forEach(System.out::println)
//Usage in Java
forEach(collapse(sequenceOf(1,1,2,3,1,3,3)),System.out::println);
```

Listing 2.12: Extension of Kotlin's `Sequence` with collapse operation

Listing 2.13 shows how the `Tinyield` solution makes user defined operations available, as shown previously in Listing 1.12. This method allows the user to provide its own operations, being less verbose than the previous two options since it only requires one method, and without breaking the fluency of the pipeline, however it does require a third-party library.

```
class Collapse {
  private U prev = null;
  <U> Traverser<U> collapse(Query<U> src) {
    return yield -> {
      src.traverse(item -> {
        if (prev == null || !prev.equals(item))
        yield.ret(prev = item);
      });
    };
  }
}
//Usage
Query.of(1,1,2,3,1,3,3)
   .then(this::collapse)
   .traverse(System.out::println)
```

Listing 2.13: Extension of `Tinyield`'s `Query` with collapse operation

`StreamEx`'s approach to user defined operations, shown in Listing 2.14, involves defining new `StreamEx` sequences using the *headTail* method, composing the new head and the new tail recursively. This approach is less verbose than the one provided out of the box by Java's `Stream` as it requires only one method to

add a new operation but it does break the pipeline's fluency, besides requiring a third-party library.

```
1  static StreamEx<Integer> collapse(StreamEx<Integer> input) {
2     return collapse(input, i -> false);
3  }
4
5  static StreamEx<Integer> collapse(StreamEx<Integer> input, Predicate<Integer>
       equalsPrevious) {
6     return input.headTail((head, tail) -> {
7        if (equalsPrevious.test(head)) {
8           return collapse(tail, head::equals);
9        } else {
10          return collapse(tail, head::equals).prepend(head);
11       }
12    });
13 }
14 //Usage
15 collapse(StreamEx.of(1,1,2,3,1,3,3))
16       .forEach(System.out::println);
```

Listing 2.14: Extension of `StreamEx` with collapse operation

`Vavr's` approach to user defined operations, shown in Listing 2.15, is very similar to the one `StreamEx` provides. This is accomplished by using the *cons* method in conjunction with the *head* method and a supplier for the new tail of the sequence recursively. Again, this approach is less verbose than the one provided out of the box by Java but it does break the pipeline's fluency, as well as requiring a third-party library.

```
1  static Stream<Integer> collapse(Stream<Integer> input) {
2     return collapse(input, i -> false);
3  }
4  static Stream<Integer> collapse(Stream<Integer> input, Predicate<Integer> equalsPrev) {
5     Option<Integer> option = input.headOption();
6     if(!option.isEmpty()) {
7        Integer head = option.get();
8        if(equalsPrev.test(head)) {
9           return collapse(input.tail(), head::equals);
10       } else {
11          return Stream.cons(head, () -> collapse(input.tail(), head::equals));
12       }
13    } else {
14       return Stream.empty();
15    }
16 }
17 // Usage
18 collapse( Stream.of(1, 1, 2, 3, 4, 5, 4, 4, 3, 5, 5))
19       .forEach(System.out::println);
```

Listing 2.15: Extension of `Vavr's Stream` with collapse operation

2.3.2 Javascript and Typescript

We also applied the `Tinyield` solution to Typescript and compared it with some of the alternatives available in *npm*, such as `IxJs`, `LazyJs`, `Lodash`, `Sequency`, `Underscore`, and the EcmaScript6 Arrays. Table 2.2 shows the design choices for each of these sequence implementations.

Sequence / Feature	Individually Traversal	Bulk Traversal	Internal Access	External Access	Traversals	Lazy
EcmaScript6 Arrays	✓	✓		✓	Multiple	
`IxJs`	✓			✓	Multiple	✓
`LazyJs`	✓			✓	Multiple	✓
`Lodash`		✓	✓	✓	Multiple	
`Sequency`	✓	✓	✓		Once	✓
`Underscore`		✓	✓	✓	Multiple	
`Tinyield`'s `Query`	✓	✓	✓		Once	✓

Table 2.2: Design choices for sequence alternatives in Javascript and Typescript

2.3.2.1 Extensibility

To add user defined operations, each of these alternatives provides its own approach. We'll go through them one by one, discussing:

- How it affects the developer's ability to chain new operations, in other words, pipeline fluency.
- If it adds operations to the sequence type's prototype, known as prototype pollution.
- If the extension requires a lot of boilerplate code making the approach verbose.
- If by using a custom operation the developer's code editor still recognizes the type of object being used, thus providing the developer with intelligent code completion, or if it loses type inference.

Adding operations to the `Array` *API* in EcmaScript6 is done through prototype extension, as shown in Listing 2.16. This method of prototype extension has been around since the beginning of Javascript and is one of the simplest ways to add operations to an object. This method requires only the definition of a method and adding it to `Array.prototype` through the `Object.defineProperty(...)` method, resulting in a verboseless way of extending the `Array` *API* without compromising pipeline fluency, losing only type inference. The downside of this approach is that it pollutes the prototype of the `Array` class which in turn could

clash with code of other vendors, the alternative would be to call the custom operation directly without adding a new property to the `Array.prototype`, thus losing pipeline fluency.

```
function collapse<T>(this: T[]): T[] {
   let prev: T;
   const result: T[] = [];
   this.forEach(item => {
      if (prev === undefined || prev !== item) {
         prev = item;
         result.push(item);
      }
   });
   return result;
}
Object.defineProperty(Array.prototype, 'collapse', {value: collapse});
// Usage
[1, 2, 3, 3, 3, 4, 5, 5, 4, 5, 4, 4, 3, 2, 1]
   .collapse()
   .forEach(console.log);
```

Listing 2.16: Extension of EcmaScript6 Arrays with collapse operation

`Lazy.js`' way of adding custom operations, shown in Listing 2.17, involves calling the define method over `Sequence` with the operation's name and an object that must have either the property *each* or *getIterator* in which the operation is defined. The property *each* is a function that receives as parameter a function that takes in both an element and the index of that element and returns a `boolean` value indicating whether the traversal has finished or not, this is useful for chaining custom operations with short-circuiting operations. This means that `Lazy.js` does not switch between bulk traversal and individual traversal, even in short-circuiting operations, which let's the user define a custom operation by defining only one property. However, if *getIterator* is called without defining the *getIterator* property the *each* property will be called to buffer all the intermediate results so that they can be iterated afterwards, this means that `Lazy.js` could become eager in this scenario. The property *getIterator* returns an `Iterator` object. This `Iterator` type is a custom type defined in `Lazy.js`, it provides two methods *moveNext()*, which returns true if the iterator is able to move to another element and false otherwise, and *current()* which returns the current element. This approach is verboseless and manages to maintain the pipeline's fluency, although in Typescript type inference is lost.

```
import * as Lazy from 'lazy.js';
(Lazy as any).Sequence.define('collapse', {
   each<T>(consume: (elem: T) => boolean) {
      let prev: any;
      return this.parent.each((item: T) => {
         if (prev === undefined || prev !== item) {
            prev = item;
            return consume(item);
         }
      });
   }
});
// Usage
Lazy([1, 2, 3, 3, 3, 4, 5, 5, 4, 5, 4, 4, 3, 2, 1])
   .collapse()
   .forEach(console.log);
```

Listing 2.17: Extension of `Lazy.js` with collapse operation

Listing 2.18 shows how to add custom operations to `Lodash`'s *API*. This approach is similar to the one `Lazy.js` provides, differing on the call the *mixin* method, instead of *define*, to extend the *API*, passing an object with the added operations as argument. This approach, like `Lazy.js`, is verboseless, maintains pipeline fluency and loses type inference in Typescript. Yet, `Lodash` differs from `Lazy.js` when processing pipeline operations. `Lodash` uses a *while loop* to iterate over its elements and becomes eager once the *value()* method is called, meaning that each operation will run to completion before starting the next one, incurring in the overhead of creating new sequences for each step, as well as processing unnecessary elements in some use-cases, such as using a *first()* operation.

```
import * as _ from 'lodash';
function collapse<T>(upstream: T[]): T[] {
   let prev: T;
   const result: T[] = [];
   upstream.forEach(elem => {
      if (prev === undefined || prev !== elem) {
         prev = elem;
         result.push(elem);
      }
   });
   return result;
}
_.mixin({'collapse': collapse});
// Usage
_.chain([1, 2, 3, 3, 3, 4, 5, 5, 4, 5, 4, 4, 3, 2, 1])
   .collapse()
   .forEach(console.log)
   .value();
```

Listing 2.18: Extension of `Lodash` with collapse operation

Underscore's approach is exactly the same as the one provided by Lodash, differing only in the library that is used, as we can see in Listing 2.19. Therefore, this approach is verboseless, maintains pipeline fluency, loses type inference and incurs in the same overheads Lodash does when processing the pipeline.

```
1  import * as _ from 'underscore';
2  function collapse<T>(upstream: T[]): T[] {
3     let prev: T;
4     const result: T[] = [];
5     upstream.forEach(elem => {
6        if (prev === undefined || prev !== elem) {
7           prev = elem;
8           result.push(elem);
9        }
10    });
11    return result;
12 }
13 _.mixin({'collapse': collapse});
14
15 // Usage
16 _.chain([1, 2, 3, 3, 3, 4, 5, 5, 4, 5, 4, 4, 3, 2, 1])
17    .collapse()
18    .forEach(console.log)
19    .value();
```

Listing 2.19: Extension of Underscore with collapse operation

Listing 2.20 shows how Tinyield allows *API* extensions in Typescript, which is identical to the one provided in Java. To add an operation the user simply needs to define a function that takes in the upstream Query and returns a Traverser using that function in conjunction with the *then* method of the Query class. This approach does not pollute any prototypes, and therefore will not clash with vendor code, is verboseless, maintains pipeline fluency and type inference.

```
1  import {Query, Traverser} from 'tinyield4ts';
2  function collapse<T>(upstream: Query<T>): Traverser<T> {
3     return yld => {
4        let prev: T;
5        upstream.forEach(item => {
6           if (prev === undefined || prev !== item) {
7              prev = item;
8              yld(item);
9           }
10       });
11    };
12 }
13
14 // Usage
15 Query.of([1, 2, 3, 3, 3, 4, 5, 5, 4, 5, 4, 4, 3, 2, 1])
16    .then(collapse)
17    .forEach(console.log);
```

Listing 2.20: Extension of Tinyield's Query with collapse operation

`IxJs` follows the `Iterator` protocol defined by EcmaScript6 which is represented by any object that provides the *next()* method. This method returns an object with two properties, *value* that holds the next element in the sequence, if there are any, and *done* which indicates when we're done traversing all the elements. For this example we used the *yield* keyword, which is used to pause and resume a generator function, in this case the `Iterator`. Once paused on a *yield* expression, the generator's code execution remains paused until the generator's *next()* method is called again. Listing 2.21 shows how to add a custom operation to `IxJs`, this is by far the most verbose alternative out of the ones presented in this section, but it does not lose either type inference nor pipeline fluency. `IxJs`' way of adding custom operations involves creating a new class that extends `IterableX`, redefining the *[Symbol.iterator]* method, in this example using the keyword *yield*, that to help define the `Iterator`, creating a function that instantiates that class with the upstream `IterableX` and finally adding said function to `IterableX`'s prototype. This again, pollutes the prototype of IterableX, which could lead to clashes with other vendor code, although less likely than polluting `Array.prototype`.

```
import {IterableX} from 'ix/iterable';
import {UnaryFunction} from 'ix/interfaces';

export class Collapse<T> extends IterableX<T> {
   private readonly _source: Iterable<T>;

   constructor(source: Iterable<T>) {
      super();
      this._source = source;
   }

   *[Symbol.iterator]() {
      let prev: T;
      const iterator: Iterator<T> = this._source[Symbol.iterator]();
      let item: IteratorResult<T, any>;
      while (!(item = iterator.next()).done) {
         if (prev === undefined || prev !== item.value) {
            prev = item.value;
            yield item.value;
         }
      }
   }
}

export function collapse<T>(): UnaryFunction<Iterable<T>,IterableX<T>> {
   return function collapseOperatorFunction(source: Iterable<T>): IterableX<T> {
      return new Collapse<T>(source);
   };
}

export function collapseProto<T>(this: IterableX<T>): IterableX<T> {
   return collapse<T>()(this);
}

IterableX.prototype.collapse = collapseProto;

declare module 'ix/iterable/iterablex' {
   interface IterableX<T> {
      collapse: typeof collapseProto;
   }
}

// Usage
IterableX.of(1, 2, 3, 3, 3, 4, 5, 5, 4, 5, 4, 4, 3, 2, 1)
   .collapse()
   .forEach(console.log);
```

Listing 2.21: Extension of `IxJs` with collapse operation

`Sequency`'s way of adding user defined operations, shown in Listing 2.22, is similar to the one of `IxJs`. `Sequency` also requires the creation of a new `Sequence` through an `Iterator`, although it does not support definition through generator functions making it impossible to use the *yield* keyword in this scenario. To add a

```
1  import Sequence, {createSequence, extendSequence, sequenceOf} from 'sequency';
2
3  class Collapse<T> {
4     collapse(this: Sequence<T>): Sequence<T> {
5        const iterator = this.asIterable()[Symbol.iterator]();
6        let prev: T;
7        return createSequence({
8           next(): IteratorResult<T, any> {
9              let next = iterator.next();
10             while(!next.done) {
11                if (prev === undefined || prev !== next.value) {
12                   prev = next.value;
13                   return next;
14                }
15                next = iterator.next();
16             }
17             return next;
18          }
19       });
20    }
21 }
22
23 declare module 'sequency' {
24    export default interface Sequence<T> extends Collapse<T> {
25    }
26 }
27
28 // Usage
29 extendSequence(Collapse);
30 sequenceOf(1, 2, 3, 3, 3, 4, 5, 5, 4, 5, 4, 4, 3, 2, 1)
31    .collapse()
32    .forEach(console.log);
```

Listing 2.22: Extension of `Sequency` with collapse operation

new operation the user needs to define a class with the new operation, and if type inference is required, the `Sequence` interface must be re-declared extending the new class as seen in lines 23 to 26. This approach is verbose and may lose type inference depending on how the user decides to implement it, yet it does not lose pipeline fluency.

2.4 Java `Stream` Internals

Java `Streams` are masterpiece of software engineering, but diving through their internals is quite complex, so in order to make it clearer, we'll look at how it works.

As we've previously discussed, a sequence's pipeline is made of three parts: a source, one or more intermediate operations and one terminal operation. Java's implementation of a sequence (`Stream`) represents each of these parts through a different interface, namely `Spliterator` as the source, `Sink` as the intermediate operations and `TerminalSink` as the terminal operation.

2.4.1 Spliterator

Spliterators, like Iterators, are for traversing the elements of a source, so a question that may arise is "Why the need for the Spliterator interface?". The Spliterator *API* was designed to support efficient parallel traversal in addition to sequential traversal, by supporting decomposition as well as single-element iteration. In addition, the protocol for accessing elements via a Spliterator is designed to impose smaller per-element *overhead* than Iterator, and to avoid the inherent race involved in having separate methods for *hasNext()* and *next()*. Thus, Spliterator provides, among other things, three core methods:

- *boolean tryAdvance(Consumer<T> action)* - This method performs the given action on the next element of the sequence, if there are any, and returns whether an element was processed or not. This method essentially combines the methods *hasNext()* and *next()* of the Iterator interface with the addition of performing the desired action on the next element.
- *void forEachRemaining(Consumer<T> action)* - This method performs the given action on every remaining element of the sequence. It's a way to bulk traverse the entire sequence while performing an action on each element.
- *Spliterator<T> trySplit()* - Out of these three methods this is the one that makes Spliterator the most distinct from Iterator. This method, as the name implies, tries to split the current Spliterator in two if possible, ideally partitioning it in the middle, with the purpose of enabling parallel processing of the sequence.

2.4.2 Sink

The Sink interface extends from Consumer. It's Java's way of representing an intermediate operation in the pipeline, which they call "stage". The *accept()* method declared by Consumer is where the Sink implements whatever processing it performs on each element, Listing 2.23 shows what the *accept()* method would look like for the *map()* operation.

```
1 public void accept(T t) {
2    downstream.accept(mapper.apply(t));
3 }
```

Listing 2.23: *accept()* method for the *map()* operation

For simple operations like *filter* and *map* the *accept()* method is enough, but other operations require state or even to have a way of halting the traversal of elements. `Sink` provides three more methods that come in handy in a these situations:

- *void begin(long size)* - This is a setup method, it's used to initialize state as well as let a downstream `Sink` know how many elements will be processed. Listing 2.24 showcases the use of this operation in the *distinct* operation, in which the method initializes a new `HashSet` to check if an element is distinct or not and tells the downstream `Sink` that it doesn't know how many elements it will have to process, due to the fact that the *distinct* operation has the potential to alter the size of the sequence.

- *void end()* - This is a tear down method, it's used to clean-up state and let the downstream `Sink` know that the processing is over. Listing 2.24 showcases the use of this operation in the *distinct* operation, in which the method cleans up the *seen* field and tells the downstream `Sink` that the processing has ended.

- *boolean cancellationRequested()* - This method Indicates that the `Sink` does not wish to receive any more data. Listing 2.25 showcases the use of this method in the *takeWhile* operation, in which this method states that the `Sink` does not wish to receive any more elements after the predicate has tested false.

```
Stream<T> distinct() {
  return new ChainedRef<T, T>(this) {
    Set<T> seen;

    public void begin(long size) {
      seen = new HashSet();
      this.downstream.begin(-1L);
    }

    public void end() {
      seen = null;
      this.downstream.end();
    }

    public void accept(T t) {
      if (this.seen.add(t)) {
        this.downstream.accept(t);
      }

    }
  };
}
```

Listing 2.24: Example implementation of the *distinct* operation

```
Stream<T> takeWhile(Predicate<T> predicate) {
  return new ChainedRef<T, T>(this) {
    boolean take = true;

    public void begin(long size) {
      this.downstream.begin(-1L);
    }

    public void accept(T t) {
      if (take && (take = predicate.test(t))) {
        this.downstream.accept(t);
      }

    }

    public boolean cancellationRequested() {
      return !take ||
        this.downstream.cancellationRequested();
    }
  };
}
```

Listing 2.25: Example implementation of the *takeWhile* operation

2.4.3 `TerminalSink` and `TerminalOp`

As previously stated `TerminalSink` is the interface responsible for representing a terminal operation in Java, this interface extends `Sink` and `Supplier` because `TerminalSinks` may need to accept values in operations such as *forEach()* as well as produce values in operations such as *max()* . Listing 2.26 showcases what a `TerminalSink` implementation for the *find* operation could look like.

```
1  class FindSink<T, O> implements TerminalSink<T, O> {
2     boolean hasValue;
3     T value;
4
5     public void accept(T value) {
6        if (!hasValue) {
7           hasValue = true;
8           this.value = value;
9        }
10    }
11
12    public boolean cancellationRequested() {
13       return hasValue;
14    }
15
16    public Optional<T> get() {
17       return hasValue ? Optional.of(value) : null;
18    }
19 }
```

Listing 2.26: Find operation `TerminalSink` example implementation

Java also provides another interface, `TerminalOp`, that is responsible for defining how to evaluate the results of a pipeline, either sequentially or in parallel, it does this through the methods *evaluateSequential()* and *evaluateParallel()*, respectively. Listing 2.27 showcases what a `TerminalOp` implementation for the *find* operation could look like. `TerminalOp`'s *evaluateSequential()* and *evaluateParallel()* make use of the `PipelineHelper`, which is a helper class that holds all of the information about a stream pipeline, such as output shape, intermediate operations, stream flags, parallelism, etc. . These methods use the `PipelineHelper`'s method *wrapAndCopyInto()* to apply the stages described by the helper to the provided `Spliterator` and then pass the results to the *Sink* provided by calling *sinkSupplier.get()*, *evaluateSequential()* uses it directly while *evaluateParallel()* calls this method for each leaf obtained from the `Spliterator` through the `FindTask`. As for which method is called, *evaluateSequential()* or *evaluateParallel()*, it is decided when the method *findFirst()* is called on the `ReferencePipeline` abstract class which gets the appropriate `FindOp` and passes it on to the *evaluate* method of the `AbstractPipeline` abstract class, that then decides to call *evaluateParallel()* if its source stage is parallel or *evaluateSequential()* otherwise.

```
1  class FindOp<T, O> implements TerminalOp<T, O> {
2     private final StreamShape shape;
3     final int opFlags;
4     final O emptyValue;
5     final Predicate<O> presentPredicate;
6     final Supplier<TerminalSink<T, O>> sinkSupplier;
7
8     FindOp(boolean mustFindFirst, StreamShape shape, O emptyValue,
9           Predicate<O> presentPredicate, Supplier<TerminalSink<T, O>> sinkSupplier) {
10     this.opFlags = ...;
11     this.shape = shape;
12     this.emptyValue = emptyValue;
13     this.presentPredicate = presentPredicate;
14     this.sinkSupplier = sinkSupplier;
15    }
16    (...)
17    @Override
18    public <S> O evaluateSequential(PipelineHelper<T> helper,
19                Spliterator<S> spliterator) {
20     O result = helper.wrapAndCopyInto(sinkSupplier.get(), spliterator).get();
21     return result != null ? result : emptyValue;
22    }
23
24    @Override
25    public <P_IN> O evaluateParallel(PipelineHelper<T> helper,
26                Spliterator<P_IN> spliterator) {
27     boolean mustFindFirst = StreamOpFlag.ORDERED.isKnown(helper.getStreamAndOpFlags());
28     return new FindTask<>(this, mustFindFirst, helper, spliterator).invoke();
29    }
30 }
```

Listing 2.27: Find operation `TerminalOp` example implementation

Another topic worth mentioning is the fact that some arguments are passed to `FindOp` through the constructor while others are passed directly to *evaluateSequential()* and *evaluateParallel()*, this distinction splits the arguments that vary depending on the pipeline that the `TerminalOp` is used in from those that don't. In other words, the constructor arguments are the same regardless of the pipeline where `FindOp` is included, they depend only on the operation called, Listing 2.28 shows the arguments for *findFirst()* and for *findAny()*.In oposition both the *PipelineHelper* and the *Spliterator* vary, seeing as one is the description of this pipeline's stages and the other is the data source.

```
1 static final TerminalOp<?, ?> OP_FIND_FIRST = new FindOp<>(true,
2       StreamShape.REFERENCE, Optional.empty(),
3       Optional::isPresent, FindSink.OfRef::new);
4
5 static final TerminalOp<?, ?> OP_FIND_ANY = new FindOp<>(false,
6       StreamShape.REFERENCE, Optional.empty(),
7       Optional::isPresent, FindSink.OfRef::new);
```

Listing 2.28: Find operation constants for *findFirst()* and findAny()

2.4.4 ReferencePipeline

`ReferencePipeline` is the class that implements Java's `Stream`. This is where all the previous interfaces come together. `ReferencePipeline` implements each operation either directly or indirectly through another class that provides the use-case, such as `DistinctOps` or `ReduceOps`. This *class* chains the pipeline together and when a terminal operation is called evaluates the result. To do this, it *wraps* the chain of `Sinks` present in the *pipeline* with the previous `Sink`, starting at the end and stopping when it reaches the source, after that it goes through the elements of the source, passing each one to the newly *wrapped* `Sink`. Take Listing 2.29 as an example pipeline to illustrate this point and Listings 2.30, 2.31, 2.32 and 2.33 as the rough translation of each line of code corresponding to the pipeline.

These Listings, although complex, are a simplified version of what actually happens, for instance Java's `Stream` doesn't often go through the source's `Spliterator` like we illustrate in Listings 2.33 line 6, but instead uses a `WrappingSpliterator` that buffers elements before consuming them, this is useful in operations such as *flatMap* in which a single iteration of the source can produce many elements at once. Another example is the fact that *filter*, in Listing 2.31, actually returns a `StatelessOp` which is a class that extends `ReferencePipeline`, but in order to make it easier to understand these details were omitted from the listings.

```
1 String[] strings = new String[]{
2   "505",
3   "Amsterdam",
4   "Mural"
5 };
6 Arrays.stream(strings)
7   .filter(s -> !s.startsWith("A"))
8   .mapToInt(String::length)
9   .max()
```

Listing 2.29: Example pipeline

```
1 sourceSpliterator = new ArraySpliterator(strings);
```

Listing 2.30: `ReferencePipeline` source - Equivalent to line 6 of Listing 2.29

```
1 upstream = source;
2 Sink<String> opWrapSink(int flags, Sink<String> downstream) {
3     return new Sink.ChainedReference<String, String>(downstream) {
4         ...
5
6         public void accept(String u) {
7             if (!u.startsWith("A"))
8                 downstream.accept(u);
9         }
10    };
11 }
```

Listing 2.31: `ReferencePipeline` filter - Equivalent to line 7 of Listing 2.29

```
1 upstream = filter
2 Sink<String> opWrapSink(int flags, Sink<Integer> downstream) {
3     return new Sink.ChainedReference<String, Integer>(downstream) {
4         ...
5
6         public void accept(String u) {
7             downstream.accept(u.length());
8         }
9     };
10 }
```

Listing 2.32: `IntPipeline` mapToInt - Equivalent to line 8 of Listing 2.29

```
1 terminalSink = ReduceOps.makeInt(Math::max);
2 pipeline = filter.opWrapSink(mapToInt.opWrapSink(terminalSink));
3
4 // evaluate
5 pipeline.begin(sourceSpliterator.getExactSizeIfKnown());
6 sourceSpliterator.forEachRemaining(pipeline);
7 pipeline.end();
8 return pipeline.get();
```

Listing 2.33: *int max* - Equivalent to line 9 of Listing 2.29

3

Minimalist Sequence traversal design

In this chapter we will present the `Tinyield` solution, going through its different components and understanding how they interact with each other, as well as explaining design choices.

3.1 Overview

The goal behind `Tinyield` was to provide a *lazy* sequence type that would allow fluent extensibility and without compromising either performance or readability for most use-cases. The `Query` class was created for this purpose, it is the sequence type provided by the `Tinyield` solution. A `Query` instance can be traversed either in *bulk* (defined by the `Traverser` interface) or element by element (defined by the `Advancer` interface) providing internal iteration in both cases. These two alternatives provide two ways of effectively traverse only a few elements of the sequence, as well as a way to traverse all elements in a more efficient way.

3.2 Yield

In order to provide yield like semantics the `Yield` functional interface was created (Listing 3.1).

```
1 @FunctionalInterface
2 public interface Yield<T> {
3    void ret(T item);
4 }
```

Listing 3.1: `Yield` interface

`Yield` is essentially a `Consumer` where the *accept* method was renamed to *ret*, short for return which is a keyword in many programming languages. This interface is used mostly when specifying how to traverse a sequence, as seen in Listing 3.2, but can also be used in an intermediate operation (mentioned in section 2.2.9.1) to specify which elements said operation should yield and how. In Listing 3.3 when calling the *then()* method `Yield` is used to define a custom operation equivalent to a *distinctBy()*, calling *ret()* only when a unique number of digits is found, in other words, yielding unique number of digits.

```
1 Yield<Integer> yld = out::println;
2 Query.of(1, 2, 3).traverse(yld);
```

Listing 3.2: `Yield` used in the *traverse()* method

```
1  HashSet<Integer> lengths = new HashSet<>();
2  Query
3    .generate(() -> random()* MAX_VALUE)
4    .limit(1024)
5    .map(Double::intValue)
6    .then(src -> yield -> src.traverse(item -> {
7      int nrOfDigits = item.toString().length();
8      if(lengths.add(nrOfDigits))
9        yield.ret(item);
10   }))
11   .forEach(out::println);
```

Listing 3.3: `Yield` used in intermediate operation

3.3 Traverser

The `Traverser` interface (Listing 3.4) was designed with bulk traversal in mind, thus providing a single method *traverse*, which can be equated with Java `Streams` *forEach* method.

```
1 @FunctionalInterface
2 public interface Traverser<T> {
3    void traverse(Yield<? super T> yield);
4 }
```

Listing 3.4: `Traverser` interface

Assuming the intent is to traverse the whole sequence, this would be the most efficient path to do so, because *traverse* receives an instance of `Yield` and the only thing it needs to do is push elements into that instance for as long as elements are available. In opposition, having to iterate each element manually through an iterator has a bigger overhead due to more method calls and potentially more context switching. In the previous Listing 3.3 we're actually declaring a *lambda* `Function` that transforms a "*src*" `Query` into a `Traverser` by programming what should be done in the *traverse* method.

3.4 Advancer

The `Advancer` interface (Listing 3.5) was designed to allow individual traversal of elements.

```
1 public interface Advancer<T> {
2    boolean tryAdvance(Yield<? super T> yield);
3 }
```

Listing 3.5: `Advancer` interface

The `Advancer` interface allows for individual traversal with a single method, *tryAdvance()* which follows the same semantics as the *tryAdvance()* provided by `Spliterator`.

3.5 Operations

Intermediate operations combine both `Traverser` and `Advancer` to allow for both types of traversal. In other words, each intermediate operation supported by `Query` is implemented through a class implementing both `Traverser` and

`Advancer`, specifying how to traverse a sequence in bulk as well as how to traverse it element by element.

In Listing 3.6 we showcase how the *map* operation is implemented following these protocols.

```
1  public class Mapping<T, R> implements Advancer<R>, Traverser<R> {
2
3     private final Query<T> upstream;
4     private final Function<? super T, ? extends R> mapper;
5
6     public Mapping(Query<T> adv, Function<? super T, ? extends R> mapper) {
7        this.upstream = adv;
8        this.mapper = mapper;
9     }
10
11    @Override
12    public void traverse(Yield<? super R> yield) {
13       upstream.traverse(e -> yield.ret(mapper.apply(e)));
14    }
15
16    @Override
17    public boolean tryAdvance(Yield<? super R> yield) {
18       return upstream.tryAdvance(item -> yield.ret(mapper.apply(item)));
19    }
20 }
```

Listing 3.6: Implementation of map operation

While traversing a sequence in *bulk* is more efficient when compared to traversing individually, the latter is still needed for specific use-cases. These are use-cases that require the sequence to maintain its *lazy* nature while iterating through the elements, such as the *zip* operation. *Zip* is an operation that takes a tuple of sequences and transforms them into a sequence of tuples. Now, if we could only *bulk* traverse a sequence, this would mean *bulk* traversing all the sequences, buffering them into data structures and then combine them into a new sequence. Not only that, but if one of the sequences were to be infinite this would result in infinite processing, until the process crashed. This is where individual traversal is needed, by traversing each element of each sequence individually we can request any element only when it's truly needed and use it to create the next tuple, without needing to buffer elements. Listing 3.7 shows how the *zip* operation is implemented.

```
1  public class Zip<T, U, R> implements Advancer<R>, Traverser<R> {
2     private final Query<T> upstream;
3     private final Query<U> other;
4     private final BiFunction<? super T, ? super U, ? extends R> zipper;
5
6     public Zip(Query<T> upstream, Query<U> other, BiFunction<? super T, ? super U, ?
          extends R> zipper) {
7        this.upstream = upstream;
8        this.other = other;
9        this.zipper = zipper;
10    }
11
12    @Override
13    public boolean tryAdvance(Yield<? super R> yield) {
14       BoolBox consumed = new BoolBox();
15       upstream.tryAdvance(e1 -> other.tryAdvance(e2 -> {
16          yield.ret(zipper.apply(e1, e2));
17          consumed.set();
18       }));
19       return consumed.isTrue();
20    }
21
22
23    @Override
24    public void traverse(Yield<? super R> yield) {
25       upstream.shortCircuit(e1 -> {
26          if(!other.tryAdvance(e2 -> yield.ret(zipper.apply(e1, e2))))
27             Yield.bye();
28       });
29    }
30 }
```

Listing 3.7: `Advancer` implementation for zip operation

`Advancers` could also be used for *short-circuiting* operations, but, as we'll explain in section 3.6.2, we decided to not use only `Advancers` for that purpose.

3.6 Query

The `Query` class is where it all comes together, it is the **sequence** type of the `Tinyield` solution. Right out of the box it provides all the operations that Java's `Stream` provides and more. The concept behind this solution is quite simple, a `Query` is, like any sequence that follows the Collection Pipeline pattern[4], composed by a source, one or more intermediate operations and a terminal operation. This by itself does not differ from any other solution out there, what makes `Tinyield` different is that it provides a simple way to extend its *api*, fluently,

as we've already seen in Listing 3.3, while also giving primacy to the fast-path iteration protocol which is defined in the `Traverser` interface, making it more efficient in most day to day use-cases when compared to Java's `Stream`. With that being said this posed some challenges and in some cases it has some pitfalls, as we'll showcase in the next few sections.

```
IntQuery top5Temps = pastWeather
   .filter(Weather::isSunny)
   .mapToInt(Weather::getTempC)
   .limit(5)
```

Listing 3.8: `Tinyield` equivalent to Listing 1.1

3.6.1 Extensibility

`Querys` *API* will satisfy most users out of the box with a collection of method's that will cater to most use-cases, but every now and then a user might find himself in the need of a new operation that is not provided out of the box by the `Tinyield` solution. To address this challenge `Query` *API* provides a *then()* method that allows users to fluently add custom operations when needed. To do so the programmer defines a `Traverser` from the upstream `Query` pipeline, as we have seen both in Listing 1.12 as well as Listing 3.3. Yet, to further exemplify let's imagine that a developer would like to use a *distinctBy()* operation, that was not provided out of the box by the `Tinyield` solution, the user could define it himself as we show in Listing 3.9 and then use it as shown in Listing 3.10, or if re-usability is not a concern use an in-place extension as shown in Listing 3.3.

```
class Traversers {
   public static <T, R> Function<Query<T>, Traverser<T>> distinctBy(Function<T, R>
       selector) {
      return src -> yield -> {
         HashSet<R> selected = new HashSet<>();
         src.traverse(item -> {
            int nrOfDigits = item.toString().length();
            if (selected.add(selector.apply(item)))
               yield.ret(item);
         });
      };
   }
}
```

Listing 3.9: Definition of *distinctBy* operation

```
1 Query
2    .generate(() -> random() * MAX_VALUE)
3    .limit(1024)
4    .map(Double::intValue)
5    .then(Traversers.distinctBy(nr -> nr.toString().length()))
6    .forEach(out::println);
```

Listing 3.10: Use of pre-defined `Querys` *api* extension

By defining the fast-path iteration through the `Traverser` we get a more efficient bulk traversal of the sequence as well as maintaining the fluency of the pipeline, but what if we want to iterate each element? That's one of the shortcomings of this solution. By extending the *api* this way we have no clean way to infer the element by element iteration. There are a number of solutions to this problem, such as dynamically generating an implementation for the `Advancer` when the `Traverser` is provided, or providing only the individual traversal definition and infer the bulk traversal, but that would be less performant. We decided to solve this problem by creating an overload of the *then()* method, with the added capability for the developer to define how to traverse a sequence both in bulk as well as individually.

3.6.2 Short-Circuiting

Another challenge posed by our implementation of *extensibility* is how to *short-circuit* the traversal of a sequence, since our extension method defines the fast-path iteration protocol, it's not possible to interrupt the iteration midway in the normal flux of an application unless the developer also provides and `Advancer` definition. As we mentioned previously in section 2.2.9.2, there are a few ways we could *short-circuit* the traversal of a sequence, but given our solution for *extensibility* we decided the best approach was to throw a light-weight exception to interrupt the traversal process. For this purpose we created a custom `Error` class, `TraversableFinishError`, which also follows the Singleton pattern, and provide the same `Error` instance every time we want to interrupt the flux of execution. This was achieved by adding an auxiliary method to the `Yield` interface, called *bye* (Listing 3.11), to be used by any *short-circuiting* operation, and added a method to handle *short-circuiting* operations in our sequence class `Query` (Listing 3.12).

```java
@FunctionalInterface
public interface Yield<T> {
   /**
    * Auxiliary function for
        traversal short circuit.
    */
   static void bye() {
      throw TraversableFinishError.
          finishTraversal;
   }

   void ret(T item);

}
```

Listing 3.11: `Yield` interface with *bye* method

```java
public class Query<T> {
// ...
   public final void shortCircuit(Yield<T> yield)
        {
      try{
         this.adv.traverse(yield);
      }catch(TraversableFinishError e){
         /* Proceed */
      }
   }
// ...
}
```

Listing 3.12: *shortCircuit* method provided by `Query`

To better illustrate, we can think of the method *takeWhile*, which takes elements while a `Predicate` holds true. Our implementation of *takeWhile* consists of creating an operation class that makes use of both these new methods when the fast-path iteration protocol is selected, as we can see in Listing 3.13. When the `Predicate` returns false for the first time, we call *Yield.bye()*, all the while traversing the upstream `Query` using the *shortCircuit* method that handles the `TraversableFinishError` thrown by *Yield.bye()*.

```
1  public class TakeWhile<T> implements Advancer<T>, Traverser<T> {
2     private final Query<T> upstream;
3     private final Predicate<? super T> predicate;
4     private boolean hasNext;
5
6     public TakeWhile(Query<T> upstream, Predicate<? super T> predicate) {
7        this.upstream = upstream;
8        this.predicate = predicate;
9        this.hasNext = true;
10    }
11
12    @Override
13    public boolean tryAdvance(Yield<? super T> yield) {
14       if(!hasNext) return false; // Once predicate is false it finishes the iteration
15       Yield<T> takeWhile = item -> {
16          if(predicate.test(item)){
17             yield.ret(item);
18          } else {
19             hasNext = false;
20          }
21       };
22       return upstream.tryAdvance(takeWhile) && hasNext;
23    }
24
25    @Override
26    public void traverse(Yield<? super T> yield) {
27       upstream.shortCircuit(item -> {
28          if(!predicate.test(item)) Yield.bye();
29          yield.ret(item);
30       });
31    }
32 }
```

Listing 3.13: Definition of the *limit* operation

3.7 AsyncQuery

Should the programmer need to run code asynchronously, for instance when running IO operations with a non-blocking *API*, the `Tinyield` solution provides the `AsyncQuery` and `AsyncTraverser`.

The iteration protocol is represented by the `AsyncTraverser` interface, shown in Listing 3.14, that provides a single method, *subscribe*. This method allows the programmer to register a `BiConsumer` to be invoked with each element emitted by the sequence and each error, as well as, retrieving a `CompletableFuture` that allows the programmer to cancel the subscription when necessary.

```
public interface AsyncTraverser<T> {
   CompletableFuture<Void> subscribe(BiConsumer<? super T,? super Throwable> cons);
}
```

Listing 3.14: Definition of the `AsyncTraverser` interface

As for the `AsyncQuery` class, it is an abstract class that provides the operations that can be performed on the sequence. This class let's the developer transform the sequence the way he intends even with asynchronous operations.

To exemplify the use of `AsyncQuery`, let's say the programmer would like to query a weather *API* for the weather of an upcoming day of the week. Listing 3.15 shows how to do this, by making an *http* request through the *URL* class, blocking the current thread until a response is attained.

```
static readonly API_URL_TEMPLATE = ...;

static WeatherForecast getWeatherForNext(String dayOfWeek) {
   try {
      InputStream resp = new URL(format(API_URL_TEMPLATE, dayOfWeek)).openStream();
      String body = new BufferedReader(new InputStreamReader(resp))
         .lines()
         .collect(joining("\n"));
      return WeatherForecast.from(body)
   } catch (IOException e) {
      e.printStackTrace();
      return null;
   }
}
```

Listing 3.15: Blocking *API* query through `URL` class

Listing 3.16 illustrates how this code would be used with a `Query`. Notice how *"Done!"* is the last outputted *string*. This happens because each element on the `Query` will block the current thread when calling *getWeatherForNext()*. In other words, the current thread will be blocked 5 times while executing this pipeline, notice also how this execution only outputs 3 days of forecasts, which means the current thread has been waiting for data that it doesn't actually need to use.

```
String[] daysOfWeek = new String[]{"Sat", "Sun", "Mon", "Tue", "Wed", "Thu", "Fri"};
Query
    .of(daysOfWeek)
    .skip(2)
    .map(n -> getWeatherForNext(n))
    .filter(Objects::nonNull)
    .filter(weather -> weather.tempC() > 15)
    .forEach(out::println);
out.println("Done!");

// Output:
// The weather for next Monday is Sunny with 21 degrees Celsius
// The weather for next Tuesday is Sunny with 19 degrees Celsius
// The weather for next Wednesday is Cloudy with 16 degrees Celsius
// Done!
```

Listing 3.16: Using blocking operation with `Query`

Now let's say the developer doesn't want to block the current thread, especially if some of the data will be discarded. One possible implementation would be to use `HttpClients` non-blocking *API*, as shown in Listing 3.17.

```
static readonly API_URL_TEMPLATE = ...;
static HttpClient httpClient = HttpClient.newHttpClient();
static HttpRequest.Builder requestBuilder = HttpRequest.newBuilder();
static HttpRequest request(String url) {
  return requestBuilder.uri(URI.create(url)).build();
}

static AsyncQuery<WeatherForecast> getWeatherForNext(String dayOfWeek) {
  return AsyncQuery.of(
    httpClient
      .sendAsync(request(format(API_URL_TEMPLATE, dayOfWeek)), BodyHandlers.ofString())
      .thenApply(HttpResponse::body)
      .thenApply(WeatherForecast::from)
      .exceptionally(throwable -> {
        throwable.printStackTrace();
        return null;
      })
  );
}
```

Listing 3.17: Non blocking *API* query through `HttpClient`, using `CompletableFuture` and `AsyncQuery`.

Listing 3.18 presents how to use this new implementation of *getWeatherForNext()* with `AsyncQuery`. Notice how in this case, the "Subscribed!" *string* is the first to

be printed. This illustrates that the currently running thread is not blocked when the *subscribe* method is called. Instead, the current thread proceeds with the code it needs to run, while the *http* requests and the handling of the results are done asynchronously, every time a response is received and a *thread* is available to handle it.

```
String[] daysOfWeek = new String[]{"Sat", "Sun", "Mon", "Tue", "Wed", "Thu", "Fri"};
AsyncQuery
   .of(daysOfWeek)
   .skip(2)
   .flatMapConcat(n -> getWeatherForNext(n))
   .filter(Objects::nonNull)
   .filter(weather -> weather.tempC() > 15)
   .subscribe((weather,err) -> {
         if(weather != null)
            out.println(weather);
         if(err != null)
            err.println(err);
      });
out.println("Subscribed!");

// Output:
// Subscribed!
// The weather for next Monday is Sunny with 21 degrees Celsius
// The weather for next Tuesday is Sunny with 19 degrees Celsius
// The weather for next Wednesday is Cloudy with 16 degrees Celsius
```

Listing 3.18: Example on using AsyncQuery

4

Benchmarks and Performance Analysis

To better understand how the `Tinyield` solution fared against state of the art sequences libraries we decided to devise a set of benchmarks that would check how each solution performed for different operations or different pipelines, as well as make them publicly available for the community. For that end we added all the code to Github repositories and devised Github actions to automatically run the benchmarks. The repositories are:

- github.com/tinyield/sequences-benchmarks - This project compares `Tinyield` with some alternatives in Java, including Java's `Stream`.
- github.com/tinyield/ts-sequences-benchmarks - This project compares `Tinyield` with some alternatives in Javascript and Typescript.
- github.com/tinyield/parallel-sequences-benchmarks - This project compares sequential performance to the performance gains attained by using `Stream` *parallel()* to determine at which point Java `Streams` parallelization becomes advantageous.

This chapter aims to explain each benchmark's pipeline and what types of data were used to run it with, showcase the results obtained for each run and analyse the results. In section 4.1 the various approaches are compared according to a set

of features. After that, section 4.2 provides details on how the approaches were benchmarked, what environments were used and what parameters. Sections 4.3, 4.4 and 4.5 each describe a set of benchmarks and present their respective results, discussing how the approaches fared when compared to each other while also explaining the results. Section 4.3 describes the pipelines benchmarked with generated data, section 4.4 discusses pipelines with real-world use-cases and section 4.5 describes pipelines that compare Java's `Stream` sequential processing versus it's *parallel()* feature, this section also makes use of pipelines described in section 4.3. Finally, section 4.6 presents all the results in a table format and draws conclusions from them.

4.1 Feature Comparison

For Java our alternatives come in two types, either a implementation of a Sequence type, which consist of Java's `Stream`, Kotlin's `Sequence`, `Eclipse Collections` `LazyIterable`, `Tinyield's` `Query`, `StreamEx`, `jOOλ's` `Seq` and `Vavr's` `Stream`, or a utility library that adds a few methods to use with the current Java's `Stream` implementation, such as `Guava` and `Protonpack`.

Table 4.1 gives us a brief comparison between our alternatives on three of the points described below:

- Verbosity - Is adding a custom operation to a pipeline verbose?
- Fluency - Does adding a custom operation to the pipeline support method chaining?
- `Stream` compatibility - Can the pipeline be used in conjunction with Java's `Stream`?

From table 4.1 we can see that we have four verboseless ways of expanding the Sequence API which are `Tinyield's` `Query`, `StreamEx` and `Vavr` and Kotlins `Sequence`, yet only `Tinyield's` `Query` and Kotlins `Sequence` maintain fluency out of these four.

For Typescript we only analyzed Sequence types, namely `Underscore`, `Tinyield's` `Query`, `Sequency`, `Lodash`, `ES6 Arrays`, `Lazy.js` and `IxJs`.

Table 4.2 gives us a brief comparison between our alternatives on the points described below:

[1] Although `Seq` is compatible with `Stream` it does not support parallel processing.

	Verboseless	Fluent	`Stream` compatibility
Java's `Stream` Ext.	X	X	N.A.
Kotlin's `Sequence`	✓	✓	*asStream()*
`Eclipse Collections LazyIterable`	X	X	*StreamSupport.stream()*
`Tinyield Query`	✓	✓	*toStream()*
`StreamEx`	✓	X	✓
`jOOλ's`	X	X	✓[1]
`Vavr`	✓	X	*toJavaStream()*
`Guavas` interpolation with `Stream`	X	X	X
`Protonpacks` interpolation with `Stream`	X	X	X

Table 4.1: Feature Comparison

- Verbosity - Is adding a custom operation to a pipeline verbose?
- Fluency - Does adding a custom operation to the pipeline support method chaining?
- Type Inference - When a custom operation is used, is type inference preserved?
- Prototype Pollution - Does adding a custom operation pollute vendor prototypes.

	Verboseless	Fluent	Type Inference	Prototype Pollution
`Underscore`	✓	✓	X	X
`Tinyield's Query`	✓	✓	✓	X
`Sequency`	X	✓	✓[2]	✓[3]
`Lodash`	✓	✓	X	X
`ES6 Arrays`	✓	✓[4]	✓[2]	✓[3]
`Lazy.js`	✓	✓	X	X
`IxJs`	X	✓[4]	✓[2]	✓[3]

Table 4.2: Feature Comparison

From table 4.2 we can observe that while all options support some kind of fluency, some of them have drawbacks, such as `IxJs` and `ES6 Arrays`, that require Prototype Pollution in order to maintain fluency. With regards to type inference `Underscore`, `Lodash` and `Lazy.js` don't support it, `ES6 Arrays`, `IxJs` and `Sequency` support it but to do so would also need to pollute the Prototype, not only that but the latter two also incur in verbosity. `Tinyield's Query` is the

[2]Can maintain type inference if Prototype is Polluted
[3]Pollutes Prototype if type inference is required
[4]Can maintain Fluency if Prototype is Polluted

only alternative, out of the ones evaluated in this document, that maintains fluency and type inference while not polluting any Prototypes and not incurring in verbosity.

4.2 Environment and Parameters

To avoid IO operations during benchmark execution we have previously collected all data into resource files and we load all that data into in-memory data structures on benchmark bootstrap. Thus the sources are performed from memory and avoid any IO.

For Java we ran the benchmarks using jmh[14], both in a local machine as well as in GitHub, with GitHub actions, which presented consistent behavior and results with the performance tests collected locally. For Typescript we ran the benchmarks using the library Benchmark.js[3], both in GitHub as well as our machine.

Our local machine presents the specs listed in Listing 4.1.

```
1 Microsoft Windows 10 Home
2 10.0.18363 N/A Build 18363
3 Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, 2801 Mhz, 4 Core(s), 8 Logical Processor(s)
4 openjdk 15.0.1 2020-10-20
5 OpenJDK Runtime Environment (build 15.0.1+9-18)
6 OpenJDK 64-Bit Server VM (build 15.0.1+9-18, mixed mode, sharing)
```

Listing 4.1: Local Machine specs

The benchmarks were run both on our local machine and on GitHub, with the following options passed to jmh[14]:
-i 4 -wi 4 -f 1 -r 2 -w 2 -tu s –jvmArgs "-Xms6G -Xmx6G -XX:ActiveProcessorCount=X"

These options correspond to the following configuration:

- -i 4 - run 4 measurement iterations.
- -wi 4 - run 4 warm up iterations.
- -f 1 - fork each benchmark once only.
- -r 2 - spend at least 2 seconds at each measurement iteration.
- -w 2 - spend at least 2 seconds at each warm up iteration.

- -tu s - use seconds as the time unit.

- –jvmArgs "-Xms6G -Xmx6G -XX:ActiveProcessorCount=*X*" - set the initial and the maximum Java heap size to 6 Gb and set the number of available cores to *X*, where *X* varies between 1 and 8.

In Typescript, Benchmark.js[3] does not have a Command Line Interface, so we created one that would let us configure the *Benchmark* `Suites` with similar configurations to the ones used in Java, such as:
-i 8 -t 2

These options correspond to the following configuration:

- -i 8 - Set the minimum sample size required to perform statistical analysis to 8.

- -t 2 - Set the time needed to reduce the percent uncertainty of measurement to 1% to 2 seconds

4.3 Generated data Benchmarks

This section will explain the benchmarks we ran with runtime generated data. This helped in creating more straight-forward pipelines, even though it consequently makes the pipelines less representative of real-world use-cases. These benchmarks were ran with collection sizes of 1000 and 100 000 elements when comparing Java alternatives, and some of them were also ran with different data types like `Integer` or `number`, `String`, or even our custom *class* `Value` shown in Listings 4.2 and 4.3. This approach was based on the strategy used by kotlin-benchmarks[13].

```java
public class Value {
   public final int value;
   public final String text;

   public Value(int value) {
      this.value = value;
      this.text = new StringBuilder(
         String.valueOf(value)
      ).reverse()
      .toString();
   }
}
```

Listing 4.2: `Value` *class* in Java

```typescript
export class Value {
   public readonly value: number;
   public readonly text: string;

   constructor(value: number) {
      this.value = value;
      this.text = `${value}`
         .split('')
         .reverse()
         .join('');
   }
}
```

Listing 4.3: `Value` *class* in Typescript

4.3.1 All Match - *allMatch(sequence, predicate)*

This benchmark's pipeline consists of a single operation, *allMatch()*:

```
evenNumberSequence.allMatch(isEven);
```

This operation returns true if all the elements of a sequence match a certain predicate passed through the parameters. In this particular benchmark, the sequence tested was a sequence of even numbers and the predicate tested if a number is even so it would have to traverse the whole sequence in order to return true.

Figures 4.1 and 4.2 present the results obtained when running the All Match benchmark in Java. From these result we can say that, while the `Tinyield` solution is not the most performant it's close to the top, being behind Kotlin's `Sequence` only, and followed closely by `Eclipse Collections`. JOOλ, `Stream` and `StreamEx` all have similar performance and `Vavr` is by far the least performant solution.

"Kotlin's `Sequence`" and "Kotlin's `Sequence` in Java" both use the same sequence type, Kotlin's `Sequence`, being programmed in Kotlin and in Java respectively. These two approaches were benchmarked to test if using Kotlin's `Sequence` in Java, using `Sequence`'s operations exported as static methods, would hinder it's performance. As we can observe in figures 4.1 and 4.2, and as we'll observe throughout these benchmarks, this was not the case.

Figures 4.3 and 4.4 show the results obtained by running the All Match benchmark in Javascript. Comparing Figure 4.3 with Figure 4.4 we can tell that `ES6 Arrays` clearly perform much better with less elements, being the most performant option for sequences of 1000 elements. Looking at Figure 4.3, `Underscore`,

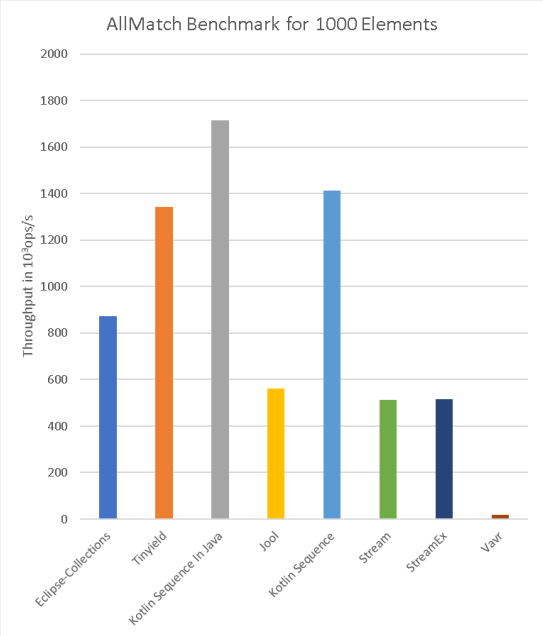

Figure 4.1: All Match benchmark results for Java with 1000 Elements

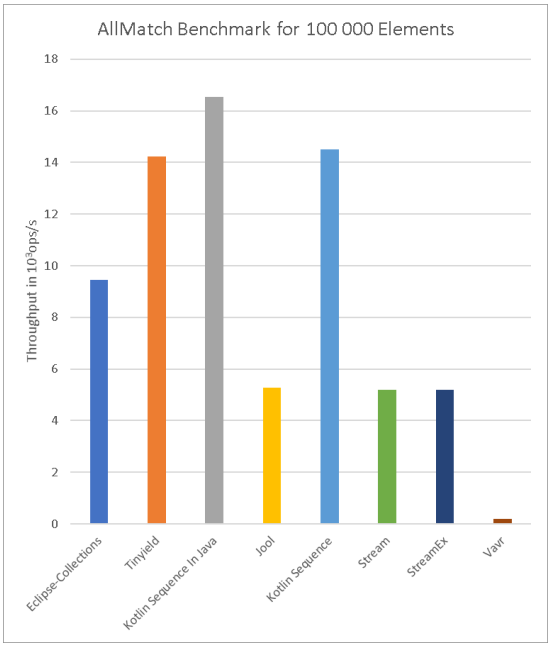

Figure 4.2: All Match benchmark results for Java with 100000 Elements

`Lodash`, `Sequency` and `Lazy.js` all have similar performance, with `Tinyield` trailing closely behind. `IxJs` manages to be the least performant in both benchmarked collections. In Figure 4.4, the results differ, with `Lodash`, `Underscore` and `Tinyield` as the top performers, in this order. Most approaches have a similar performance, with the exception of `IxJs` which is the least performant option.

`ES6 Arrays` disparity in performance presented in the results of figures 4.3 and 4.4 is justified by the eager nature of the approach, having little overhead when compared to the alternatives and thus faring better when presented with few elements. In opposition, lazy approaches, such as `Tinyield` and `Lazy.js`, will have the advantage when processing big collections.

4.3.2 Every - *every(sequence1, sequence2, predicate)*, Find - *find(sequence1, sequence2, predicate)* and First - *first(sequence, predicate)*

Every, Find and First are three similar benchmarks, therefore, they will be explained together in this subsection.

Every's pipeline relies on two distinct operations.

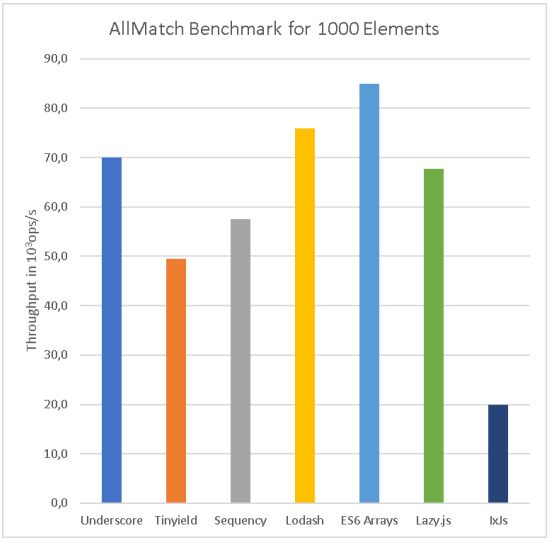

Figure 4.3: All Match benchmark results for Javascript/Typescript for 1000 elements

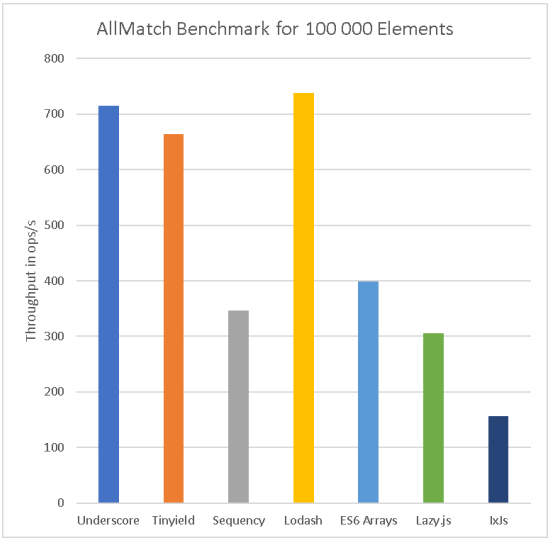

Figure 4.4: All Match benchmark results for Javascript/Typescript for 100000 elements

The first is *zip()* [15], a function that maps a Tuple of sequences into a sequence of tuples. To illustrate, let's imagine that zip is supported by `Stream` and that we have one *stream* with song names and another with the same song's artist's names, we could use *zip()* to pair them up as shown in Listing 4.4.

```
1 Stream songs = Stream.of("505", "Amsterdam", "Mural");
2 Stream artists = Stream.of("Arctic Monkeys", "Nothing But Thieves", "Lupe Fiasco");
3 songs
4     .zip(artists, (song, artist) -> String.format("%s by %s", song, artist))
5     .forEach(song -> System.out.println(song));
6
7 // -- Output --
8 // 505 by Arctic Monkeys
9 // Amsterdam by Nothing But Thieves
10 // Mural by Lupe Fiasco
```

Listing 4.4: Zip example

The other operation is an *allMatch()*, like the one presented in section 4.3.1. This benchmark's pipeline is as follows:

```
seq1.zip(seq2, pred::test).allMatch(Boolean.TRUE::equals);
```

In sum, *every* is an operation that, based on a user-defined predicate, tests if all the elements of a sequence match between corresponding positions. Listing 4.5 exemplifies this use-case.

```
1 Stream seq1 = Stream.of("505", "Amsterdam", "Mural");
2 Stream seq2 = Stream.of("505", "Amsterdam", "Mural");
3 Stream seq3 = Stream.of("Mural", "Amsterdam", "505");
4 BiPredicate pred = (s1, s2) -> s1.equals(s2);
5
6 every(seq1, seq2, pred); // returns true
7 every(seq1, seq3, pred); // returns false
```

Listing 4.5: Every example

We ran this Benchmark with different data types, namely we used sequences of `String`, `Integer` and `Value`, where both input sequences were equal, meaning we would have to go through the entire sequences.

Find, like Every, starts by applying the operation *zip()* between two sequences, but returns true as soon as a pair of elements match a predicate. Listing 4.6 showcases the pipeline.

```
1 seq1.zip(seq2, (t1, t2) -> predicate.test(t1, t2) ? t1 : null)
2    .filter(Objects::nonNull)
3    .findFirst()
4    .orElse(null);
```

Listing 4.6: Find pipeline

Listing 4.7 exemplifies the use-case of the Find operation, while also making clear the difference between the Find and Every operations. For instance the if line 6 of Listing 4.7 called *every()* instead of *find()* it would return *false* instead of "Amsterdam" which is only a match on the second element of each sequence.

```
1 Stream seq1 = Stream.of("505", "Amsterdam", "Mural");
2 Stream seq2 = Stream.of("400", "Amsterdam", "Stricken");
3 Stream seq3 = Stream.of("Amsterdam", "505", "Stricken");
4 BiPredicate pred = (s1, s2) -> s1.equals(s2);
5
6 find(seq1, seq2, pred); // returns "Amsterdam"
7 find(seq1, seq3, pred); // returns null
8 find(seq2, seq3, pred); // returns "Stricken"
```

Listing 4.7: Find example

Due to this similarity, we had to be careful not to benchmark the same behaviour, since if we had a match on the last element of a sequence the benchmark would run through the entire sequence, as Every would. With this in mind, we decided that benchmarking find with sequences with many elements and matching it only in the last element would not add much value to this analysis. So we devised a benchmark in which the match index would vary from the first index up until the last and another that would have the same match index every time depending on the collection size. As Every, this benchmark was run with sequences of `String`, `Integer` and `Value`.

Finally, First is quite simple, it benchmarks the usage of the *findFirst()* operation, after applying a *filter()* to a sequence. For this benchmark we used a sequence of even numbers with the exception of one element, the match, changing it between the first element, the middle or the last one, to see the performance difference in each case. The following code snippet shows this benchmark's pipeline:

```
sequence.filter(isOdd).findFirst();
```

4.3.2.1 Java Results

Every

Figure 4.5 shows Every benchmark results for 1000 element collections of Classes, Integers and Strings. These results are proportionally similar to each other across the different sequence types, although they all seem to be more performant with Class types. We have `Tinyield` as the most performant solution for this use-case with Kotlins `Sequence` close behind as well as `Eclipse Collections`. `Guava`, `StreamEx`, `Protonpack` and our custom *zip()* implementation through the use of a `Stream` *map()* and an `Iterator`, which we titled `Zipline`, all outperformed Java `Streams`, yet the performance isn't too different. Finally `jOOλ` and `Vavr` under-perform when compared with Java `Streams`, `Vavr` having close to a tenth of the performance.

Figure 4.6 show Every benchmark results for 100 000 element collections of Classes, Integers and Strings. String data shows different results than the other two, indicating that there may be some differences in the way different data types are handled by the different sequence types. Looking at the generic picture we can extract some conclusions, `Vavr` is by far the least performant option in all three, `jOOλ` is the only other approach to under-perform when compared to Java

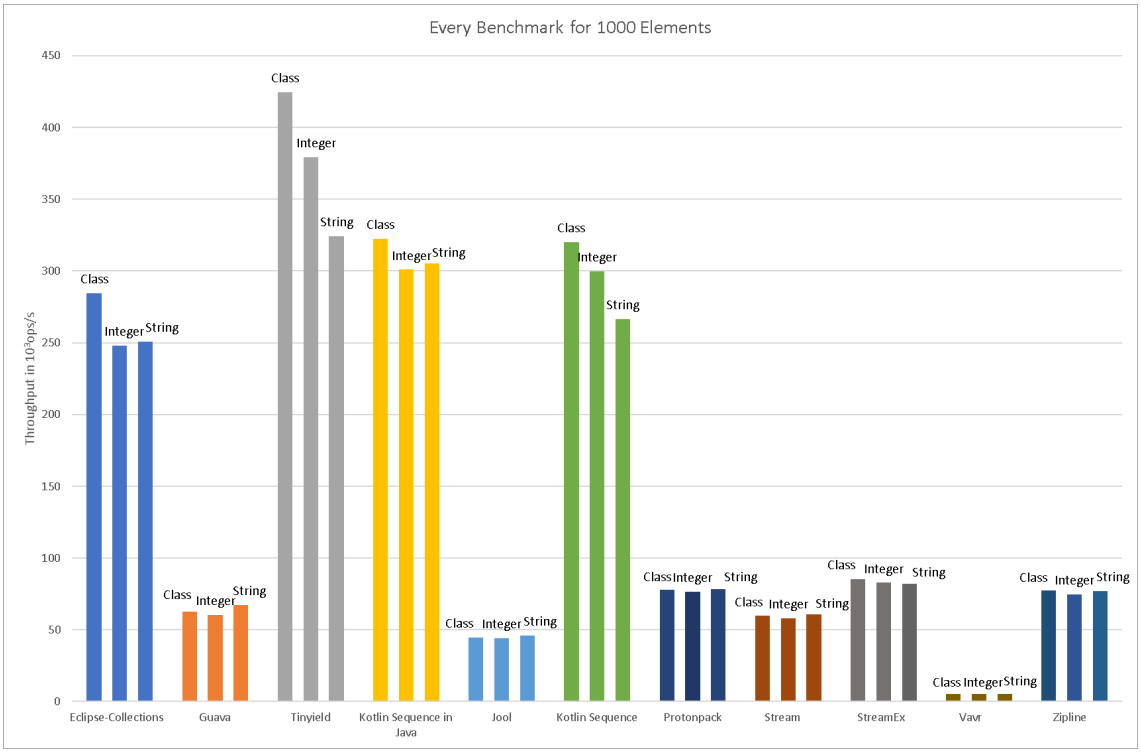

Figure 4.5: Every benchmark results for 1000 Elements

`Streams`. `Tinyield`, Kotlins `Sequence`, `Eclipse Collections`, `Guava`, `StreamEx`, `Protonpack` and `Zipline` all outperform Java `Streams` with most data types. For classes `Eclipse Collections` was the most performant, with Kotlins `Sequence` close behind, and in this particular use-case `Guava` underperforms in relation to Java `Streams`. For Integers Kotlins `Sequence` and `Eclipse Collections` are the most performant options with Kotlins `Sequence` slightly ahead. Finally, for Strings `Tinyield` is by far the most performant option followed by Kotlin's `Sequence`, although we could not explain this disparity in performance between data types for `Tinyield`.

Find

Find benchmark results for 1000 element collections are found in Figure 4.7. Kotlins `Sequence` clearly outperform all the competition in this use-case, specially for Integer data types. `Vavr` is once more the least performant option, yet this time, Java `Streams` outperform not only `jOOλ` but `Tinyield` as well, and even `Protonpack` with String data.

Figure 4.8 presents the Find benchmark results for 100 000 element collections

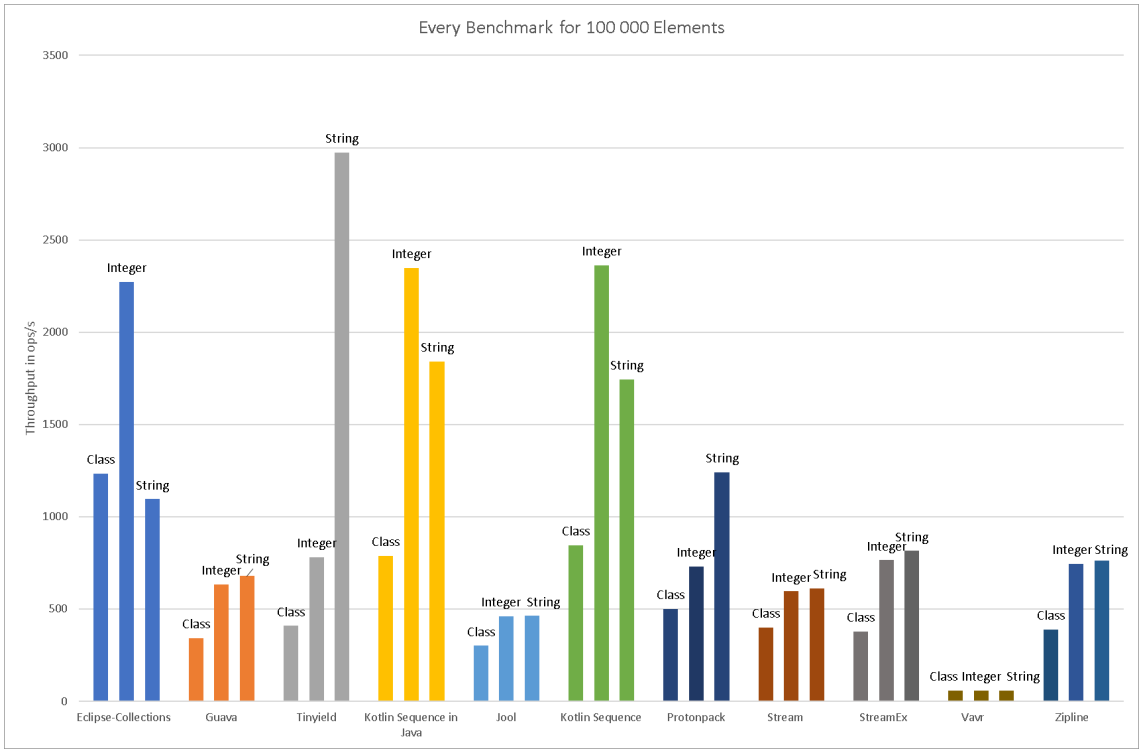

Figure 4.6: Every benchmark results for 100000 Elements

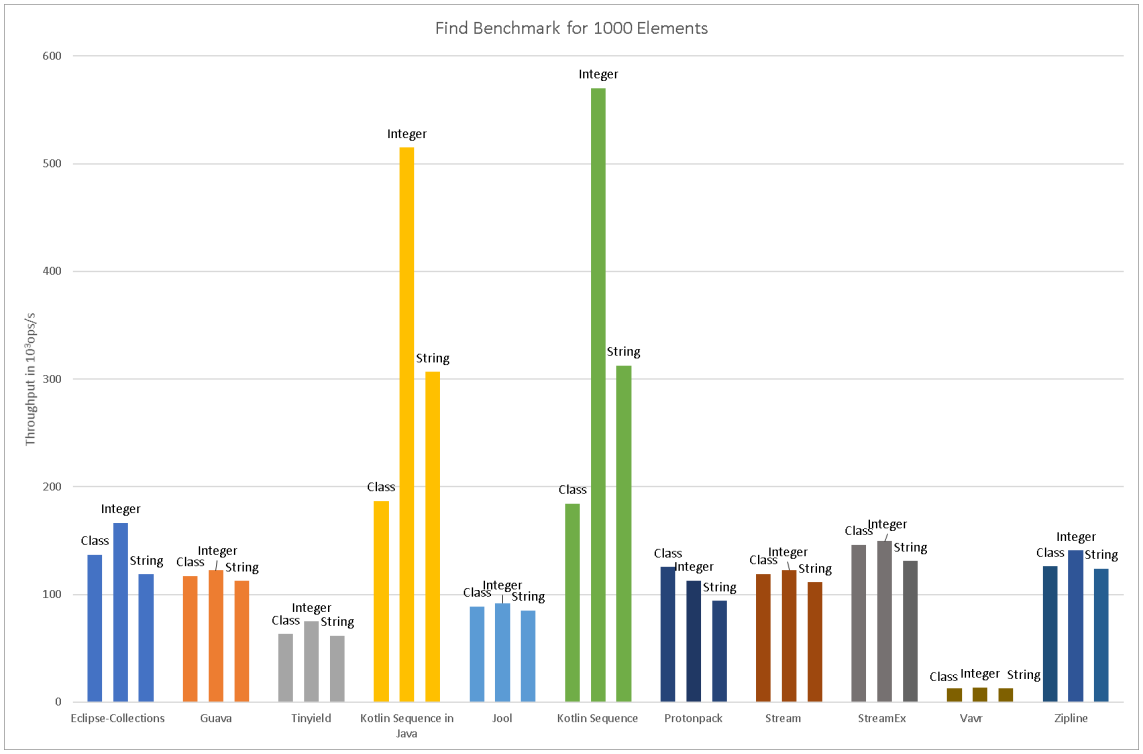

Figure 4.7: Find benchmark results for 1000 Elements

of Classes, Integers and Strings. When comparing with the results for 1000 element collections, these results seem to be much more uniform. Nonetheless, `Eclipse Collections` is the most performant approach when using Classes and Strings, while Kotlins `Sequence` outperforms the competition when using Integers. `Vavr` is once again the least performant option, yet Java `Streams` performance is on par with most solutions, outperforming `jOOλ`, `Guava` and `Tinyield` when using Integers, as well as `Zipline` when using Classes.

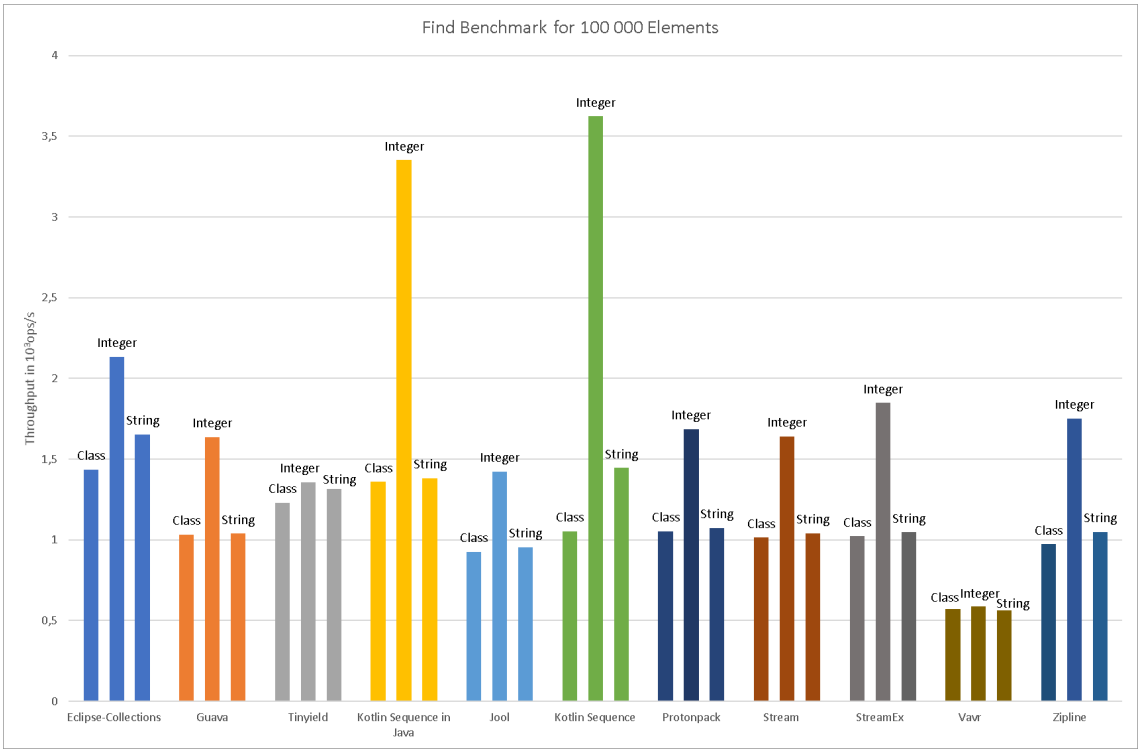

Figure 4.8: Find benchmark results for 100000 Elements

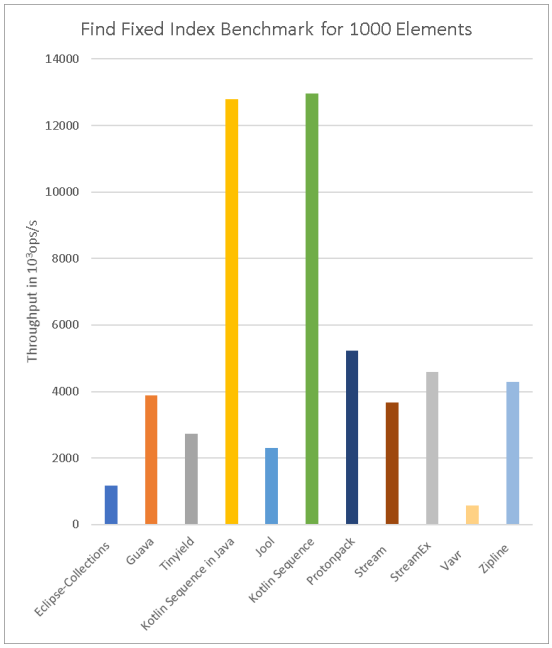

Figure 4.9: Find benchmark results with fixed match index with 1000 Elements

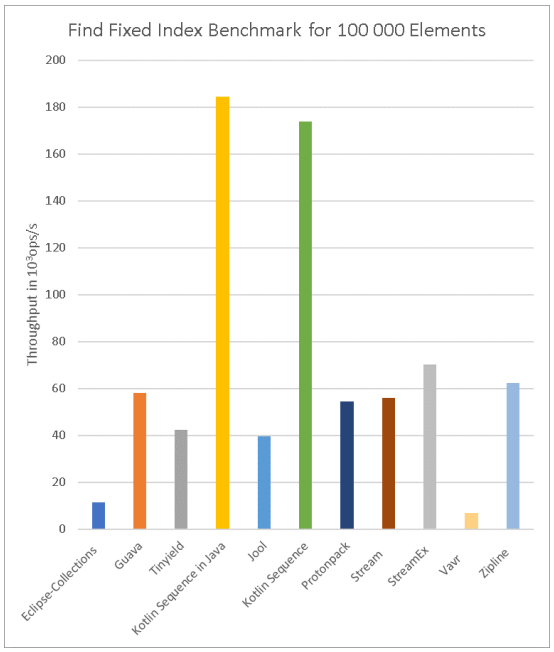

Figure 4.10: Find benchmark results with fixed match index with 100000 Elements

The Find in a fixed index benchmark results are presented in Figures 4.9 and 4.10. These results are similar in proportion, Kotlins `Sequence` outperform by far all the other approaches, this can be explained by how Kotlin handles terminal operations as well as the lack of creating wrapper objects such as `Optionals`. `Vavr` is once again the least performant solution being outperformed just slightly by `Eclipse Collections` in this benchmark.

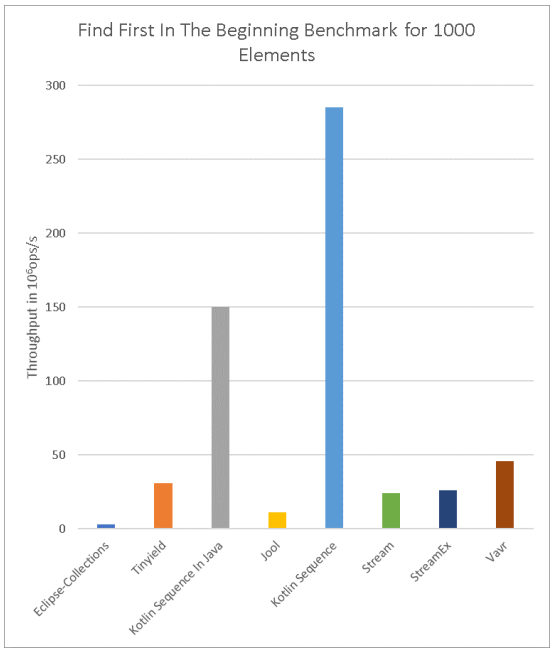

Figure 4.11: First benchmark results for match in the beginning of a sequence with 1000 elements

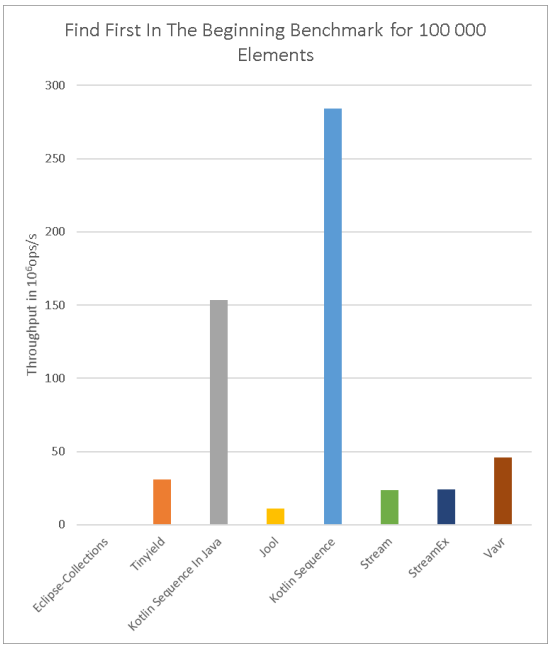

Figure 4.12: First benchmark results for match in the beginning of a sequence with 100000 elements

First

The First benchmark results are split into 6 different figures. Figures 4.11 and 4.12 are the results for this benchmark with the match made in the first element, Figures 4.13 and 4.14 are the results for this benchmark with the match made in the middle element and finally, Figures 4.51 and 4.16 are the results for this benchmark with the match made in the last element. From these results we can say that Kotlins `Sequence` has the best performance by far. `Eclipse Collections` and `jOOλ` gain performance comparing with Java `Streams` the latter in the match is made in the sequence, while `Tinyield` and `Vavr` lose it. `StreamEx` is the only approach that keeps on par with Java `Streams` throughout these benchmarks. We can also say that for the results obtained where the match element is the first of the sequence, in Figures 4.11 and 4.12, the number of elements in the sequence seems to be rather irrelevant for most sequence types, as the throughput is almost the same, the clear exception to this is `Eclipse Collections`, this makes sense when taking into account the behaviour of `Eclipse Collections`, as it will first process the whole sequence through the *filter()* operation and only then will it find the first element, this leads to a lot of unnecessary processing.

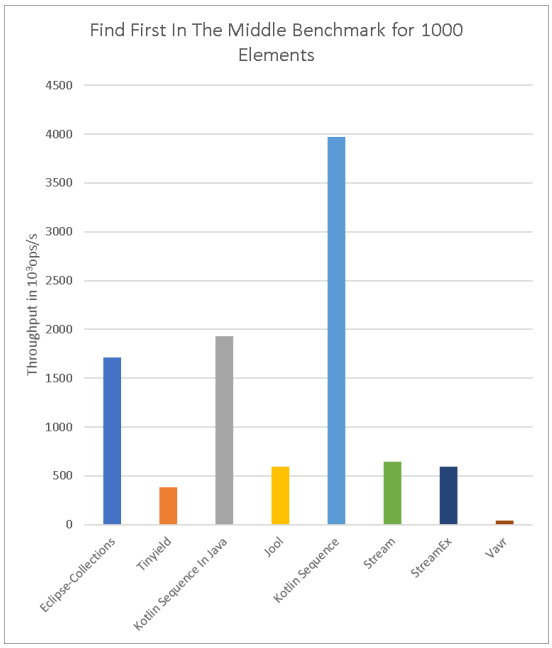

Figure 4.13: First benchmark results for match in the middle of a sequence with 1000 elements

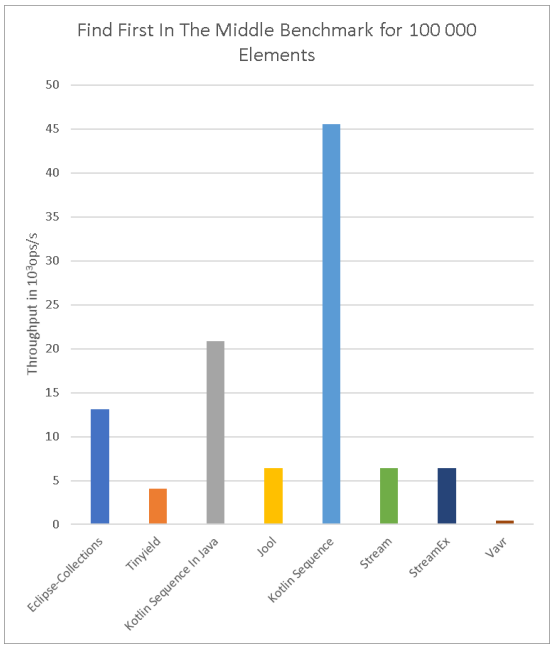

Figure 4.14: First benchmark results for match in the middle of a sequence with 100000 elements

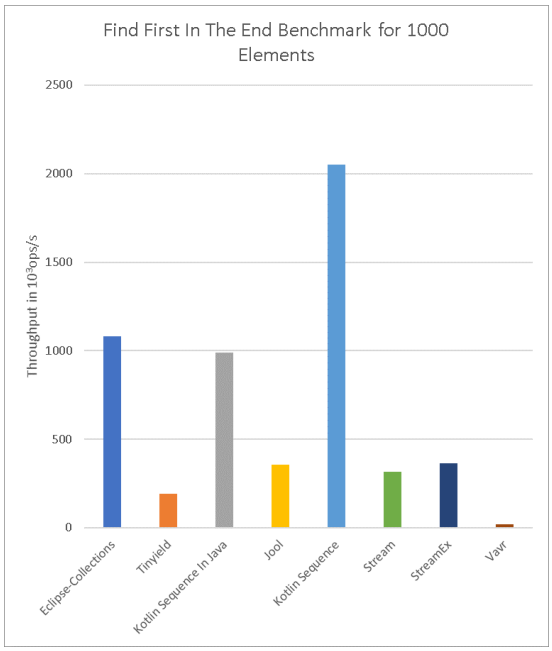

Figure 4.15: First benchmark results for match in the end of a sequence with 1000 elements

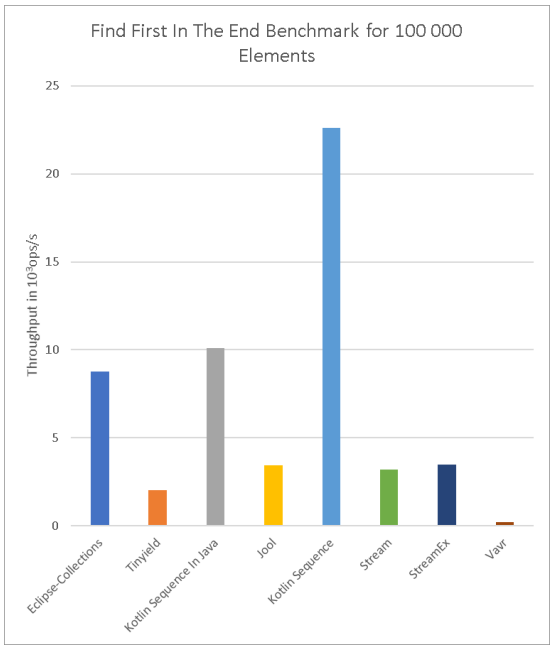

Figure 4.16: First benchmark results for match in the end of a sequence with 100000 elements

4.3.2.2 Javascript Results

Every

Figure 4.17 show the results for Every with sequences of 1000 elements. In sequences of this size `Es6 Arrays` and `Tinyield` are the top two performers, with `Lazy.js` and `Lodash close behind`. On the other hand `IxJs` clearly underperforms comparing to all other solutions.

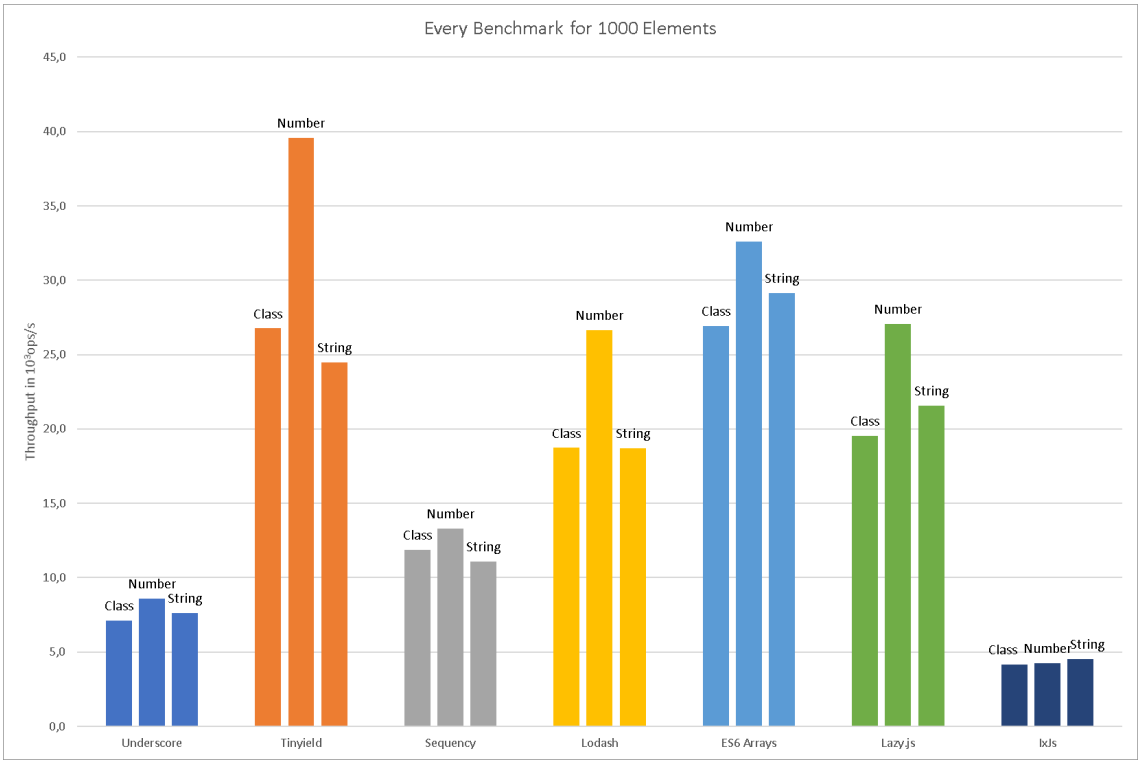

Figure 4.17: Every benchmark results for 1000 Elements

For Every run with sequences of 100 000 elements things clearly change, as shown in Figure 4.18. In sequences of this size `Tinyield` is now the most performant of all the selected approaches, followed by `Lazy.js`. On the other end, `Underscore` is now the least performant option, as for `IxJs`, while it performs better, it still underperforms `ES6 Arrays` slightly, which have dropped in performance comparing to all alternatives except these two.

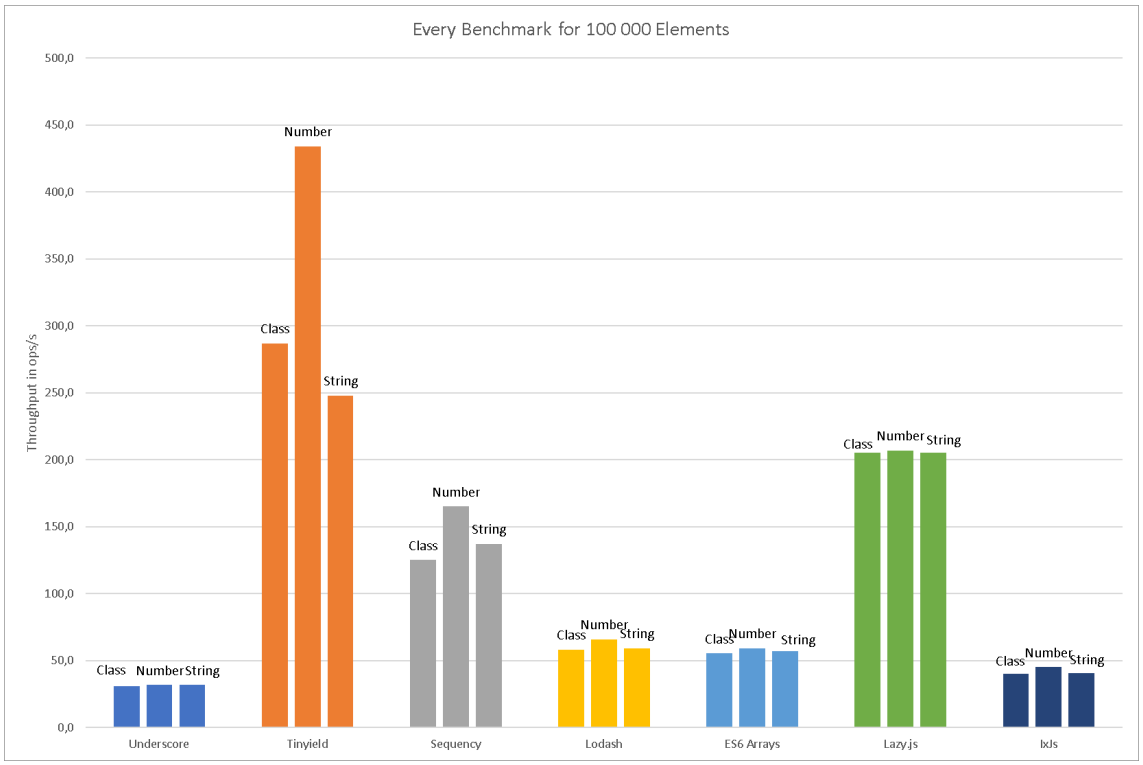

Figure 4.18: Every benchmark results for 100000 Elements

Find

Figure 4.19 shows the results for Find with sequences of 1000 elements. The top two performing solutions here are `Tinyield`, `Lazy.js`, in this order, although `Tinyield` is surpassed by `Lazy.js` when running with class data. `Underscore` is the least performant, being just below `IxJs`.

For sequences of 100 000 elements in the Find benchmark the results change, as shown in Figure 4.20. The top three performing solutions are now `Tinyield`, `Lazy.js` and `IxJs`, in this order, followed by `Sequency`. `Underscore` is the least performant option, even though it has similar performance to `Lodash` and even `ES6 Arrays`.

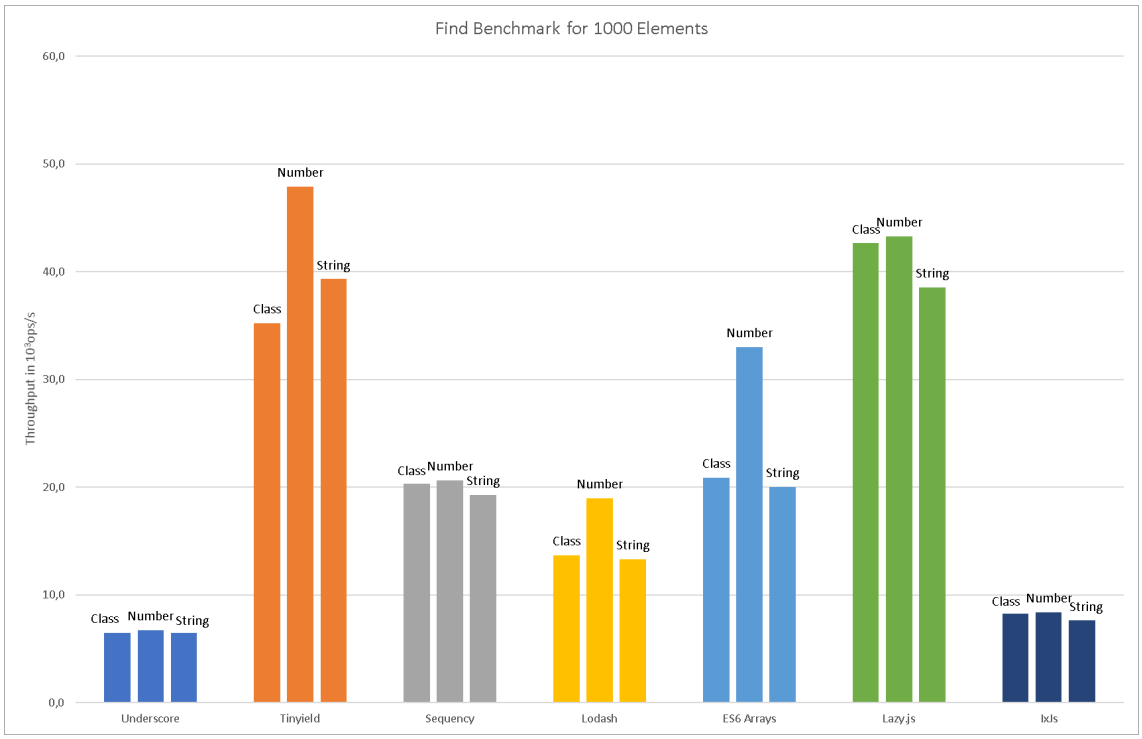

Figure 4.19: Find benchmark results for 1000 Elements

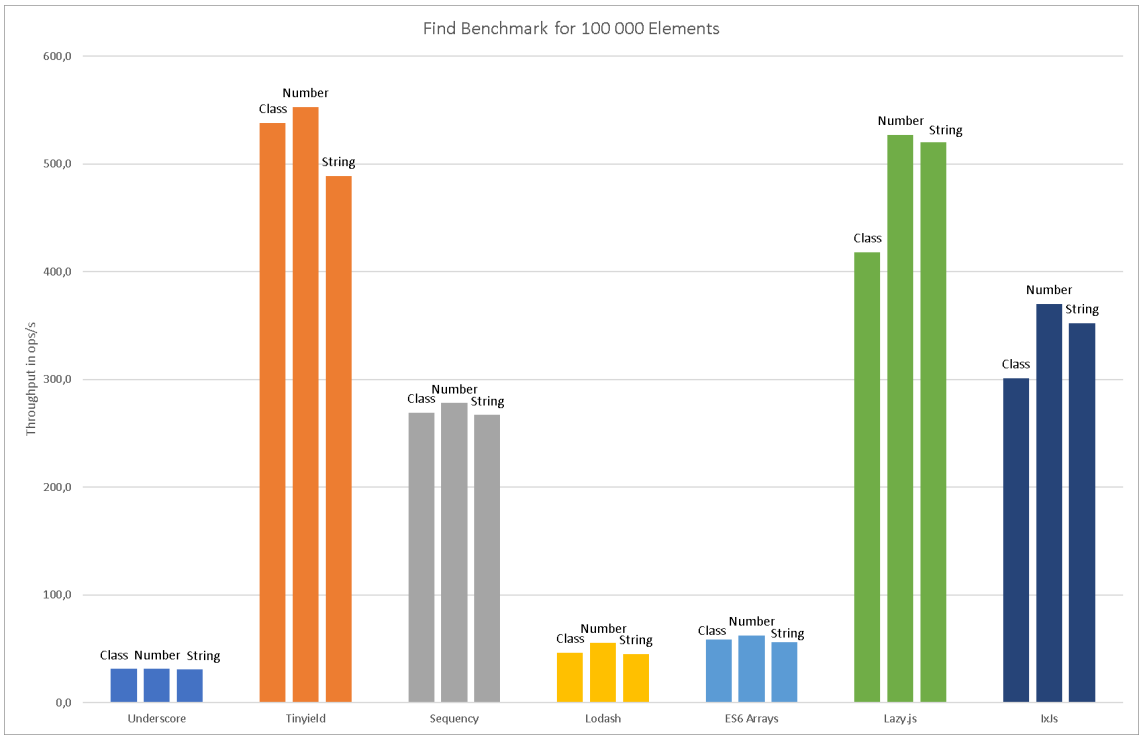

Figure 4.20: Find benchmark results for 100000 Elements

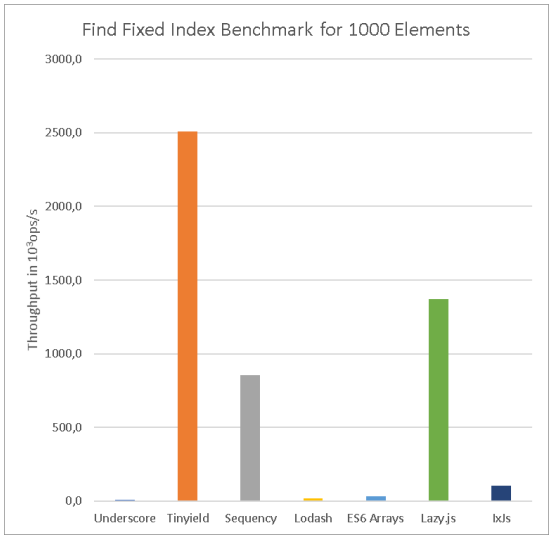

Figure 4.21: Find benchmark results with fixed match index with 1000 Elements

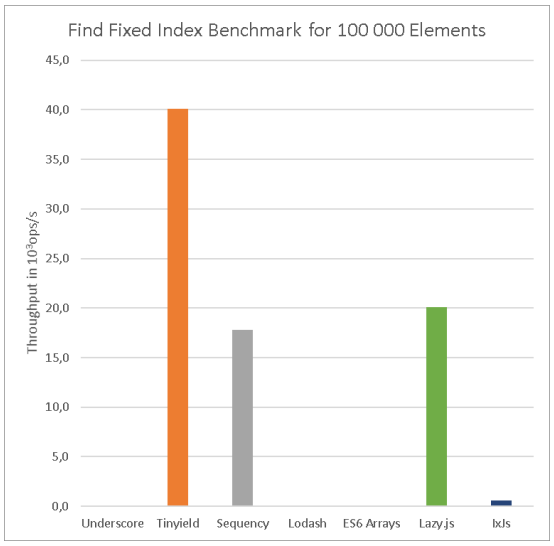

Figure 4.22: Find benchmark results with fixed match index with 100000 Elements

In the Fixed index benchmark results, shown in Figures 4.21 and 4.22, we also see a big discrepancy between the top 3 performant solutions, which are `Tinyield`, `Lazy.js` and `Sequency`, in this order, and the remaining selected approaches. This is because all the other solutions apart from `IxJs` save intermediate results for each operation as they go through the whole sequence for each operation, this leads to a lot of unnecessary processing.

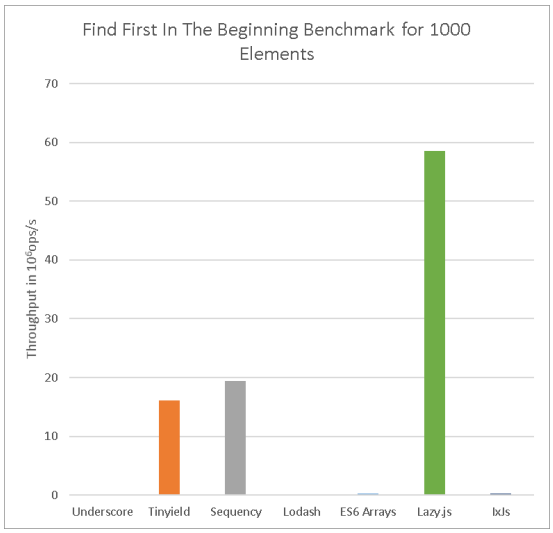

Figure 4.23: First benchmark results for match in the beginning of a sequence with 1000 elements

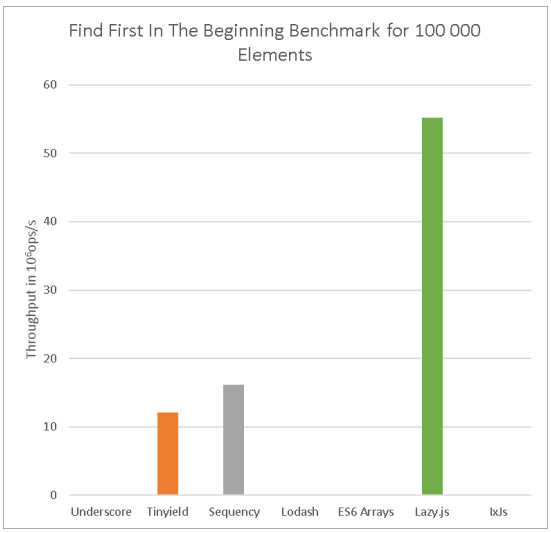

Figure 4.24: First benchmark results for match in the beginning of a sequence with 100000 elements

First

The First benchmark results are split into 6 different figures. Figures 4.23 and 4.24 are the results for this benchmark with the match made in the first element, Figures 4.25 and 4.26 are the results for this benchmark with the match made in the middle element and finally, Figures 4.27 and 4.28 are the results for this benchmark with the match made in the last element. Looking at these figures we can tell that `Lazy.js` is by far the best alternative for this benchmark. `Tinyield` and `Sequency` perform perform well with less elements and the sooner the element is found. `IxJs` is once again the least performant solution. Nevertheless, `Underscore`, `Lodash` and `ES6 Arrays` all have poor performance, specially the sooner the element is in the sequence and the more elements the sequence has, this is due to how these sequences process operations, only finding the element after the *filter()* operation is concluded, processing a lot of unnecessary elements.

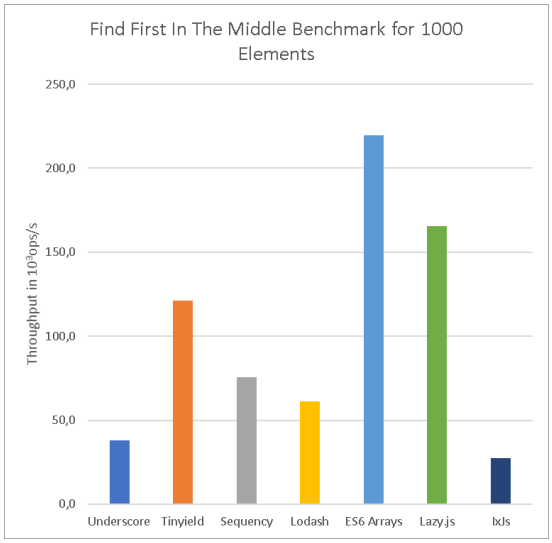

Figure 4.25: First benchmark results for match in the middle of a sequence with 1000 elements

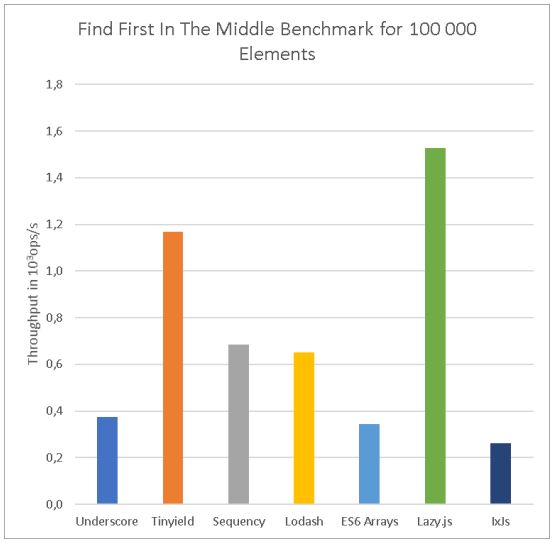

Figure 4.26: First benchmark results for match in the middle of a sequence with 100000 elements

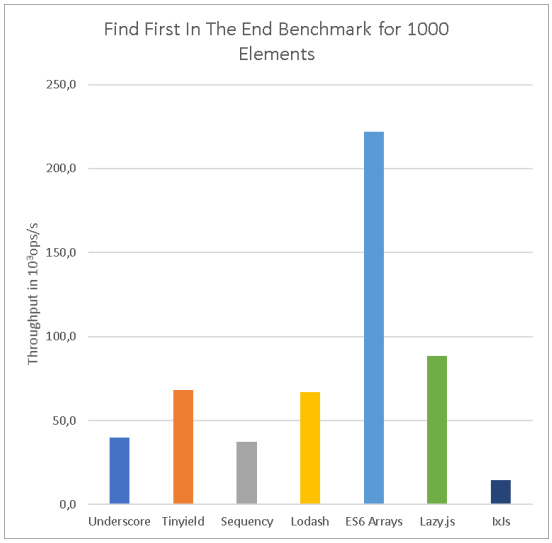

Figure 4.27: First benchmark results for match in the end of a sequence with 1000 elements

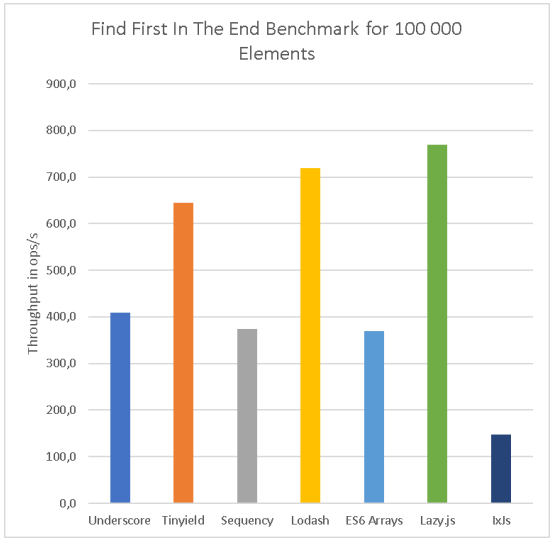

Figure 4.28: First benchmark results for match in the end of a sequence with 100000 elements

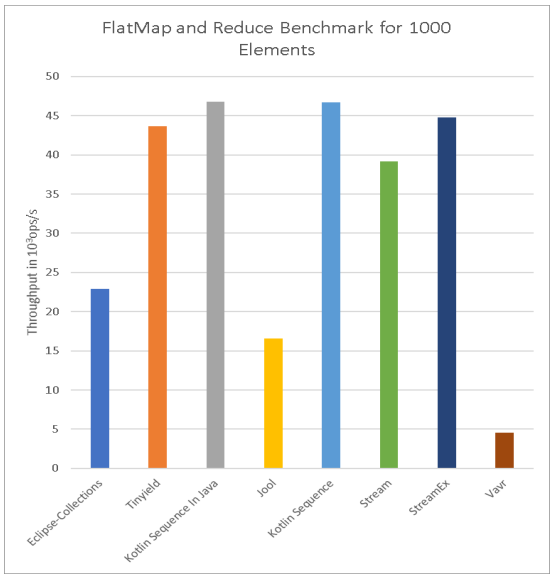

Figure 4.29: Flatmap and Reduce benchmark results for Java with 1000 Elements

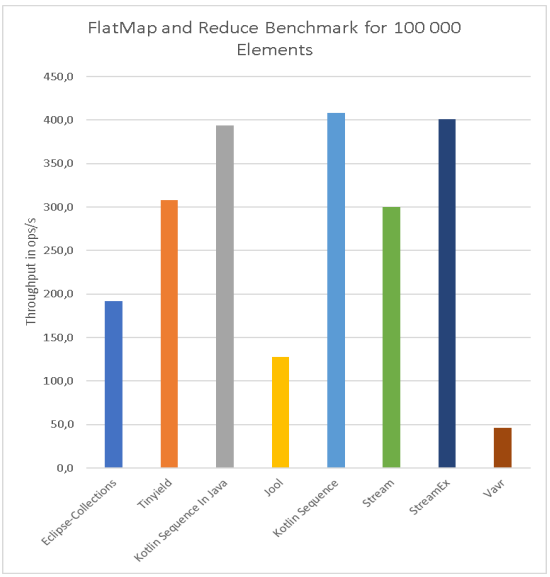

Figure 4.30: Flatmap and Reduce benchmark results for Java with 100000 Elements

4.3.3 Flatmap and Reduce

This benchmark evaluates the combination of two operations in what is a common use-case, these are the operations *flatMap()* and *reduce()*. For this benchmark we simply used sequences of sequences of numbers in order to *flatMap()* them into a single sequence of all the numbers and then *reduce()* them by summing all the numbers together. This benchmark's pipeline can be represented by the following snippet:

```
sequenceOfSequencesOfNumbers.flatMap(s->s).reduce(Integer::sum);
```

Figures 4.29 and 4.30 present the results for running the Flatmap and Reduce benchmark in Java. Here we can see that Kotlins `Sequence` is the top performer, followed closely by `StreamEx` and `Tinyield` in that order. Java `Streams` isn't much behind the top performers, yet it leaves `Eclipse Collections` and `jOOλ` behind in performance, having almost double the performance of these two. Yet again, `Vavr` is by far the least performant option.

In the Javascript / Typescript world, we have a consistent top two consisting of `Tinyield` and `Lazy.js`,`Tinyield` performing better with less elements and `Lazy.js` performing better with more, as is presented by Figures 4.31 and 4.32. `ES6 Arrays` do not perform very well in this benchmark, having close performance to that of `IxJs` and `Undersocre`, and with 100 000 elements even `Lodash`. `Undersocre` is the least performant approach for this benchmark.

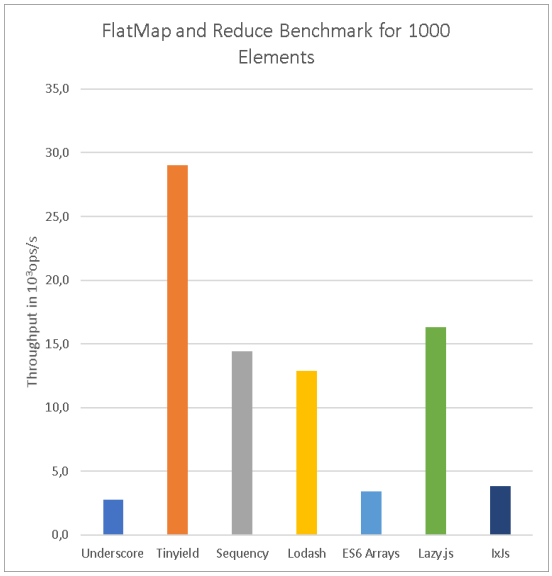

Figure 4.31: Flatmap and Reduce benchmark results for Javascript with 1000 Elements

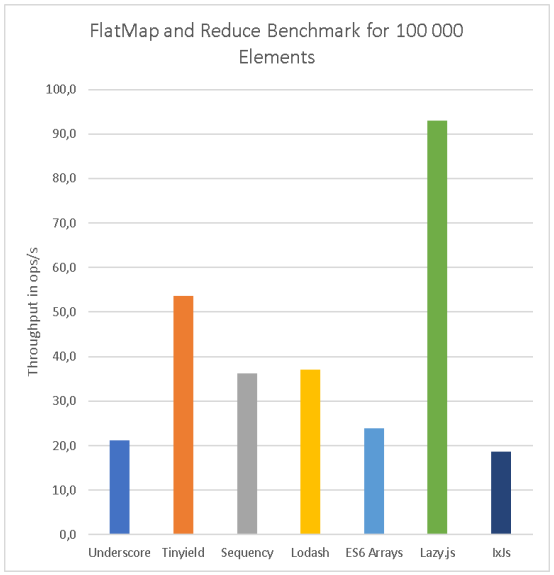

Figure 4.32: Flatmap and Reduce benchmark results for Javascript with 100000 Elements

4.3.4 Zip Primes with Values - *zip(primes, values)*

This benchmark mostly aims to isolate the *zip()* operation and see how each solution performs. To achieve this we simply *zip()* a sequence of prime numbers with a sequence of instances of our custom `Class Value`. Listing 4.8 showcases the pipeline.

```
1 sequenceOfIntegers.filter(isPrime)
2     .zip(sequenceOfValues, Pair::with)
3     .forEach(bh::consume)
```

Listing 4.8: Zip Primes with Values pipeline

Java's results, shown in Figures 4.33 and 4.34, show equivalent performance across the board, with the exception of `Vavr` which still performs worse than every other option. For 1000 element sequences `Tinyield` is the top performer, followed closely by `Eclipse Collections` and Kotlins `Sequence`, in that order. With 100 000 elements the performance is really close, yet `StreamEx` is the top performer.

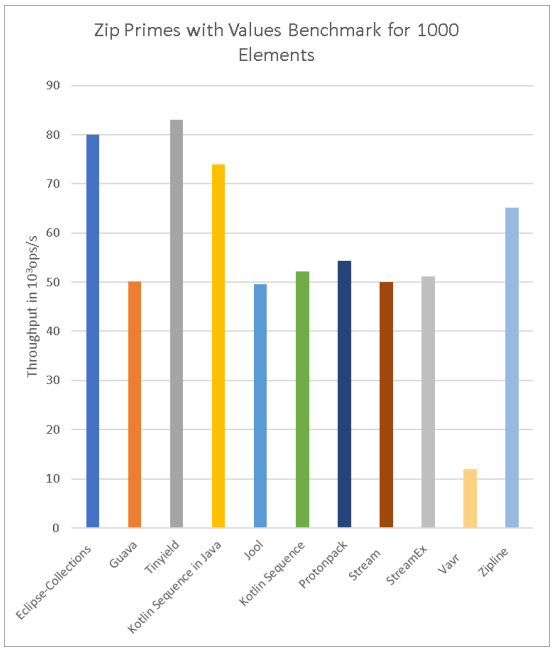

Figure 4.33: Zip Primes with Values benchmark results for Java with 1000 Elements

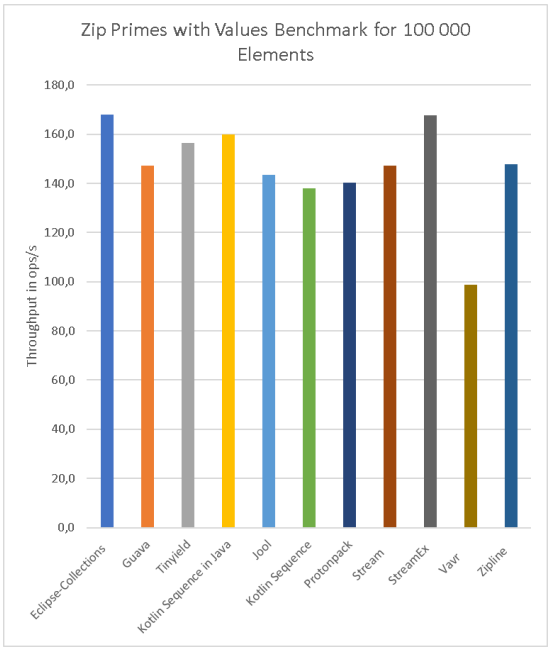

Figure 4.34: Zip Primes with Values benchmark results for Java with 100000 Elements

In Typescript `Tinyield` is always the top performing option for this benchmark, followed closely by `ES6 Arrays`, `Sequency` and `Lazy.js`. In opposition, `Underscore` is the least performant option for this benchmark.

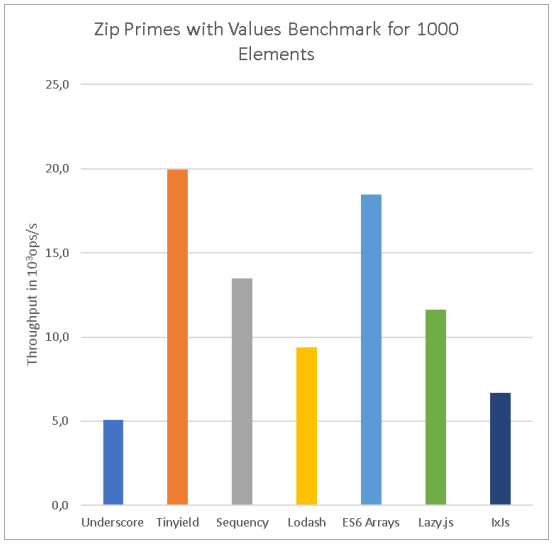

Figure 4.35: Zip Primes with Values benchmark results for Javascript with 1000 Elements

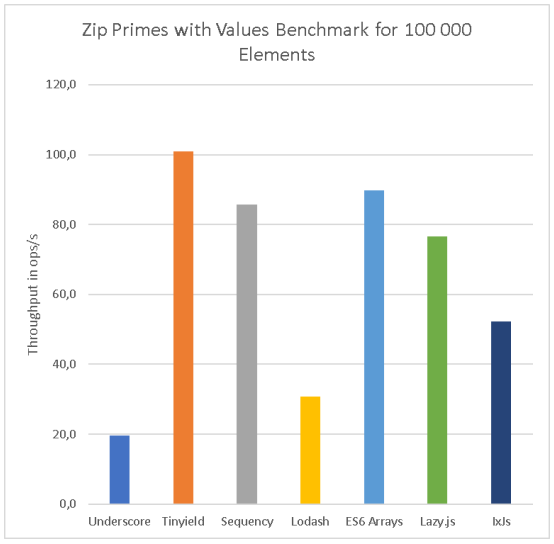

Figure 4.36: Zip Primes with Values benchmark results for Javascript with 100000 Elements

4.4 Real World data Benchmarks

Looking for more complex pipelines with realistic data, we resort to publicly available Web APIs, namely REST Countries[2] and Last.fm[3], as well as World Weather Online[4]. We retrieved from the REST Countries a list of 250 countries which we then used to query the Last.fm API in order to retrieve both the top Artists as well as the top Tracks by country, leaving us with a total of 7500 entries for each. From World Weather Online we retrieved a *.csv* file with the weather data from 2017-02-01 to 2017-04-30.

4.4.1 REST Countries and Last.fm

With the use of REST Countries and Last.fm we devised two benchmarks, one that would zip together the top Artist and Track for each Country discarding tuples with the same artist and another that would pair a list of Artists that are

[2]https://restcountries.eu/
[3]https://www.last.fm/api/
[4]https://www.worldweatheronline.com/developer/

in a Country's top ten who also have Tracks in that same Country's top ten, with the Country.

Our domain model can be summarized into the following definition of these classes: Country, Language, Artist, and Track. Figure 4.37 shows the UML representation of this model.

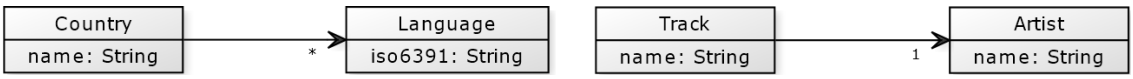

Figure 4.37: Domain Model

Both our benchmarks have the same basis. We first query all the countries, filter those that don't speak English and from there we retrieve two Sequences: one with the pairing of the Country and its top Tracks, shown in Listing 4.10, and another with the Country and its top Artists, shown in Listing 4.9.

```
1 Sequence<Pair<Country, Sequence<Artist>>> getArtists() {
2    return getCountries()
3          .filter(this::isNonEnglishSpeaking)
4          .map(country -> Pair.with(country, getArtistsForCountry(country)));
5 }
```

Listing 4.9: Artists by Country

```
1 Sequence<Pair<Country, Sequence<Track>>> getTracks() {
2    return getCountries()
3          .filter(this::isNonEnglishSpeaking)
4          .map(country -> Pair.with(country, getTracksForCountry(country)));
5 }
```

Listing 4.10: Tracks by Country

From here on out these two benchmarks diverge, we'll explain each one in the next sections.

4.4.2 Distinct Top Artist and Top Track by Country Benchmark - *zip(artists, tracks)*

This pipeline relies on both the previous presented pipelines (Listings 4.9 and 4.10), combining them into a Trio of the top Artist and Track by country through a *zip()* operation, and finally using *distinct* to keep only unique artists, displayed in Listing 4.11.

```
1  Sequence<Trio<Country,Artist,Track>> zipTopArtistAndTrackByCountry() {
2     return getArtists()
3           .zip(getTracks())
4           .map(pair -> Trio.with(
5                 pair.first().country,
6                 pair.first().artists.first(),
7                 pair.second().tracks.first()
8           ))
9           .distinctByKey(Trio::getArtist);
10 }
```

Listing 4.11: Distinct Top Artist and Top Track by Country pipeline

From the results in Java, shown in Figure 4.38, we can see that `Tinyield` only performs worse than Kotlin's `Sequence`, although there is similar performance ac cross the all alternatives. Yet, `Vavr` is the least performant option. In Javascript the results differ, as presented in Figure 4.39. `Tinyield` is the top three performer, having similar performance to `Sequency` and coming behind `Lazy.js` and `ES6 Arrays`. In opposition, `Underscore` is the least performant option for this benchmark.

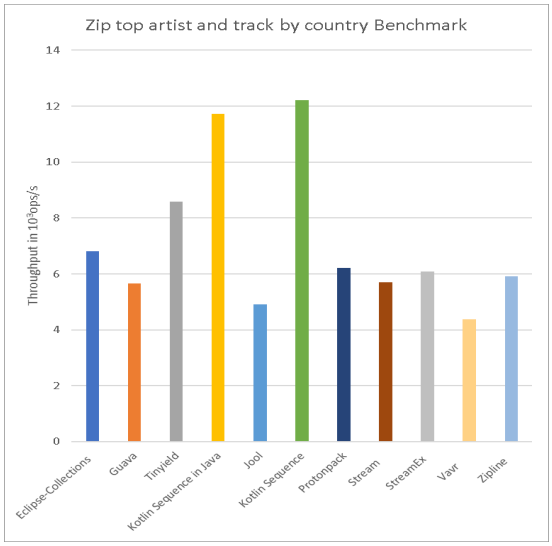

Figure 4.38: Distinct Top Artist and Top Track by Country benchmark results in Java

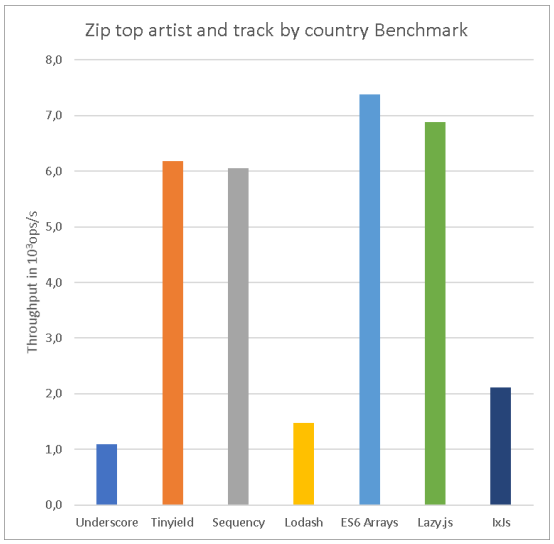

Figure 4.39: Distinct Top Artist and Top Track by Country benchmark results in Javascript

4.4.3 Artists who are in a Country's top ten who also have Tracks in the same Country's top ten Benchmark - *zip(artists, tracks)*

Building on the same two operations as a basis, this pipeline also combines both sequences through a *zip()* operation, but this time into a trio of the top ten artists and top ten tracks by country, finishing by transforming this trio into a pair of the top ten artists filtered by whether or not they have a track in the top ten for the current country through a *map()* operation, shown in Listing 4.12.

```
1  Sequence<Pair<Country,Sequence<Artist>>> artistsInTopTenWithTopTenTracksByCountry() {
2     return getArtists()
3           .zip(getTracks())
4           .map(pair -> Trio.with(
5                 pair.first().country,
6                 pair.first().artists.limit(10),
7                 pair.second().tracks.limit(10)
8                       .map(Track::getArtist)
9                       .map(Artist::getName)
10          )).map(trio -> Pair.with(
11             trio.country,
12             trio.artists.filter(artist -> trio.tracksArtists.contains(artist.name))
13          ));
14 }
```

Listing 4.12: Artists who are in a Country's top ten who also have Tracks in the same Country's top ten pipeline

Looking at the results in Java, presented in Figure 4.40, we can observe that `Eclipse Collections` outperform the competition, followed closely by `Tinyield` and Kotlins `Sequence`, while the rest of the alternatives fare similarly to each other with the exception of `Vavr` that performs quite worse. From the Javascript results, in Figure 4.41, the top three performers are `Lazy.js`, `ES6 Arrays` and `Tinyield` in this order. In opposition, `Underscore` is once more the least performant option, yet, it has similar performance to `Lodash` and `IxJs`.

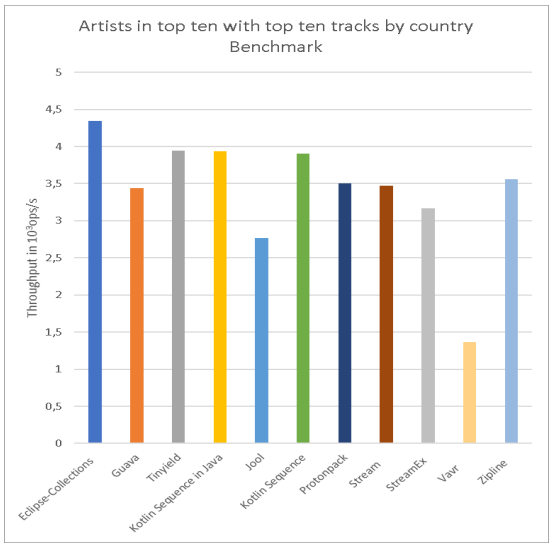

Figure 4.40: Artists who are in a Country's top ten who also have Tracks in the same Country's top ten benchmark results in Java

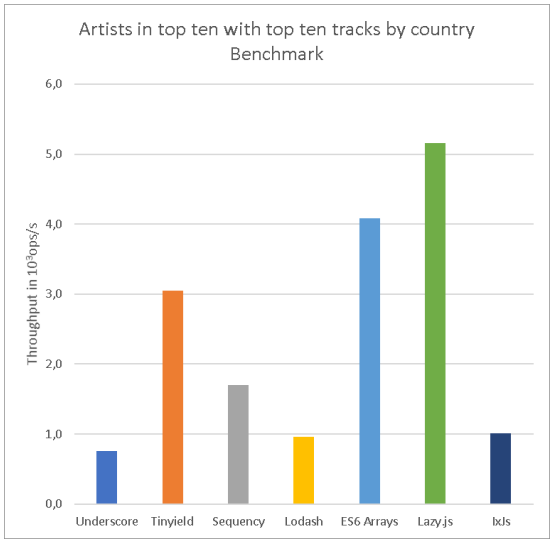

Figure 4.41: Artists who are in a Country's top ten who also have Tracks in the same Country's top ten benchmark results in Javascript

4.4.4 World Weather Online

Using World Weather Online's data we devised three benchmarks, one that would find the number of distinct temperatures during the defined period, another that would find the highest temperature and yet another that would count the number of temperature transitions.

These benchmarks needed to process a *.csv* file with the data, we used this oportunity to create a custom operation *oddLines* that yields only the oddlines in a sequence of `strings`, in order to benchmark the overhead for adding a user defined operation to a pipeline. We also added a user defined *collapse* operation, even though some of the alternatives provide it out of the box.

4.4.5 Query Distinct Temperatures

As stated above this benchmark needs to process a *.csv* file, to do this, we first convert the file into a data source, we then proceed to *filter()* out the comments, we *skip()* the first line seeing as it has no relevant information and then proceed to call our *oddLines* operation that, by yielding only the odd lines, filters out the hourly info of each day, after that we *mapToInt()* parsing the temperatures selecting only the *distinct()* and finally we call *count()* to retrieve the number of distinct temperatures in our data set. Listing 4.13 showcases the pipeline.

```
source
    .filter(s -> s.charAt(0) != '#') // Filter comments
    .skip(1) // Skip line: Not available
    .oddLines() // Filter hourly info
    .mapToInt(line -> parseInt(line.substring(14, 16))) // Parse Temperature
    .distinct()
    .count();
```

Listing 4.13: Query Distinct Temperatures pipeline

In Java `Tinyield` is the top performer, as we can see in Figure 4.42, followed closely by Java `Streams` and, surprisingly, by `jOOλ`. We attribute this result of `jOOλ` to the fact that it is really just a wrapper to Java `Streams` with more operations. On the other end `Vavr` is still the least performant, but `StreamEx` is not to far from it in regards to performance for this benchmark. In Javascript, the top performer is also `Tinyield`, as shown in Figure 4.43, followed closley by `ES6 Arrays` and `Lazy.js`, in this order. For this benchmark, the least performant options are `Sequency` and `IxJs`. Although `IxJs` is the least performant, the fact that `Sequency` has about half the throughput of other options, seems to indicate that it doesn't handle user-defined operations well, or at the very least not as well as the other sequence types.

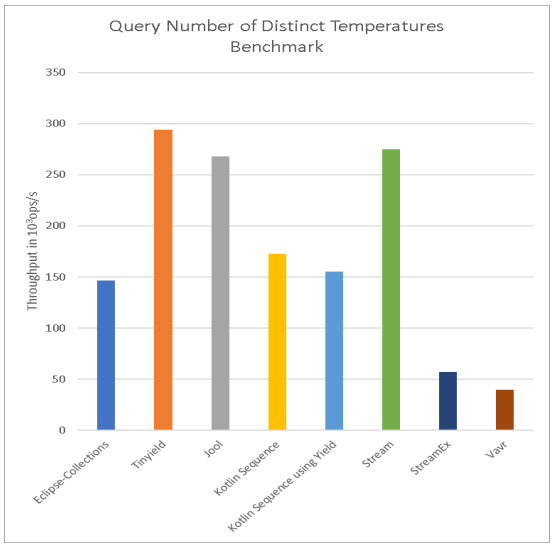

Figure 4.42: Query Distinct Temperatures benchmark results in Java

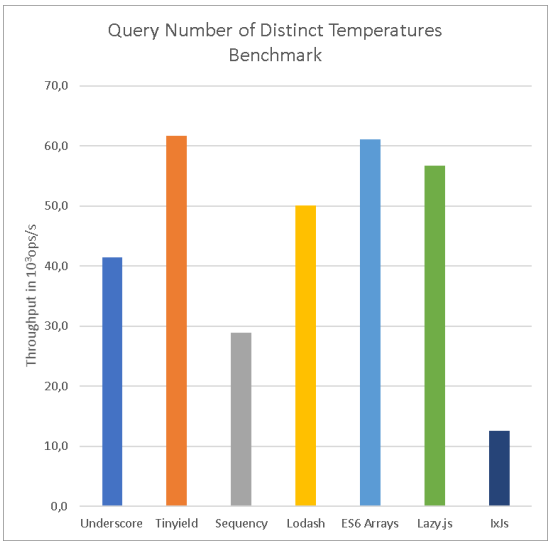

Figure 4.43: Query Distinct Temperatures benchmark results in Javascript

4.4.6 Query Max Temperature

This benchmark is very similar to the one in section 4.4.5, differing from it after the *mapToInt* operation, where instead of calling *distinct()* and *count()* we simply call *max()* to retrieve the highest temperature in the data set. Listing 4.14 showcases the pipeline.

```
1 source
2     .filter(s -> s.charAt(0) != '#') // Filter comments
3     .skip(1) // Skip line: Not available
4     .oddLines() // Filter hourly info
5     .mapToInt(line -> parseInt(line.substring(14, 16))) // Parse Temperature
6     .max();
```

Listing 4.14: Query Max Temperature pipeline

Figures 4.44 and 4.45 show the results for this benchmark in Java and Javascript, respectively. These results are similar in to the ones obtained for section 4.4.5 as we again have `Tinyield` as the best performer followed by Java `Streams` and `jOOλ`, as well as `Vavr` and `StreamEx` as the least performant options in Java. In Typescript `Tinyield` was the second best performer, scoring below `ES6 Arrays` only and once again we have `Sequency` and `IxJs` as the least performant options, reinforcing the point about `Sequency` not handling custom operations well.

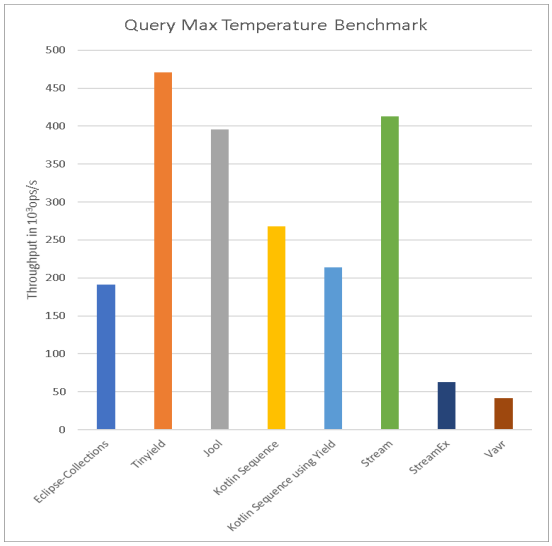

Figure 4.44: Query Max Temperature benchmark results in Java

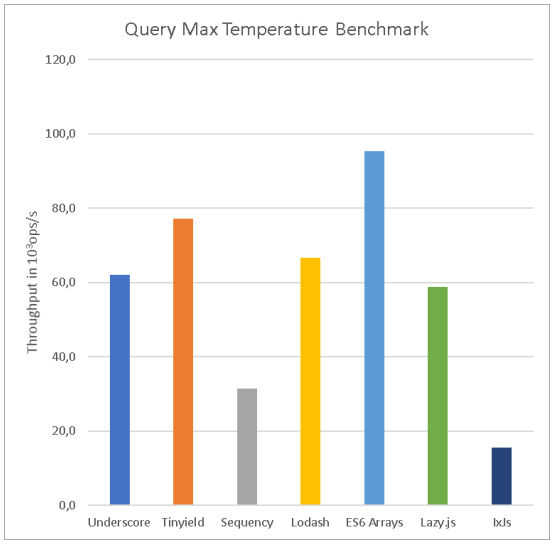

Figure 4.45: Query Max Temperature benchmark results in Javascript

4.4.7 Query Temperature Transitions

Once again, this benchmark is very similar to the ones above in sections 4.4.5 and 4.4.6, differing from them after calling our user defined operation *oddLines()*, where we call *map()* instead of *mapToInt()* to get the temperatures, and then use our user defined *collapse()* operation to filter out equal consecutive elements and finally calling *count()*, which leaves us with the number of temperature transitions. Listing 4.15 showcases the pipeline.

```
source
    .filter(s -> s.charAt(0) != '#') // Filter comments
    .skip(1) // Skip line: Not available
    .oddLines() // Filter hourly info
    .map(line -> line.substring(14, 16)) // Parse Temperature
    .collapse()
    .count();
```

Listing 4.15: Query Temperature Transitions pipeline

As we've observed so far in the two benchmarks before this, our results, shown in Figures 4.46 and 4.47, are consistent once more. We have `Tinyield` as the top performer in Java and as the second best in Typescript, behind `ES6 Arrays` only. This time however, Kotlins `Sequence` and `Eclipse Collections` both perform better than Java `Streams`, yet `StreamEx` and `Vavr` are still the least performant options. In typescript we once again have `Sequency` and `IxJs` as the least performant approaches for this use-case.

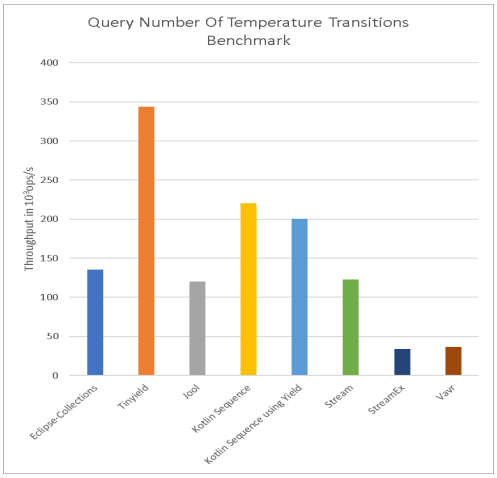

Figure 4.46: Query Temperature Transitions benchmark results in Java

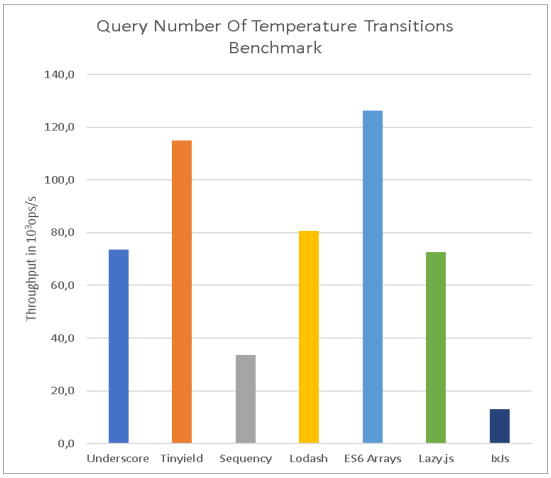

Figure 4.47: Query Temperature Transitions benchmark results in Javascript

4.5 Parallelism Benchmarks

This section presents the results for benchmarks that were run with `Stream` *parallel()* as well as the benchmarks that were devised for this specific purpose. These benchmarks were run with collection sizes from 5 all the way to 50 000 000 using a logarithmic scale, comparing sequential runs and parallel runs of Java's `Stream`.

All Match

Figure 4.48 presents the results obtained by running the All Match benchmark, comparing `Streams` sequential processing against *parallel()*. From these results we can see that parallelization becomes an advantage from 50 000 elements onward, in this scenario. This is due to the fact that the All Match benchmark is not computationally heavy, meaning the only way for `Streams` *parallel()* to become an advantage would be through the volume of data, making sequential processing a better choice for a volume of less than 50 000 elements, approximately.

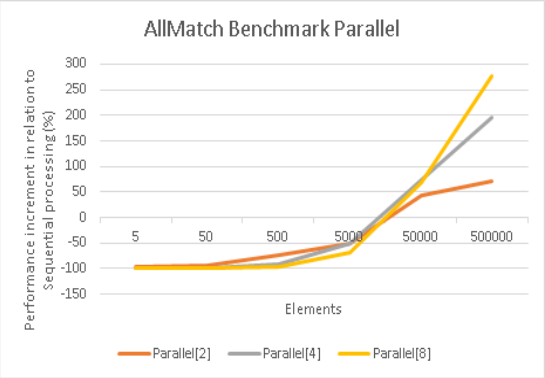

Figure 4.48: All Match benchmark results comparing sequential vs parallel processing

First

For the results on parallelization for the First benchmark with a match in the middle, Figure 4.49 shows that Java's `Stream` parallelization only becomes an advantage with more than 500 000 elements, which is our last data-point for this benchmark, and with at least 4 threads. This once again is not a computationally heavy benchmark therefore it makes sense that to take advantage of `Streams` *parallel()* we need to process more elements. The results in Figure 4.50 show much

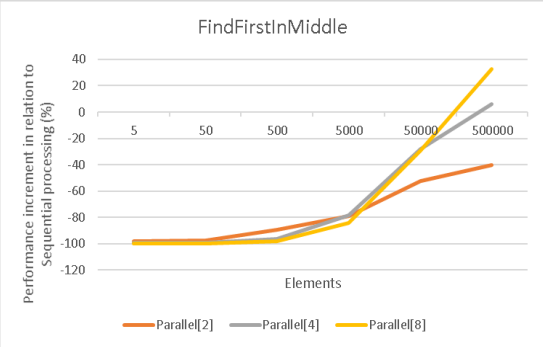

Figure 4.49: First benchmark results for match in the middle, comparing sequential vs parallel processing

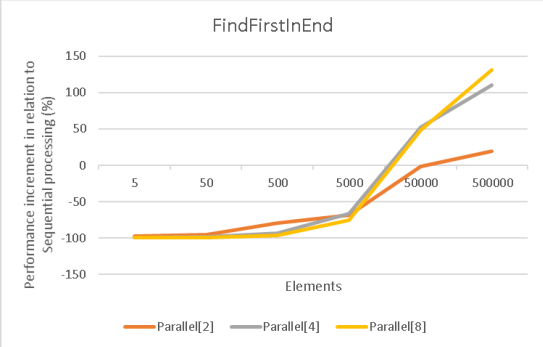

Figure 4.50: First benchmark results for match in the end, comparing sequential vs parallel processing

of the same observed in Figure 4.49 as well, with the difference that in this benchmark Java's `Stream` parallelization becomes advantageous with more than 50 000 elements and with at 2 threads or more. This makes sense due to the nature of the benchmark, as the First with the match at the end processes twice as many elements as the one with a match in the middle.

Flatmap

Finally, when it comes to the Flatmap and Reduce benchmark, gaining an advantage through parallelism, becomes advantageous after 5000 elements, in other words, 5000 sequences of sequences, with 2 or more threads. This benchmark is slightly more computationally heavy than the ones describe above and therefore parallelism is advantageous with less elements.

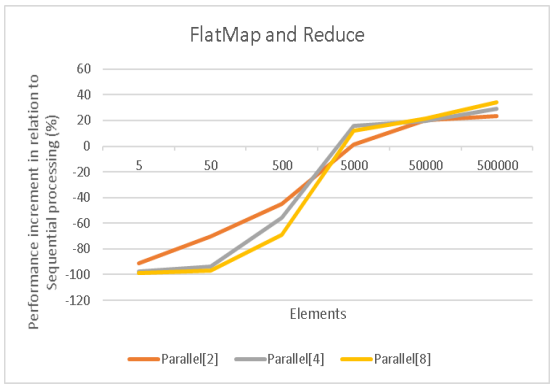

Figure 4.51: Flatmap and Reduce benchmark results, comparing sequential vs parallel processing

4.5.1 Distinct - *distinct(sequence)*

This benchmark's pipeline consists of a single operation, *distinct()*:

```
numbers.distinct();
```

Distinct returns all the elements of a sequence that occur only once in that same sequence. This benchmark was run with a sequence of all unique elements so *distinct()* would yield all of them, using the *forEach()* method and using a `Blackhole` to consume the results.

Figure 4.52 presents the results we obtained by running the Distinct benchmark. This benchmark was only run to compare parallelization performance gains, we tested it with ordered and unordered sequences yet we didn't observe any gains.

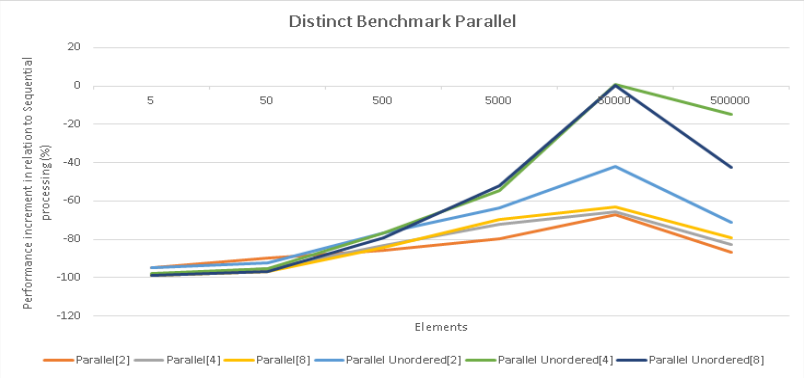

Figure 4.52: Distinct benchmark results

4.5.2 Distinct & Collect - *distinct(sequence).collect()*

This benchmark's pipeline consists of a chaining the operation *distinct()* operation with a *collect()*:

```
numbers.distinct().collect(toSet());
```

This benchmark was run with a sequence of all unique elements so *distinct()* would yield all of them.

Figure 4.53 presents the results we obtained by running the Distinct & Collect benchmark. In this case we compared performance gains of the out of the box parallel collectors and the Java library Parallel Collectors[5] yet in both cases no gains were observed over sequential processing.

[5]https://github.com/pivovarit/parallel-collectors

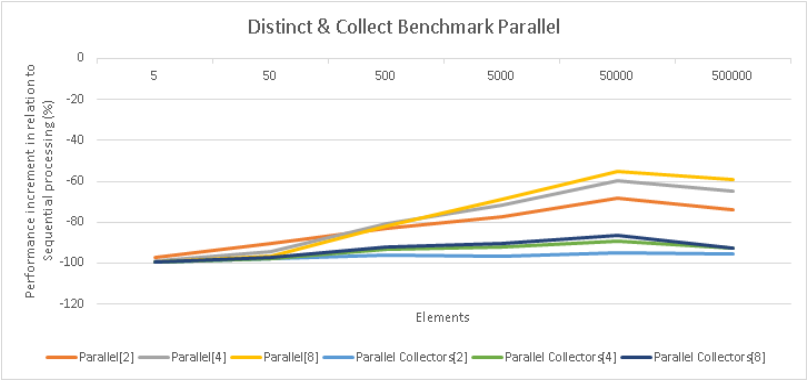

Figure 4.53: Distinct & Collect benchmark results

4.5.3 Max

This benchmark evaluates a *reduce()* operation with `Math.max()` as the *accumulator*. This benchmark was only run to evaluate performance gains with the use `Stream.parallel()`, although it was run with two different sources, namely `Arrays.stream()` and `IntStream.of()`. The following is the pipeline used by this benchmark:

```
numbers.reduce(Math::max);
```

Looking at the results obtained for this pipeline, in Figures 4.54 and 4.55, we observe that parallelization starts being an advantage after 50 000 elements with any number of threads, without much difference between each type of source.

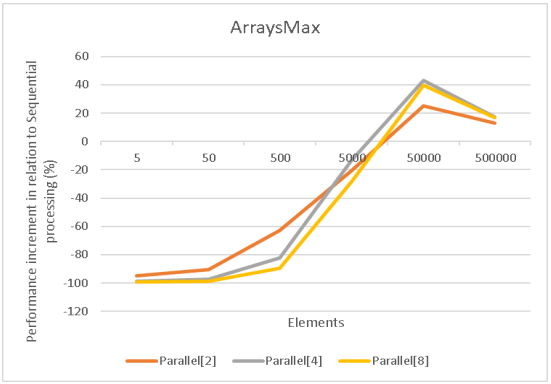

Figure 4.54: Results for Max with `Arrays.stream()`

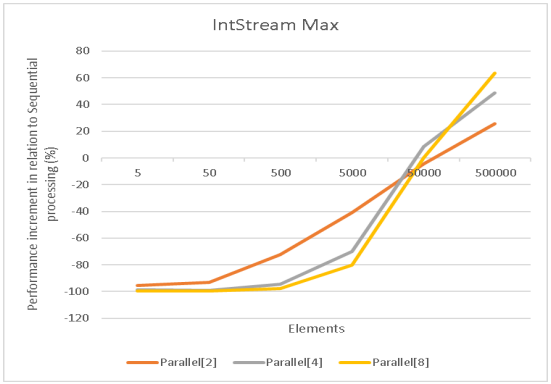

Figure 4.55: Results for Max with `IntStream.of()`

4.5.4 SlowSin

This benchmarks uses a private operation from the `FastMathCalc`[6] class in Apache Commons. To use this operation we had to make a copy of the class to our project and make it available through there. The operation is essentially calculates the sin for any number between 0 and pi/4 using Taylor's Expansion, this is a very slow operation. This benchmark was run to measure parallelization benefits only. The following snipped shows this benchmark's pipeline:

```
numbers.mapToDouble(Sine::slowSin).reduce(Math::max);
```

As we can see from Figure 4.56, this operation benefits from parallelization with very few elements, starting as low as 500 elements, this is due to the fact that it is computationally heavy, making it an excelent candidate for `Stream`.*parallel()*.

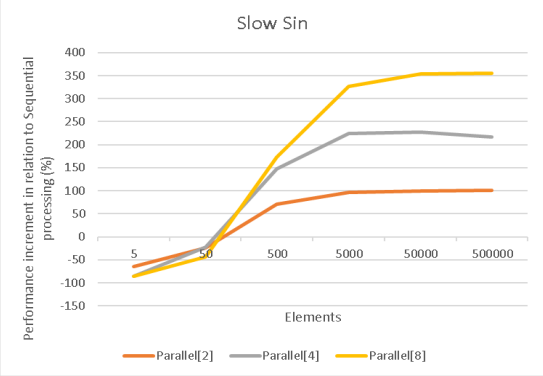

Figure 4.56: SlowSin benchmark results

4.6 Benchmark Results

In this section we'll display the results we've obtained from running the benchmarks described in this chapter. The results are presented in relation to either Java's `Stream` or `ES6 Arrays` performance (which corresponds to 1.0) also, for pipelines where collection sizes vary results are separated by a "|", like so: 1K results | 100K results.

[6]http://home.apache.org/ luc/commons-math-3.6-RC2-site/jacoco/org.apache.commons-.math3.util/FastMathCalc.java.html

Benchmark	Query Temperature Transitions	Query Max Temperature	Query Distinct Temperatures
Time Complexity	Linear	Linear	Linear
Eclipse Collections	1.1	0.5	0.5
Tinyield	**2.8**	1.1	1.1
jOOλ	1.0	1.0	1.0
Kotlin Sequence	**1.8**	0.6	0.6
Kotlin Sequence using Yield	**1.6**	0.5	0.6
StreamEx	0.3	0.2	0.2
Vavr	0.3	0.1	0.1

Table 4.4: Results for benchmarks with one input Sequence with Real World use-cases

4.6.1 Java

Benchmark	All Match	First In the Beginning	First In the Middle	First In the End	Flatmap & Reduce
Time Complexity	Linear	Constant	Linear	Linear	Linear
Eclipse Collections	**1.7 ǀ 1.8**	0.1 ǀ 0.0	**2.7 ǀ 2.0**	**3.4 ǀ 2.7**	0.6 ǀ 0.6
Tinyield	**2.6 ǀ 2.7**	1.3 ǀ 1.3	0.6 ǀ 0.6	0.6 ǀ 0.6	1.1 ǀ 1.0
Kotlin Sequence In Java	**3.3 ǀ 3.2**	**6.2 ǀ 6.5**	**3.0 ǀ 3.3**	**3.1 ǀ 3.2**	1.2 ǀ 1.3
jOOλ	1.1 ǀ 1.0	0.5 ǀ 0.5	0.9 ǀ 1.0	1.1 ǀ 1.1	0.4 ǀ 0.4
Kotlin Sequence	**2.8 ǀ 2.8**	**11.7 ǀ 12.0**	**6.2 ǀ 7.1**	**6.5 ǀ 7.1**	1.2 ǀ 1.4
StreamEx	1.0 ǀ 1.0	1.1 ǀ 1.0	0.9 ǀ 1.0	1.1 ǀ 1.1	1.1 ǀ 1.3
Vavr	0.0 ǀ 0.0	1.9 ǀ 1.9	0.1 ǀ 0.1	0.1 ǀ 0.1	0.1 ǀ 0.2

Table 4.3: Results for benchmarks with one input Sequence

Benchmark	Every Class	Every Integer	Every String	Find Class	Find Integer	Find String	Find Fixed Index	Zip Primes
Time Complexity	Linear	Linear	Linear	Linear	Linear	Linear	Constant	Linear
Eclipse Collections	**4.8 ǀ 3.1**	**4.3 ǀ 3.8**	**4.1 ǀ 1.8**	1.2 ǀ 1.4	1.4 ǀ 1.3	1.1 ǀ **1.6**	0.3 ǀ 0.2	**1.6** ǀ 1.1
Guava over Stream	1.1 ǀ 0.9	1.0 ǀ 1.1	1.1 ǀ 1.1	1.0 ǀ 1.0	1.0 ǀ 1.0	1.0 ǀ 1.0	1.1 ǀ 1.0	1.0 ǀ 1.0
Tinyield	**7.1** ǀ 1.0	**6.5** ǀ 1.3	**5.3 ǀ 4.9**	0.5 ǀ 1.2	0.6 ǀ 0.8	0.6 ǀ 1.3	0.7 ǀ 0.8	**1.7** ǀ 1.1
Kotlin Sequence In Java	**5.4 ǀ 2.0**	**5.2 ǀ 3.9**	**5.0 ǀ3.0**	**1.6** ǀ 1.3	**4.2 ǀ 2.0**	**2.8** ǀ 1.3	**3.5 ǀ 3.3**	**1.5** ǀ 1.1
jOOλ	0.7 ǀ 0.8	0.8 ǀ 0.8	0.8 ǀ 0.8	0.7 ǀ 0.9	0.7 ǀ 0.9	0.8 ǀ 0.9	0.6 ǀ 0.7	1.0 ǀ 1.0
Kotlin Sequence	**5.4 ǀ 2.1**	**5.2 ǀ 4.0**	**4.4 ǀ 2.8**	**1.6** ǀ 1.0	**4.6 ǀ 2.2**	**2.8** ǀ 1.4	**3.5 ǀ 3.1**	1.0 ǀ 0.9
Protonpack over Stream	1.3 ǀ 1.2	1.3 ǀ 1.2	1.3 ǀ **2.0**	1.1 ǀ 1.0	0.9 ǀ 1.0	0.8 ǀ 1.0	1.4 ǀ 1.0	1.1 ǀ 1.0
StreamEx	1.4 ǀ 0.9	1.4 ǀ 1.3	1.4 ǀ 1.3	1.2 ǀ 1.0	1.2 ǀ 1.1	1.2 ǀ 1.0	1.3 ǀ 1.3	1.0 ǀ 1.1
Vavr	0.1 ǀ 0.1	0.1 ǀ 0.1	0.1 ǀ 0.1	0.1 ǀ 0.6	0.1 ǀ 0.4	0.1 ǀ 0.5	0.2 ǀ 0.1	0.2 ǀ 0.7
Zipline	1.3 ǀ 1.0	1.3 ǀ 1.2	1.3 ǀ 1.2	1.1 ǀ 1.0	1.1 ǀ 1.1	1.1 ǀ 1.0	1.2 ǀ 1.1	1.3 ǀ 1.0

Table 4.5: Results for benchmarks with two input Sequences

Benchmark	Distinct Top Artist and Top Track by Country Benchmark	Artists who are in a Country's top ten who also have Tracks in the same Country's top ten Benchmark
Time Complexity	Linear	Linear
Eclipse Collections	1.2	1.2
Guava over Stream	1.0	1.0
Tinyield	**1.5**	1.1
Kotlin Sequence In Java	**2.1**	1.1
jOOλ	0.9	0.8
Kotlin Sequence	**2.1**	1.1
Protonpack over Stream	1.1	1.0
StreamEx	1.1	0.9
Vavr	0.8	0.4
Zipline	1.0	1.0

Table 4.6: Results for benchmarks with two input Sequences with Real World use-cases

Table 4.3 shows top performance by Kotlin's `Sequence` for almost every benchmark being the only one that managed to have performance above Java's `Stream` on all benchmarks. Table 4.4 on the other hand shows top performance by `Tinyield's Query` every benchmark with Kotlin droping below Java's `Stream` on two of the benchmarks. Table 4.5, which is where the *zip* use case is benchmarked, shows a more scattered scenario as the top performant spot is distributed between `Eclipse Collections`, Kotlin's `Sequence` and `Tinyield's Query` over the different benchmarks. Finally, in Table 4.6 shows more of the same with `Eclipse Collections` being the top performer in one of the benchmarks and Kotlin's `Sequence` in the other with `Tinyield's Query` close behind.

From our investigation about Kotlin's performance over Java's `Stream`, we identified a few advantages that Kotlin has, namely operations that in Java would return `Optional`, return *nullable* in Kotlin, meaning no wrapper is created resulting in less overhead, not only that but Kotlin's terminal operations are inline so there is no indirection when calling terminal operations.

For `Tinyield`, we attribute the performance gains to our fast-path iteration protocol that has less overhead when bulk traversing a sequence than a `Stream` does. Not only that but `Tinyield` is the only option with a verboseless way of defining new operations while also maintaining the fluency of the pipeline.

`Eclipse Collections` has a lot of optimizations in place with regards to the data-source of the pipeline, namely if an array was at the source then iteration will be as fast as using a *for* loop, on the other hand it performs worse in short-circuiting operations due to it processing every operation in bulk, meaning that for a pipeline consisting of *source.filter(...).findFirst()* `Eclipse Collections` will

first calculate the result of *source.filter(...)* and then access the first element of the resulting sequence, this leads to a lot of unnecessary processing and it shows in the *First In the Beginning* and *Find Fixed Index* benchmarks.

`StreamEx` has some performance gains over Java's `Stream` but, like `jOOλ`, `Vavr`, `Guava` and `Protonpack` the main gain is the fact that these libraries bring extra functionality to the user out of the box.

4.6.2 Java's `Stream` Parallel

Benchmark	Time Complexity	Threads	5 Elements	50 Elements	500 Elements	5000 Elements	50000 Elements	500000 Elements
All Match	Linear	2	-1.0	-0.9	-0.7	-0.5	**0.4**	**0.7**
		4	-1.0	-1.0	-0.9	-0.5	**0.7**	**1.9**
		8	-1.0	-1.0	-1.0	-0.7	**0.7**	**2.8**
Distinct	Linear	2	-0.9	-0.9	-0.9	-0.8	-0.7	-0.9
		4	-1.0	-1.0	-0.8	-0.7	-0.7	-0.8
		8	-1.0	-1.0	-0.8	-0.7	-0.6	-0.8
Distinct & Collect	Linear	2	-1.0	-0.9	-0.8	-0.8	-0.7	-0.7
		4	-1.0	-0.9	-0.8	-0.7	-0.6	-0.6
		8	-1.0	-1.0	-0.8	-0.7	-0.6	-0.6
First In the Middle	Linear	2	-1.0	-1.0	-0.9	-0.8	-0.5	-0.4
		4	-1.0	-1.0	-1.0	-0.8	-0.3	**0.1**
		8	-1.0	-1.0	-1.0	-0.8	-0.3	**0.3**
First In the End	Linear	2	-1.0	-0.9	-0.8	-0.7	0.0	**0.2**
		4	-1.0	-1.0	-0.9	-0.7	**0.5**	**1.1**
		8	-1.0	-1.0	-1.0	-0.8	**0.5**	**1.3**
Flatmap & Reduce	Linear	2	-0.9	-0.7	-0.5	0.0	**0.2**	**0.2**
		4	-1.0	-0.9	-0.6	**0.2**	**0.2**	**0.3**
		8	-1.0	-1.0	-0.7	**0.1**	**0.2**	**0.3**
Arrays Max	Linear	2	-0.9	-0.9	-0.6	-0.2	**0.3**	**0.1**
		4	-1.0	-1.0	-0.8	-0.1	**0.4**	**0.2**
		8	-1.0	-1.0	-0.9	-0.3	**0.4**	**0.2**
IntStream Max	Linear	2	-1.0	-0.9	-0.7	-0.4	0.0	**0.3**
		4	-1.0	-1.0	-0.9	-0.7	**0.1**	**0.5**
		8	-1.0	-1.0	-1.0	-0.8	0.0	**0.6**
Slow Sin	Linear	2	-0.6	-0.2	**0.7**	**1.0**	**1.0**	**1.0**
		4	-0.9	-0.2	**1.5**	**2.2**	**2.3**	**2.2**
		8	-0.9	-0.4	**1.7**	**3.3**	**3.5**	**3.6**

Table 4.7: Results for Java's `Stream` Parallel benchmarks

Table 4.7 shows the benchmark results for performance gain of using `Stream`'s *parallel()* feature over sequential processing of the same `Stream`. These benchmarks clearly show that `Stream`.*parallel()* is not always advantageous, it becomes advantageous the more elements we are processing, the more computationally heavy processing an element is and the more threads we have, although it also seems to have a threshold from which point it start to be an advantage.

As stated by Brian Goetz, the head of Java Streams project, in his article "Optimize stream pipelines for parallel processing" [6], namely in the section about "The NQ Model", where N is the number of data elements, and Q is the amount of work

performed per element. This section describes how we are more likely to see performance gains due to parallelization the larger the product of N*Q, stating that to see any significant speedups the relation must be N*Q > 10000. With that said we didn't observe this relation in pipelines where the computational cost per element was too low, in other words, although it is true that N*Q > 10000, either the threshold is too low for some pipelines or we are missing some minimum values for N and for Q.

In regards to the number of threads, we did not see the speedup occurring with lower values of N*Q with the increment of threads, we did however see a larger speedup the more threads we had working on the pipeline.

4.6.3 Javascript / Typescript

Benchmark	All Match	First In the Beginning	First In the Middle	First In the End	Flatmap & Reduce
Time Complexity	Linear	Constant	Linear	Linear	Linear
Underscore	0.8 | **1.8**	0.2 | 1.1	0.2 | 1.1	0.2 | 1.1	0.8 | 0.9
Tinyield	0.6 | **1.7**	**66.0 | 34773.0**	0.6 | **3.4**	0.3 | **1.7**	**8.5 | 2.2**
Sequency	0.7 | 0.9	**79.4 | 46235.6**	0.3 | **2.0**	0.2 | 1.0	**4.2 | 1.5**
Lodash	0.9 | **1.9**	0.3 | **2.0**	0.3 | **1.9**	0.3 | **1.9**	**3.8 | 1.6**
Lazy.js	0.8 | 0.8	**240.4 | 158175.9**	0.8 | **4.5**	0.4 | 2.1	**4.8 | 3.9**
IxJs	0.2 | 0.4	1.1 | **3.3**	0.1 | 0.8	0.1 | 0.4	1.1 | 0.8

Table 4.8: Results for benchmarks with one input Sequence

Benchmark	Query Temperature Transitions	Query Max Temperature	Query Distinct Temperatures
Time Complexity	Linear	Linear	Linear
Underscore	0.5	0.6	0.6
Tinyield	0.7	0.8	0.8
Sequency	0.3	0.4	0.5
Lodash	0.5	0.6	0.7
Lazy.js	0.6	0.6	1.0
IxJs	0.1	0.2	0.2

Table 4.9: Results for benchmarks with one input Sequence with Real World use-cases

Benchmark	Every Class	Every Integer	Every String	Find Class	Find Integer	Find String	Find Fixed Index	Zip Primes
Time Complexity	Linear	Linear	Linear	Linear	Linear	Linear	Constant	Linear
Underscore	0.3 | 0.6	0.3 | 0.5	0.3 | 0.6	0.3 | 0.5	0.2 | 0.5	0.3 | 0.5	0.2 | 0.5	0.3 | 0.2
Tinyield	1.0 | **5.2**	1.2 | **7.3**	0.8 | **4.4**	**1.7 | 9.2**	**1.5 | 8.9**	**2.0 | 8.7**	**75.0 | 662.3**	1.1 | 1.1
Sequency	0.4 | **2.3**	0.4 | **2.8**	0.4 | **2.4**	1.0 | **4.6**	0.6 | **4.5**	1.0 | **4.8**	**25.5 | 294.0**	0.7 | 1.0
Lodash	0.7 | 1.1	0.8 | 1.1	0.6 | 1.0	0.7 | 0.8	0.6 | 0.9	0.7 | 0.8	0.6 | 0.9	0.5 | 0.3
Lazy.js	0.7 | **3.7**	0.8 | **3.5**	0.7 | **3.6**	**2.0 | 7.1**	1.3 | **8.5**	**1.9 | 9.3**	**40.9 | 331.3**	0.6 | 0.9
IxJs	0.2 | 0.7	0.1 | 0.8	0.2 | 0.7	0.4 | **5.1**	0.3 | **5.9**	0.4 | **6.3**	**3.1 | 10.1**	0.4 | 0.6

Table 4.10: Results for benchmarks with two input Sequences

Benchmark	Distinct Top Artist and Top Track by Country Benchmark	Artists who are in a Country's top ten who also have Tracks in the same Country's top ten Benchmark
Time Complexity	Linear	Linear
Underscore	0.1	0.2
Tinyield	0.8	0.8
Sequency	0.8	0.4
Lodash	0.2	0.2
Lazy.js	0.9	1.2
IxJs	0.3	0.2

Table 4.11: Results for benchmarks with two input Sequences with Real World use-cases

This analysis is quite different as the out of the box solution `ES6 Arrays` seem to outperform all sequence alternatives in virtually every benchmark as we can observe in the Tables 4.8, 4.9, 4.10 and 4.11. The exceptions here are on pipelines that require short-circuiting, as `ES6 Arrays` are *eager* it calculates all intermediate results before continuing to the next operation of the pipeline, this again leads to a lot of unnecessary processing as we can observe in benchmarks such as *Find* with any data source for 100 000 elements, but specially in *Find Fixed Index* seeing as the match index is made in either the 100 position or the 10 000 it means `ES6 Arrays` will process 90 000 elements unnecessarily.

From the remaining alternatives `Lazy.js` is the top performer with `Tinyield's Query` and `Sequency` close behind. `IxJs`'s performance can be justified by the fact that not only does it only traverse elements individually but with each iteration it creates a wrapper object that represents the state of the traversal, namely if it is done or not. `Lodash` and `Underscore`, although they are quite popular in the Javascript world, they suffer from some of the same problems as `ES6 Arrays`, in other words, when processing a sequence pipeline these libraries will calculate all intermediate results before proceeding to the next operation, which incurs in the same unnecessary processing observed in `ES6 Arrays`.

5

Conclusions

Java `Streams` are a masterpiece of software engineering, allowing querying and traversal of sequences, sequentially or in parallel. Nevertheless, it does not contain, nor could it, every single operation that will be needed for every use case. The same is true for most out of the box sequence type implementations in other programming languages. But whereas most environments provide *API* extensibility through the yield generators feature, Java uses an object-oriented approach that incurs in verbosity and performance overhead.

In this document, we highlight concerns regarding the choice of a workaround to suppress the absence of any operation in a sequence implementation. As well as the complexity added by some sequence designs.

We brought to our analysis options, such as Kotlin and the `Tinyield` library for the JVM as well as `Lodash` and `Lazy.js` for Javascript, and we also built benchmarks based on a realistic data sources to fairly compare all alternatives. These benchmarks are available on Github and their workflows are publicly visible in Github Actions. The results and analysis are also available in our published articles in DZone [11] and InfoQ [12].

We expect that all this research, evaluated characteristics and our proposals can serve as a basis to help developers make their choice of a sequence type.

5.1 Main Contributions

The research and development done in this dissertation aimed to better understand the state of the art in sequence design, how the performance was measured and how did the alternatives fare. It showcases how a minimalist implementation of a sequence type, on any programming language, can accommodate most use-cases needed by programmers while also being an efficient solution. And finally, understand how and when parallelism becomes an advantage. With that said, these are the main contributions of this work:

- State of the art on different sequences design proposals
- Benchmarks to assess sequences performance for both JVM and Javascript
- Minimalist yet efficient design for sequences traversal
- Analysis of the effectiveness of parallel processing for sequences workloads

5.1.1 State of the art on different sequences design proposals

What have `Iterable` and Java `Streams` in common? Or `Vavr` and `jOOλ`? Are they completely different? Or are they similar alternatives? To answer these questions, we identified the main features provided by sequences and grouped them in categories to make a consistent classification among different designs, features such as verbosity of extension, or fluency when using a user-defined operations. We can use this classification to make a fair and unbiased comparison between sequences. This analysis was described and presented in the article titled: *"Bridge the Gap of Zip Operation"*, published by *DZone* in May 2020 [11]. In this article we state that no library may cover all the use-cases a programmer may require. Furthermore, we analyzed each alternatives approach to extend their *APIs* with custom operations and evaluated the performance of each solution against realistic workloads.

5.1.2 Benchmarks to assess sequences performance for both JVM and Javascript

How can we compare those libraries in performance? This work devised a set of benchmarks, both with generated data sets, as well as real world data, using *APIs* such as REST Countries[1], Last.fm[2] and World Weather Online[3]. These benchmarks were used to compare the state of the art alternatives for those developing with the *JVM* as well as those in the *Javascript* world. These comparisons led us to investigade many alternatives, such as `IxJS`, `Lodash` and `Sequency` in Javascript and Typescript, as well as `StreamEx`, `jOOλ` and Kotlins `Sequence` in Java.

The use of Kotlins `Sequence` in the Java ecosystem was especially interesting due to Kotlin being a different programming language with integration with the JVM. Not only that but the performance that `Sequence` presented in the benchmarks we devised was quite astounding. However, using `Sequence` in Java would cause a loss in pipeline fluency, seeing as all `Sequences` methods are added to Java through static methods, which are not chainable. This led us to create `Sek`[4], a Java wrapper for Kotlins `Sequence`, in order to have `Sequences` performance and full set of operations while still maintaining pipeline fluency even in Java. We presented and discussed all these topics in an article published in InfoQ titled: *"Enhanced Streams Processing with Kotlin's Sequence Interface"* [12], adding a performance evaluation through the benchmarks mentioned above as well.

These benchmarks are publicly available in two Github repositories:

- github.com/tinyield/sequences-benchmarks
- github.com/tinyield/ts-sequences-benchmarks

Not only that but we added workflows to those repositories in order to have Github Actions perform the benchmarks automatically. The results are publicly available in these repositories, for the community to see and use when comparing sequence types.

[1] https://restcountries.eu/
[2] https://www.last.fm/api/
[3] https://www.worldweatheronline.com/developer/
[4] https://github.com/tinyield/sek

5.1.3 Minimalist yet efficient design for sequences traversal

Despite the indisputable and ingenious design of Java `Streams` they are also complex and difficult to dive into its implementation. Comparing to Java `Collections` and `Iterator` implementations where you may simply navigate to the `ArrayList` *forEach()* implementation and find a clear

```
for (int i = 0; ...  && i < size; i++)  action.accept(elementAt(es, i));
```

Navigating to the `Stream` *forEach()* implementation, the reader will jump to a `ReferencePipeline`, `AbstractPipeline`, `TerminalOp`, `Sink`, etc. Moreover, all this complexity is not directly related with parallel processing requirements, since other alternatives such as .NET `ParallelQuery` have similar issues and chose a different approach, inheriting from the existent `IEnumerable`, equivalent to Java `Iterable`.

Java `Streams` complexity is not related with their querying and operation chaining capabilities either, which could be easily implemented over `Iterable`, as has been done by some libraries such as `Guava` and `Eclipse Collections`.

Thus, we claim that a sequence design should preserve both transparency and readability. And, Java `Streams` masterpiece could give the biased idea that it was the only way to achieve a sequences pipeline design. To that end, we have shown that if we discard the parallel processing requirements we may achieve a simpler design that is still efficient in performance. Furthermore, our proposal has several advantages over Java `Streams` in extensibility, which may be a more realistic need than parallelism, as we show in the Subsection 5.1.4.

Finally, our proposal shows that we can achieve an effective way of sequence extensibility with similar semantics to the yield generator operator provided by many programming languages. The lack of yield in Java may be simply suppressed with our proposal at the *API* design level.

These ideas have been synthesized in my latest article *"Deconstructing yield operator to enhance streams processing"* accepted in the *"16th International Conference on Software Technologies"*.

5.1.4 Analysis of the effectiveness of parallel processing for sequences workloads

There is a lot of performance advice around the advantages of sequences parallelization, ranging from questions in Stackoverflow[16], to full blown articles[6],

but little of it points to real performance considerations in benchmarks. To this end, we devised a set of benchmarks that would showcase at which point `Streams` *parallel()* becomes a performance advantage. We used the research already available online to see if the conclusions matched what our benchmarks showed, which wasn't always the case.

These benchmarks are also publicly available in our Github repository github.com/tinyield/parallel-sequences-benchmarks where we also added Github Action workflows in order to have Github perform the benchmarks automatically. The results are also publicly available in this repository, for the community to see and use when considering if parallelism is the solution to the performance problems they may be encountering.

5.2 Future Work

As we already mentioned in chapter 3 there are a few drawbacks in our solution that could use some work, such as the extensibility being made through the fast-path iteration protocol and therefore having no direct translation to element by element iteration unless the overload is used. One solution to this problem would be to generate the code for the `Advancer` at runtime, when the use-case requires element by element traversal. This could be achieved, assuming that in the source `Traverser` one of the following is true:

- A call to *upstream.traverse(...)* is made.
- A call to *yield.ret(...)* is made.

Assuming any of these are true, we could generate an `Advancer` doing the following:

- Instead of calling *upstream.traverse(...)* we would need to call *upstream.tryAdvance(...)*.
- When calling yield.ret(...) we would register that an element has been found that meets the criteria.
- The call to *upstream.tryAdvance(...)* is made like so: *while(noElementFound && upstream.tryAdvance(...))*

```
1 <T, R> Traverser<R> map(Query<T> upstream, Function<T, R> mapper) {
2   return yield -> {
3      upstream.traverse(e -> yield.ret(mapper.apply(e)));
4   };
5 }
```

Listing 5.1: `Traverser` definition

So, given the `Traverser` definition in Listing 5.1 for a *map* operation in bulk, the element by element traversal could be inferred as the one presented in Listing 5.2.

```
<T, R> Advancer<R> map(Query<T> upstream, Function<T, R> mapper) {
  @boolean[] noElementFound = new boolean[] {true};@
  return yield -> {
    @noElementFound[0] = true;@
    while(noElementFound[0] && upstream.tryAdvance(e -> {
       yield.ret(mapper.apply(e)));
       @noElementFound[0] = false;@
     });
    @return noElementFound[0] == false;@
  };
}
```

Listing 5.2: Inferred *Advancer* from Listing 5.1

With that said, there are corner cases that are not so easily mapped, as is the case of the *flatMap()* operation which traverses over multiple sequences, these cases need a different approach. Nevertheless, with this approach the code for the `Advancer` could be generated using a library for *bytecode* manipulation at *runtime* such as ASM[2].

Another thing to consider is the way operations are short-circuited. Operations could take into account the type of sequence pipeline they have been called upon and throw an `Error` only when necessary.

Not only that but, more benchmarks could be added to our public Github repositories, to better help developers who want to compare the state of the art alternatives and make a more informed decision when picking a sequence type.

Finally, we could also expand the `Tinyield` solution to other programming languages such as C# or Python, as we have done with Javascript/Typescript (see tinyield4ts[5] in Github) and Java, and make the same sort of comparisons between the solutions of each programming language.

[5]https://github.com/tinyield/tinyield4ts

Bibliography

[1] Henry G. Baker. Iterators: Signs of weakness in object-oriented languages. *SIGPLAN OOPS Mess.*, 4(3):18–25, July 1993. ISSN 1055-6400. doi: 10.1145/165507.165514. URL http://doi.acm.org/10.1145/165507.165514. (pp. 7 and 12)

[2] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002. (p. 103)

[3] Mathias Bynens and John-David Dalton. Bulletproof javascript benchmarks. 2010. URL https://calendar.perfplanet.com/2010/bulletproof-javascript-benchmarks/. (pp. 56 and 57)

[4] Martin Fowler. Collection pipeline. June 2015. URL https://martinfowler.com/articles/collection-pipeline/. (pp. 20 and 45)

[5] Daniel P. Friedman and David S. Wise. CONS should not evaluate its arguments. In S. Michaelson and Robin Milner, editors, *Automata, Languages and Programming*, pages 257–284, Edinburgh, Scotland, 1976. Edinburgh University Press. (p. 11)

[6] Brian Goetz. Optimize stream pipelines for parallel processing. 2016. URL https://developer.ibm.com/technologies/java/articles/j-java-streams-5-brian-goetz/#thenqmodel. (pp. 93 and 100)

[7] P. J. Landin. Correspondence between algol 60 and church's lambda-notation: Part i. *Commun. ACM*, 8(2):89–101, February 1965. ISSN 0001-0782. doi: 10.1145/363744.363749. URL http://doi.acm.org/10.1145/363744.363749. (p. 11)

[8] Angelika Langer and Klaus Kreft. Stream performance. JAX London Online Conference, 2015. URL https://jaxlondon.com/wp-content/uploads/2015/10/The-Performance-Model-of-Streams-in-Java-8-Angelika-Langer-1.pdf. (p. 12)

[9] Barbara Liskov. *CLU Reference Manual*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1983. ISBN 038710836X. (pp. 1 and 5)

[10] Nicolai Parlog. Github, 2019. URL https://github.com/CodeFX-org/benchmarks#parallel-stream-vectorization. (p. 13)

[11] Diogo Poeira and Miguel Gamboa. Bridge the gap of zip operation. *DZone, Java Zone*, 2020. URL https://dzone.com/articles/bridge-the-gap-of-zip-operation. (pp. 97 and 98)

[12] Diogo Poeira and Miguel Gamboa. Enhanced streams processing with kotlin's sequence interface. *InfoQ*, 2021. URL https://www.infoq.com/articles/enhanced-stream-kotlin-sequence/. (pp. 97 and 99)

[13] Ilya Ryzhenkov. JetBrains, 2014. URL https://github.com/Kotlin/kotlin-benchmarks. (pp. 12 and 57)

[14] Aleksey Shipilev. Java microbenchmark harness (the lesser of two evils), 2013. URL https://openjdk.java.net/projects/code-tools/jmh/. (p. 56)

[15] Guy Steele. *Common LISP: the language*. Elsevier, 1990. (p. 60)

[16] Mats Krüger Svensson. StackOverflow, 2013. URL https://stackoverflow.com/questions/20375176/should-i-always-use-a-parallel-stream-when-possible. (p. 100)

[17] Guido van Rossum and Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2011. ISBN 1906966141, 9781906966140. (p. 11)

[18] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, January 1988. ISSN 0304-3975.

doi: 10.1016/0304-3975(90)90147-A. URL http://dx.doi.org/10.1016/0304-3975(90)90147-A. (p. 15)

[19] Ka-Ping Yee and Guido van Rossum. Pep 234 – iterators. Technical report, Python, 2001. URL https://www.python.org/dev/peps/pep-0234/. (p. 12)