Learning Objectives in this Part of the Lesson

- Understand parallel stream internals, e.g.
  - Know what can change & what can’t
  - Partition a data source into “chunks”
  - Process chunks in parallel
  - Configure the Java 8 parallel stream common fork-join pool
  - Avoid pool starvation & improve performance w/ ManagedBlocker
  - Perform a reduction that combines partial results into a single result
  - Learn to implement a concurrent collector
  - Recognize how a parallel stream is constructed & executed
Parallel Stream
Construction & Execution
Parallel Stream Construction & Execution

• Recall that intermediate operations are “lazy”

Parallel Stream Construction & Execution

• Recall that intermediate operations are “lazy”
• i.e., they don’t start to run until a terminal operator is reached

A stream pipeline is constructed at runtime via an internal representation. At runtime a linked list of stream source & intermediate operations is build, one per “stage” in pipeline.

List<String> ls = ...
List<String> sortedAWords = ls
        .stream()
        .map(String::toUpperCase)
        .filter(s -> s.startsWith("A"))
        .sorted()
        .collect(toList());
Parallel Stream Construction & Execution

- A stream pipeline is constructed at runtime via an internal representation.
- Each pipeline stage is described by a bitmap of \textit{stream flags} internally.

<table>
<thead>
<tr>
<th>Stream Flag</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIZED</td>
<td>Size of stream is known</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>Elements of stream are distinct</td>
</tr>
<tr>
<td>SORTED</td>
<td>Elements of the stream are sorted in natural order</td>
</tr>
<tr>
<td>ORDERED</td>
<td>Stream has meaningful encounter order</td>
</tr>
</tbody>
</table>

These flags are a subset of the flags that can be defined by a spliterator.

\textbf{Input} \(x\)  
\textbf{Output} \(f(x)\)  
\textbf{Output} \(g(f(x))\)  
\textbf{Output} \(h(g(f(x)))\)  
\textbf{R collect(Collector<...> collector)}
A stream pipeline is constructed at runtime via an internal representation.

Each pipeline stage is described by a bitmap of stream flags internally.

Source stage stream flags are derived from spliterator characteristics, e.g.

```
<table>
<thead>
<tr>
<th>Collection</th>
<th>Sized</th>
<th>Ordered</th>
<th>Sorted</th>
<th>Distinct</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HashSet</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>TreeSet</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
```

Parallel Stream Construction & Execution

Stream generate() & iterate() methods create streams that are not sized!
Parallel Stream Construction & Execution

• A stream pipeline is constructed at runtime via an internal representation
  • Each pipeline stage is described by a bitmap of stream flags internally
  • Source stage stream flags are derived from spliterator characteristics
  • Each intermediate operation affects the stream flags

Input $x$

Stream map(Function<$\ldots$> mapper)

Output $f(x)$

Stream filter(Predicate<$\ldots$> pred)

Output $g(f(x))$

Stream sorted()

Output $h(g(f(x)))$

R collect(Collector<$\ldots$> collector)
A stream pipeline is constructed at runtime via an internal representation

Each pipeline stage is described by a bitmap of stream flags internally

Source stage stream flags are derived from spliterator characteristics

Each intermediate operation affects the stream flags, e.g.

map()

- Clears SORTED & DISTINCT but keeps SIZED
A stream pipeline is constructed at runtime via an internal representation.

- Each pipeline stage is described by a bitmap of *stream flags* internally.
- Source stage stream flags are derived from spliterator characteristics.
- Each intermediate operation affects the stream flags, e.g.
  - `map()` keeps SORTED & DISTINCT but clears SIZED.
  - `filter()`.
A stream pipeline is constructed at runtime via an internal representation.

- Each pipeline stage is described by a bitmap of stream flags internally.
- Source stage stream flags are derived from spliterator characteristics.
- Each intermediate operation affects the stream flags, e.g.
  - map()
  - filter()
  - sorted()
    - Keeps SIZED & DISTINCT & adds SORTED.

Parallel Stream Construction & Execution

<table>
<thead>
<tr>
<th>Operations</th>
<th>Stream Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input x</td>
<td>Input x</td>
</tr>
<tr>
<td>Stream map()</td>
<td>Output f(x)</td>
</tr>
<tr>
<td>Stream filter()</td>
<td>Output g(f(x))</td>
</tr>
<tr>
<td>Stream sorted()</td>
<td>Output h(g(f(x)))</td>
</tr>
<tr>
<td>R collect()</td>
<td></td>
</tr>
</tbody>
</table>
Parallel Stream Construction & Execution

- A stream pipeline is constructed at runtime via an internal representation
- Each pipeline stage is described by a bitmap of stream flags internally
- Source stage stream flags are derived from spliterator characteristics
- Each intermediate operation affects the stream flags
- As the pipeline is being constructed the flags at each stage are updated

```
Input x
Stream map(Function<...> mapper)
Output f(x)
Stream filter(Predicate<...> pred)
Output g(f(x))
Stream sorted()
Output h(g(f(x)))
R collect(Collector<...> collector)
```
A stream pipeline is constructed at runtime via an internal representation:

- Each pipeline stage is described by a bitmap of *stream flags* internally.
- Source stage stream flags are derived from spliterator characteristics.
- Each intermediate operation affects the stream flags.
- As the pipeline is being constructed, the flags at each stage are updated.
  - e.g., flags for a previous stage are combined with the current stage's behavior to derive a new set of flags.

**Parallel Stream Construction & Execution**

- **Input x**
  - **Stream map** *(Function<...> mapper)*
    - **Output f(x)**
      - **Stream filter** *(Predicate<...> pred)*
        - **Output g(f(x))**
          - **Stream sorted**
            - **Output h(g(f(x)))**
              - **R collect** *(Collector<...> collector)*
A stream pipeline is constructed at runtime via an internal representation.

- Each pipeline stage is described by a bitmap of *stream flags* internally.
- Source stage stream flags are derived from spliterator characteristics.
- Each intermediate operation affects the stream flags.
- As the pipeline is being constructed, the flags at each stage are updated.
  - e.g., flags for a previous stage are combined with the current stage’s behavior to derive a new set of flags.

```java
Set<String> ts = new TreeSet<>(...);
List<String> sortedAWords =
    ts.stream()
    .filter(s -> s.startsWith("a"))
    .sorted()
    .collect(toList());
```

Redundant operation can be elided since the source is already sorted.
Parallel Stream Construction & Execution

- When terminal operation runs the stream framework picks an execution plan:

  **Input x**

  **Stream map(Function<...> mapper)**

  **Output f(x)**

  **Stream filter(Predicate<...> pred)**

  **Output g(f(x))**

  **Stream sorted()**

  **Output h(g(f(x)))**

  **R collect(Collector<...> collector)**

Parallel Stream Construction & Execution

- When terminal operation runs the stream framework picks an execution plan
- The plan is based on properties of the source & aggregate operations

```
Input x
Stream map(Function<…> mapper)
Output f(x)
Stream filter(Predicate<…> pred)
Output g(f(x))
Stream sorted()
Output h(g(f(x)))
R collect(Collectors<…> collector)
```
Parallel Stream Construction & Execution

- When terminal operation runs the stream framework picks an execution plan
  - The plan is based on properties of the source & aggregate operations
  - Intermediate operations are divided into two categories

```
Streammap(Function<...> mapper)
Streamfilter(Predicate<...> pred)
R collect(Collector<...> collector)
```

Example Diagram:
```
Input x

Stream map(Function<...> mapper)
Output f(x)

Stream filter(Predicate<...> pred)
Output g(f(x))

Stream sorted()
Output h(g(f(x)))

R collect(Collector<...> collector)
```
Parallel Stream Construction & Execution

- When terminal operation runs the stream framework picks an execution plan
- The plan is based on properties of the source & aggregate operations
- Intermediate operations are divided into two categories:
  - Stateless
    - e.g., filter(), map(), flatMap(), etc.

A pipeline with only stateless operations runs in one pass (even if it’s parallel)
Parallel Stream Construction & Execution

• When terminal operation runs the stream framework picks an execution plan
  • The plan is based on properties of the source & aggregate operations
  • Intermediate operations are divided into two categories:
    • Stateless
    • Stateful
      • e.g., sorted(), limit(), distinct(), etc.

A pipeline with stateful operations is divided into sections & runs in multiple passes
Parallel Stream Construction & Execution

- When terminal operation runs the stream framework picks an execution plan
  - The plan is based on properties of the source & aggregate operations
  - Intermediate operations are divided into two categories
  - Terminal operations are also divided into two categories

![Diagram of parallel stream construction and execution]

- **Input x**
  - **Stream map** (Function<…> mapper)
    - Output **f(x)**
  - **Stream filter** (Predicate<…> pred)
    - Output **g(f(x))**
  - **Stream sorted**
    - Output **h(g(f(x)))**
  - **R collect** (Collector<…> collector)
Parallel Stream Construction & Execution

• When terminal operation runs the stream framework picks an execution plan
  • The plan is based on properties of the source & aggregate operations
  • Intermediate operations are divided into two categories
  • Terminal operations are also divided into two categories
    • Non-short-circuiting
      • e.g., reduce(), collect(), forEach(), etc.

Terminal operation can process data in bulk using spliterator’s forEachRemaining()
Parallel Stream Construction & Execution

- When terminal operation runs the stream framework picks an execution plan
  - The plan is based on properties of the source & aggregate operations
  - Intermediate operations are divided into two categories
  - Terminal operations are also divided into two categories
    - Non-short-circuiting
    - Short-circuiting
      - e.g., anyMatch(), findFirst(), etc.

Terminal operation must process data one element at a time using tryAdvance()
End of Java 8 Parallel Stream Internals (Part 6)