



Parallel Streams

Objectives

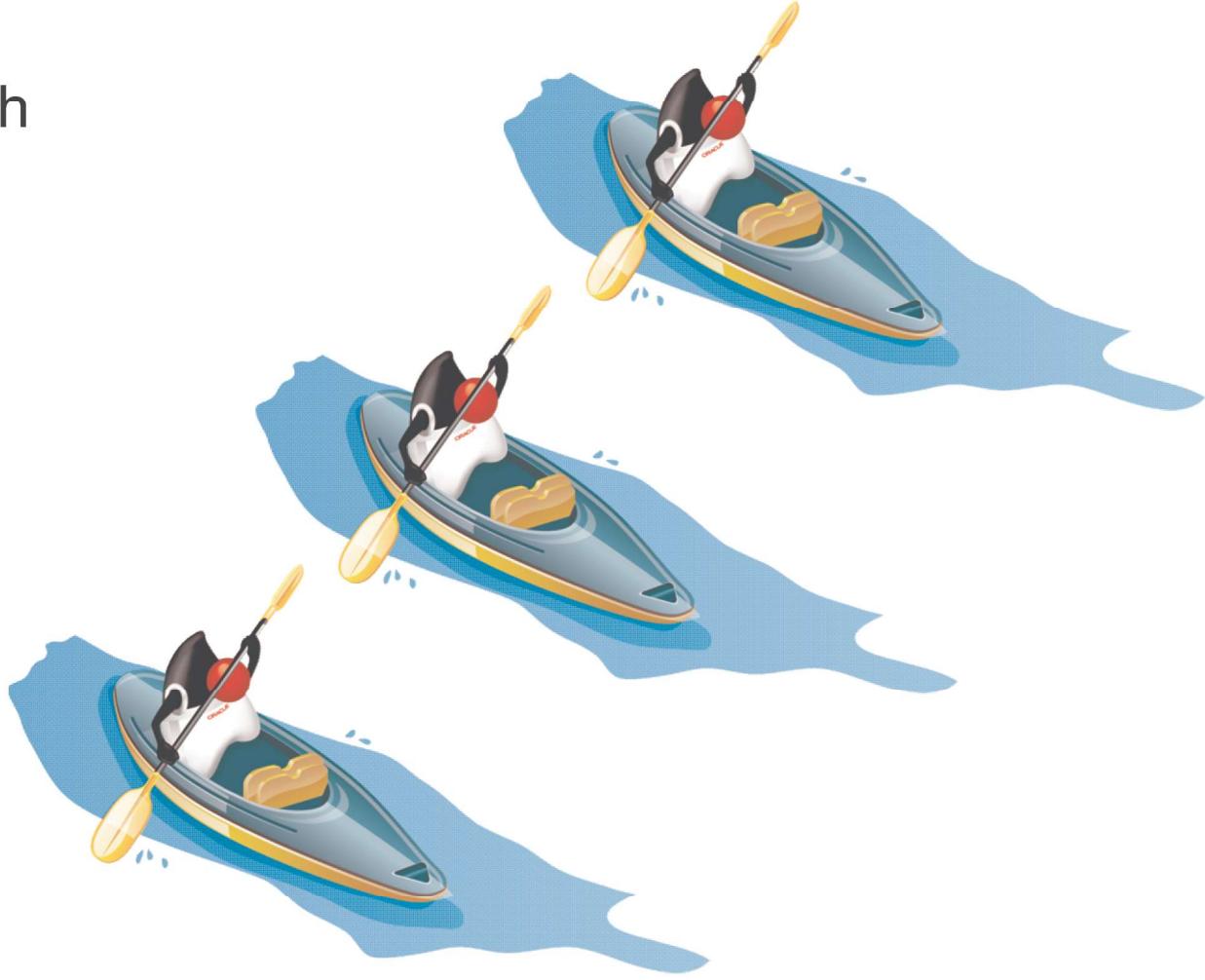
After completing this lesson, you should be able to do the following:

- Review the key characteristics of streams
- Contrast old style loop operations with streams
- Describe how to make a stream pipeline execute in parallel
- List the key assumptions needed to use a parallel pipeline
- Describe why reduction requires an associative function
- Calculate a value using reduce
- Describe the process for decomposing and then merging work
- List the key performance considerations for parallel streams



Streams

- Iterate through collections using `forEach`
- Streams and Filters



Streams Review

- Pipeline
 - Multiple streams passing data along
 - Intermediate, Terminal, and Short-Circuit Terminal Operations
- Stream characteristics
 - Immutable
 - Once elements are consumed they are no longer available from the stream.
 - Can be sequential (default) or **parallel**

Old Style Collection Processing

```
15     double sum = 0;
16
17     for (Employee e:eList) {
18         if(e.getState().equals("CO") &&
19             e.getRole().equals(Role.EXECUTIVE)) {
20             e.printSummary();
21             sum += e.getSalary();
22         }
23     }
24
25     System.out.printf("Total CO Executive Pay: $%,9.2f %n",
sum);
```

New Style Collection Processing

```
15     double result = eList.stream()
16         .filter(e -> e.getState().equals("CO"))
17         .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18         .peek(e -> e.printSummary())
19         .mapToDouble(e -> e.getSalary())
20         .sum();
21
22     System.out.printf("Total CO Executive Pay: $%,9.2f %n",
result);
```

- What are the advantages?
 - Code reads like a problem.
 - Acts on the data set
 - Operations can be lazy.
 - Operations can be serial or parallel.

Stream Pipeline: Another Look

```
13     public static void main(String[] args) {  
14  
15         List<Employee> eList = Employee.createShortList();  
16  
17         Stream<Employee> s1 = eList.stream();  
18  
19         Stream<Employee> s2 = s1.filter(  
20             e -> e.getState().equals("CO"));  
21  
22         Stream<Employee> s3 = s2.filter(  
23             e -> e.getRole().equals(Role.EXECUTIVE));  
24         Stream<Employee> s4 = s3.peek(e -> e.printSummary());  
25         DoubleStream s5 = s4.mapToDouble(e -> e.getSalary());  
26         double result = s5.sum();  
27  
28         System.out.printf("Total CO Executive Pay: $%,9.2f %n",  
29             result);  
30     }
```

Styles Compared

Imperative Programming

- Code deals with individual data items.
- Focused on how
- Code does not read like a problem.
- Steps mashed together
- Leaks extraneous details
- Inherently sequential

Streams

- Code deals with data set.
- Focused on what
- Code reads like a problem.
- Well-factored
- No "garbage variables" (Temp variables leaked into scope)
- Code can be sequential or parallel.

Parallel Stream

- May provide better performance
 - Many chips and cores per machine
 - GPUs
- Map/Reduce in the small
- Fork/join is great, but too low level
 - A lot of boilerplate code
 - Stream uses fork/join under the hood
- Many factors affect performance
 - Data size, decomposition, packing, number of cores
- Unfortunately, not a magic bullet
 - Parallel is not always faster

Using Parallel Streams: Collection

➤ Call from a Collection

```
15      double result = eList.parallelStream()
16          .filter(e -> e.getState().equals("CO"))
17          .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18          .peek(e -> e.printSummary())
19          .mapToDouble(e -> e.getSalary())
20          .sum();
21
22      System.out.printf("Total CO Executive Pay: $%,9.2f
%n", result);
```

Using Parallel Streams: From a Stream

```
27     result = eList.stream()
28         .filter(e -> e.getState().equals("CO"))
29         .filter(e -> e.getRole().equals(Role.EXECUTIVE))
30         .peek(e -> e.printSummary())
31         .mapToDouble(e -> e.getSalary())
32         .parallel()
33         .sum();
34
35     System.out.printf("Total CO Executive Pay: $%,9.2f
%n", result);
```

- Specify with `.parallel` or `.sequential` (**default is sequential**)
- Choice applies to entire pipeline.
 - Last call wins
- Once again, the API doc is your friend.

Pipelines Fine Print

- Stream pipelines are like Builders.
 - Add a bunch of intermediate operations, and then execute
 - Cannot "branch" or "reuse" pipeline
- Do not modify the source during a query.
- Operation parameters must be stateless.
 - Do not access any state that might change.
 - **This enables correct operation sequentially or in parallel.**
- Best to banish side effects completely.

Embrace Statelessness

```
17  List<Employee> newList02 = new ArrayList<>();  
...  
23  newList02 = eList.parallelStream() // Good Parallel  
24      .filter(e -> e.getDept().equals("Eng"))  
25      .collect(Collectors.toList());
```

- Mutate the stateless way
 - The above is preferable.
 - It is designed to parallelize.

Avoid Statefulness

```
15      List<Employee> eList = Employee.createShortList();  
16      List<Employee> newList01 = new ArrayList<>();  
17      List<Employee> newList02 = new ArrayList<>();  
18  
19      eList.parallelStream() // Not Parallel. Bad.  
20          .filter(e -> e.getDept().equals("Eng"))  
21          .forEach(e -> newList01.add(e));
```

- Temptation is to do the above.
 - Do not do this. It does not parallelize.

Streams Are Deterministic for Most Part

```
14     List<Employee> eList = Employee.createShortList();
15
16     double r1 = eList.stream()
17         .filter(e -> e.getState().equals("CO"))
18         .mapToDouble(Employee::getSalary)
19         .sequential().sum();
20
21     double r2 = eList.stream()
22         .filter(e -> e.getState().equals("CO"))
23         .mapToDouble(Employee::getSalary)
24         .parallel().sum();
25
26     System.out.println("The same: " + (r1 == r2));
```

- Will the result be the same?

Some Are Not Deterministic

```
14     List<Employee> eList = Employee.createShortList();  
15  
16     Optional<Employee> e1 = eList.stream()  
17         .filter(e -> e.getRole().equals(Role.EXECUTIVE))  
18         .sequential().findAny();  
19  
20     Optional<Employee> e2 = eList.stream()  
21         .filter(e -> e.getRole().equals(Role.EXECUTIVE))  
22         .parallel().findAny();  
23  
24     System.out.println("The same: " +  
25         e1.get().getEmail().equals(e2.get().getEmail()));
```

- Will the result be the same?
 - In this case, maybe not.

Reduction

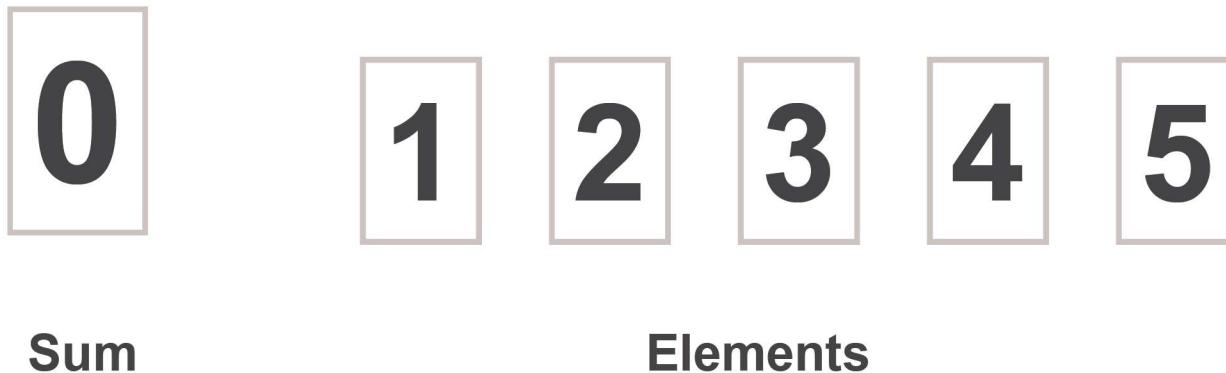
- Reduction
 - An operation that takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation.
 - Implemented with the `reduce()` method
- Example: `sum` is a reduction with a base value of 0 and a combining function of `+`.
 - $((((0 + a_1) + a_2) + \dots) + a_n)$
 - `.sum()` is equivalent to `reduce(0, (a, b) -> a + b)`
 - `(0, (sum, element) -> sum + element)`

Reduction Fine Print

- If the combining function is associative, reduction parallelizes cleanly
 - Associative means the order does not matter.
 - The result is the same irrespective of the order used to combine elements.
- Examples of: sum, min, max, average, count
 - `.count()` is equivalent to `.map(e -> 1).sum()`.
- **Warning:** If you pass a nonassociative function to `reduce`, you will get the wrong answer. The function must be associative.

Reduction: Example

```
18     int r2 = IntStream.rangeClosed(1, 5).parallel()  
19         .reduce(0, (sum, element) -> sum + element);  
20  
21     System.out.println("Result: " + r2);
```



Reduction: Example

```
18     int r2 = IntStream.rangeClosed(1, 5).parallel()  
19         .reduce(0, (sum, element) -> sum + element);  
20  
21     System.out.println("Result: " + r2);
```

1

Sum

2

Elements

3

4

5

Reduction: Example

```
18     int r2 = IntStream.rangeClosed(1, 5).parallel()  
19         .reduce(0, (sum, element) -> sum + element);  
20  
21     System.out.println("Result: " + r2);
```

3

Sum

3

Elements

4

5

Reduction: Example

```
18     int r2 = IntStream.rangeClosed(1, 5).parallel()  
19         .reduce(0, (sum, element) -> sum + element);  
20  
21     System.out.println("Result: " + r2);
```

6

Sum

4

5

Elements

Reduction: Example

```
18     int r2 = IntStream.rangeClosed(1, 5).parallel()  
19         .reduce(0, (sum, element) -> sum + element);  
20  
21     System.out.println("Result: " + r2);
```

10

Sum

5

Elements

Reduction: Example

```
18     int r2 = IntStream.rangeClosed(1, 5).parallel()  
19         .reduce(0, (sum, element) -> sum + element);  
20  
21     System.out.println("Result: " + r2);
```

15

Sum

Elements

A Look Under the Hood

- Pipeline decomposed into subpipelines.
 - Each subpipeline produces a subresult.
 - Subresults combined into final result.



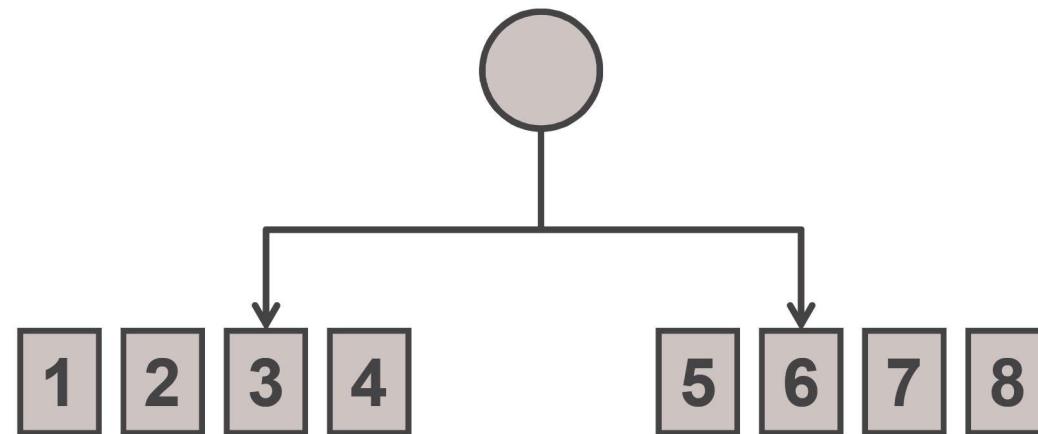
Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()  
19         .reduce(0, (sum, element) -> sum + element);
```



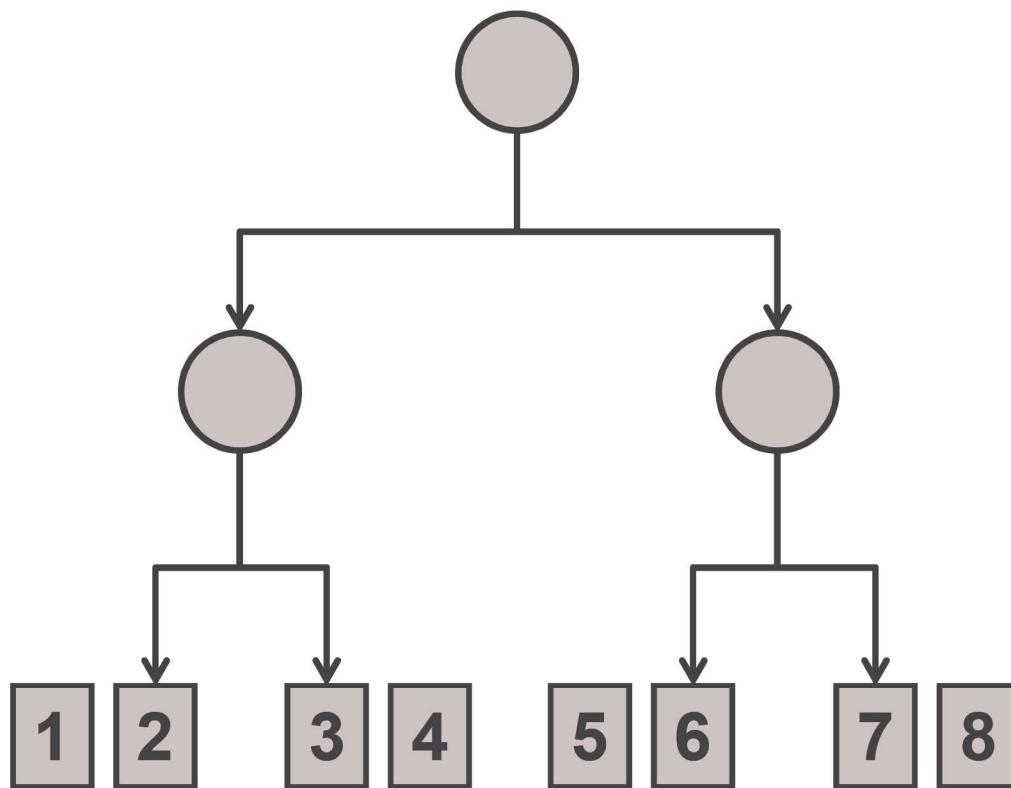
Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()  
19         .reduce(0, (sum, element) -> sum + element);
```



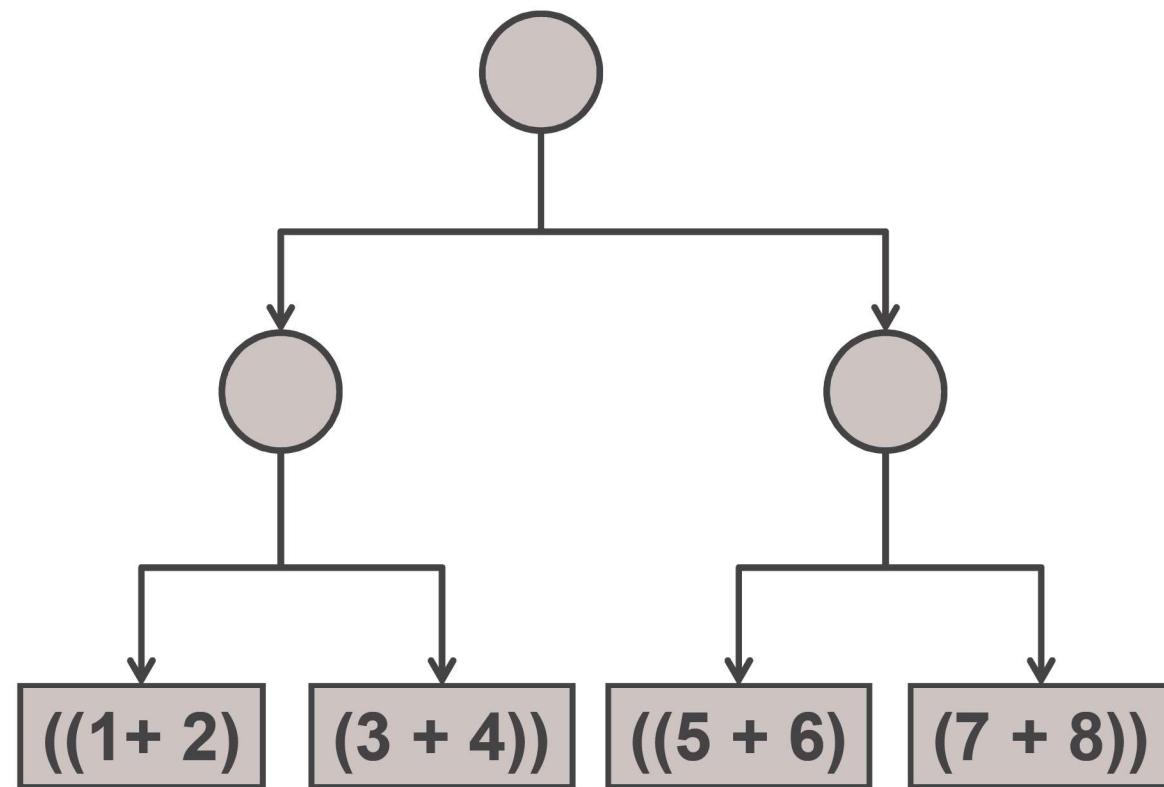
Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()  
19         .reduce(0, (sum, element) -> sum + element);
```



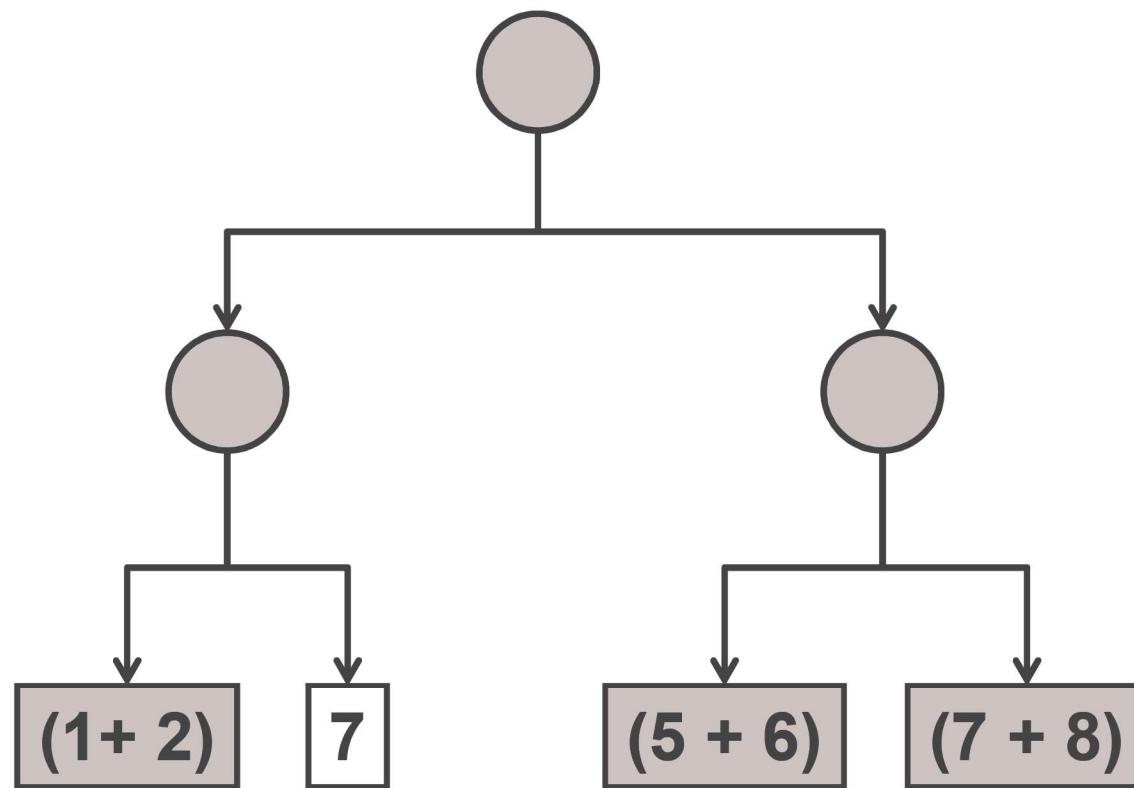
Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()  
19         .reduce(0, (sum, element) -> sum + element);
```



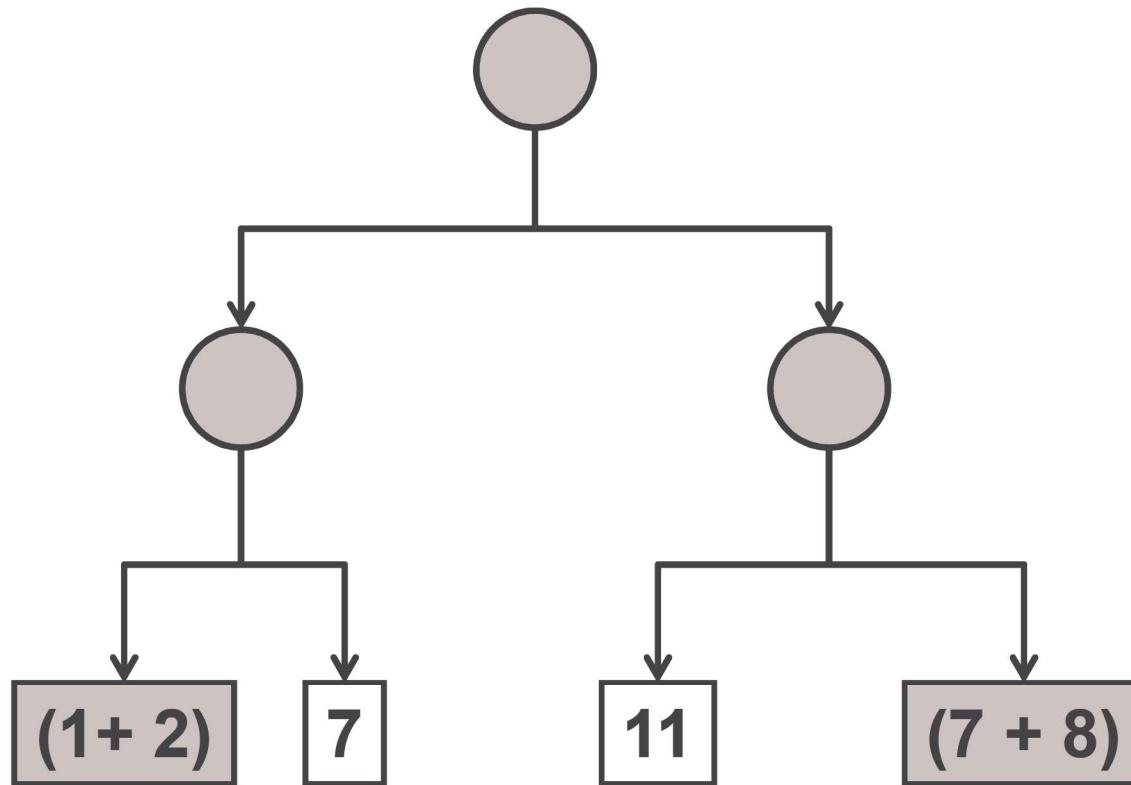
Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()  
19         .reduce(0, (sum, element) -> sum + element);
```



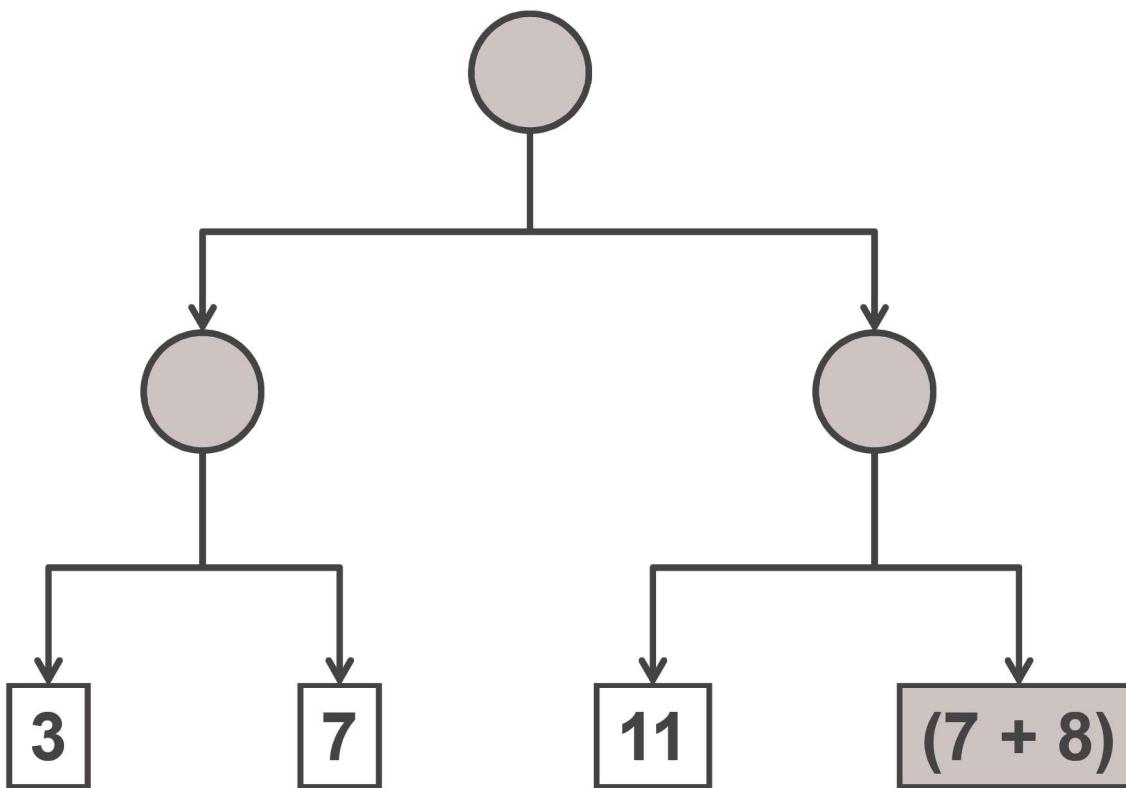
Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()  
19         .reduce(0, (sum, element) -> sum + element);
```



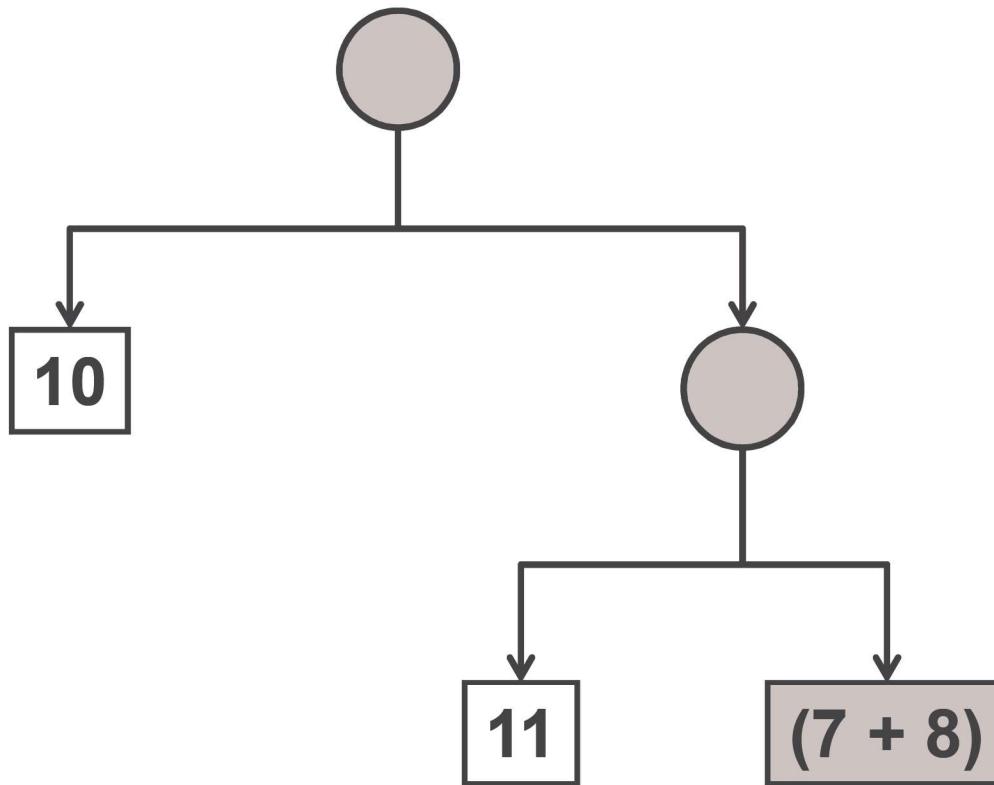
Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()  
19         .reduce(0, (sum, element) -> sum + element);
```



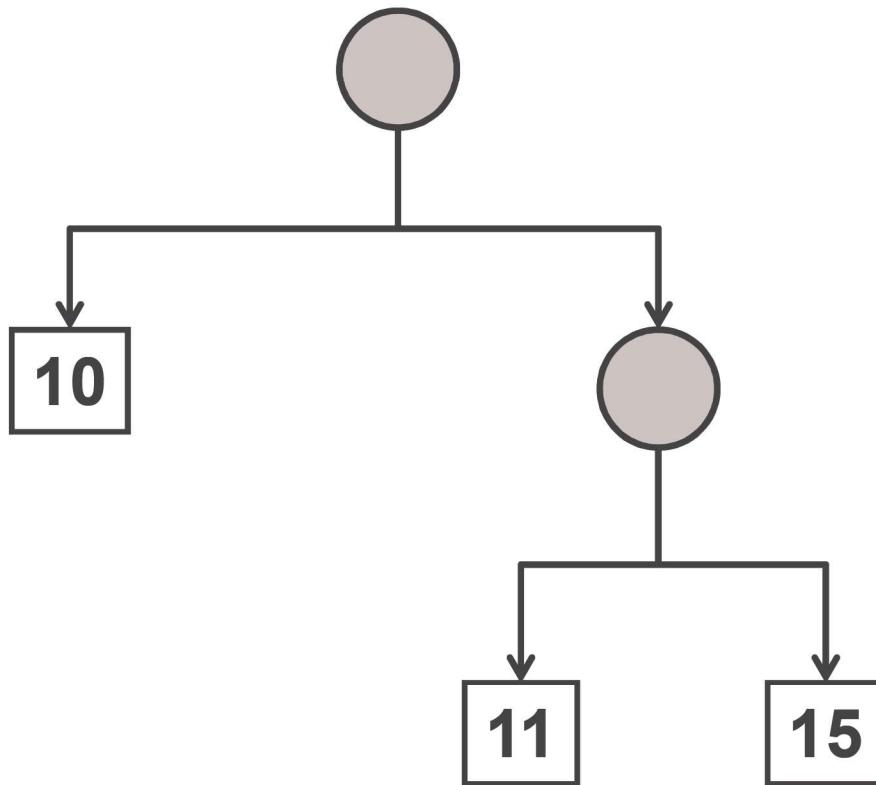
Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()  
19         .reduce(0, (sum, element) -> sum + element);
```



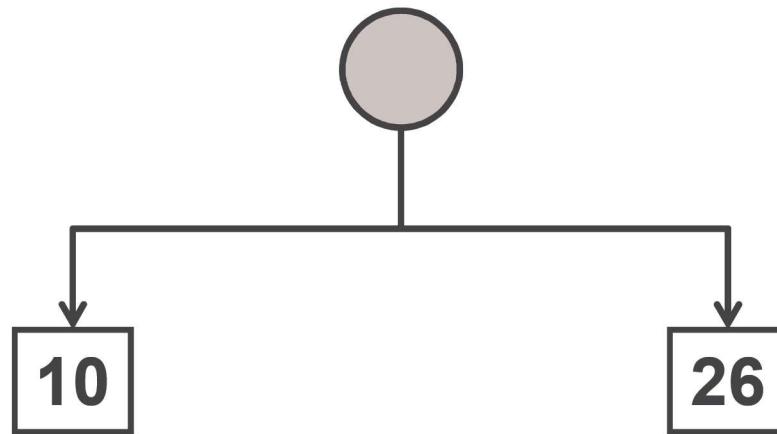
Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()  
19         .reduce(0, (sum, element) -> sum + element);
```



Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()  
19         .reduce(0, (sum, element) -> sum + element);
```



Illustrating Parallel Execution

```
18     int r2 = IntStream.rangeClosed(1, 8).parallel()  
19         .reduce(0, (sum, element) -> sum + element);
```

36

Performance

- Do not assume parallel is always faster.
 - Parallel not always the right solution.
 - Sometimes parallel is slower than sequential.

A Simple Performance Model

N = Size of the source data set

Q = Cost per element through the pipeline

$N * Q \approx$ Cost of the pipeline

- Larger $N * Q \rightarrow$ Higher chance of good parallel performance
- Easier to know N than Q
- You can reason qualitatively about Q
 - Simple pipeline example
 - $N > 10K$. $Q=1$
 - Reduction using sum
 - Complex pipelines might
 - Contain filters
 - Contain limit operation
 - Complex reduction using `groupingBy()`

Practice

- Practice 5-1: Calculate Total Sales Without a Pipeline
- Practice 5-2: Calculate Sales Totals Using Parallel Streams
- Practice 5-3: Calculate Sales Totals Using Parallel Streams and Reduce

Summary

In this lesson, you should have learned how to:

- Review the key characteristics of streams
- Describe how to make a stream pipeline execute in parallel
- List the key assumptions needed to use a parallel pipeline
- Define reduction & Describe why reduction requires an associative function
- Calculate a value using reduce
- Describe the process for decomposing and then merging work
- List the key performance considerations for parallel streams



Practice 1: Overview

- This practice covers the following topics:
 - Using SQL Developer
 - Selecting all data from different tables
 - Describing the structure of tables
 - Performing arithmetic calculations and specifying column names

