Java 8 Parallel Stream Internals  
(Part 7)  
Douglas C. Schmidt 
d.schmidt@vanderbilt.edu  
www.dre.vanderbilt.edu/~schmidt  
Professor of Computer Science 
Institute for Software Integrated Systems  
Vanderbilt University  
Nashville, Tennessee, USA
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 concurrent & non-concurrent collectors
- Recognize how a parallel stream is constructed & executed
Parallel Stream
Construction & Execution
Recall that intermediate operations are “lazy”

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)`

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

See www.logicbig.com/tutorials/core-java-tutorial/java-util-stream/lazy-evaluation
A stream pipeline is constructed at runtime via an internal representation.

At runtime a linked list of stream source & intermediate operations is built, one per “stage” in pipeline.

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

A stream pipeline is constructed at runtime via an internal representation.

Each pipeline stage is described by a bitmap of *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.

```
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, e.g.:

- `Stream map(Function<...> mapper)`
- `Stream filter(Predicate<...> pred)`
- `Stream sorted()`
- `R collect(Collector<...> collector)`

<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>

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(Collectors$<\ldots>\ collector$)
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

```
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, e.g.
  - `map()` keeps SORTED & DISTINCT but clears 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()
  - filter()
  - sorted()
  - Keeps SIZED & DISTINCT & adds SORTED
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:

1. Each pipeline stage is described by a bitmap of stream flags internally.
2. Source stage stream flags are derived from spliterator characteristics.
3. Each intermediate operation affects the stream flags.
4. 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

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(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

\[ \text{Input } x \]

- \textit{Stream map}(Function<\ldots> mapper)
  - Output \( f(x) \)
- \textit{Stream filter}(Predicate<\ldots> pred)
  - Output \( g(f(x)) \)
- \textit{Stream sorted()}
  - Output \( h(g(f(x))) \)
- \textit{R collect}(Collector<\ldots> 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 stream operations](image)

- Stream map(Function<...> mapper)
- Stream filter(Predicate<...> pred)
- Stream sorted()
- R collect(Collector<...> collector)

Input x

Output f(x)

Output g(f(x))

Output h(g(f(x)))
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()
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 7)