JavaOne™

ORACLE®

Distributed Streams

The Stream API on Steroids

Brian Oliver
Architect | Specification Lead
Coherence Engineering | Oracle Corporation
October 2015

Email: brian.oliver@oracle.com
Twitter: @pinocchiocode

JavaOne ORACLE

Safe Harbor Statement

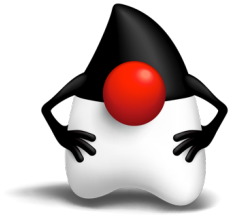
The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Program Agenda

1. Streams
2. Distributed Streams
3. Demonstrations
4. Optimizations
5. Summary

Streams

Streams

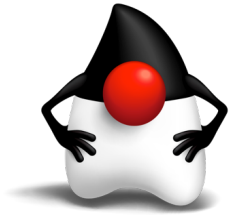


A defining new feature of the Java 8 Platform

- Provide Functional Programming constructs for processing information
  - Typically requires heavy use of lambdas
- Both a query and aggregation API for various ‘stream sources’
  - Collections, files, sockets…
- Define a pipeline of zero or more intermediate operations and a single terminal operation
- Replaces external with internal iteration
- Enables execution parallelization without code changes

Streams

Query Example

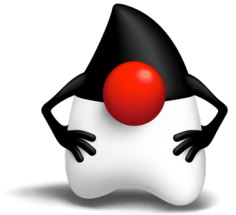


```
Map<Long, Order> orders = new HashMap<>();

Set<Long> customersForProduct = orders.values().stream()
    .flatMap(order -> order.getItems().stream())
    .filter(orderItem -> orderItem.getProductId() == productId)
    .map(OrderItem::getOrder)
    .map(Order::getCustomerId)
    .collect(toSet());
```

Streams

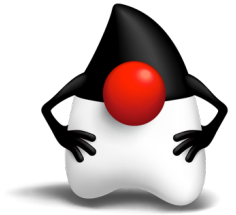
Aggregation Example



```
Map<Long, Double> salesByProduct = orders.values().stream()
    .flatMap(order -> order.getItems().stream())
    .collect(groupingBy(OrderItem::getProductId,
                        summingDouble(OrderItem::getTotal)));
```

Streams

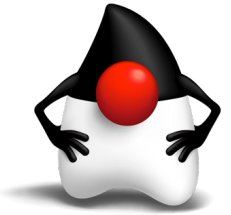
Intermediate Operations



- *Stateless* intermediate operations are:
  - `filter(Predicate<? super T> predicate)`
  - `map(Function<? super T,? extends R> mapper)`
  - `flatMap(Function<? super T,? extends Stream<? extends R>> mapper)`
  - `peek(Consumer<? super T> action)`
- *Stateful* intermediate operations are:
  - `distinct()`
  - `limit(long maxSize)`
  - `skip(long n)`
  - `sorted([Comparator<? super T> comparator])`

Streams

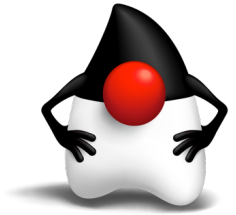
Terminal Operations



- `forEach(Consumer<? super T> action)`
- `reduce(BinaryOperator<T> accumulator)`
- `toArray(IntFunction<A[]> generator)`
- `min(Comparator<? super T> comparator)`
- `max(Comparator<? super T> comparator)`
- `count()`
- `anyMatch(Predicate<? super T> predicate)`
- `allMatch(Predicate<? super T> predicate)`
- `noneMatch(Predicate<? super T> predicate)`
- `findFirst()`
- `findAny()`
- `collect(Collector<? super T,A,R> collector)`

Streams

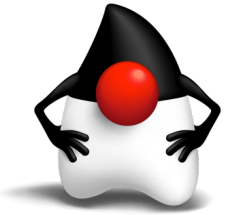
Collectors



- Some of the most powerful use cases require *Collectors*
- Allow custom “collection” of results in pipeline
- A *Collector* has four components:
  - **Supplier**: creates a new result container
  - **Accumulator**: incorporates single data element into a result container
  - **Combiner**: merges two result containers into one
  - **Finisher**: performs final transformation of a result container

Streams

Collectors



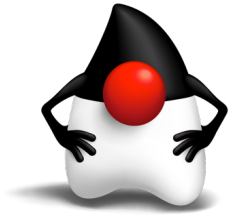
- Some Collectors, such as *groupingBy*, can have nested Collectors:

```
Map<Long, Map<String, Double>> salesByCustomerByProduct =
    orders.values().stream()
        .flatMap(order -> order.getItems().stream())
        .collect(groupingBy(
                    item -> item.getOrder().getCustomerId(),
                    groupingBy(OrderItem::getName,
                               summingDouble(OrderItem::getTotal))));
```

- Powerful and flexible
  - No need for custom code
- Much better than a bunch of nested loops?

Streams

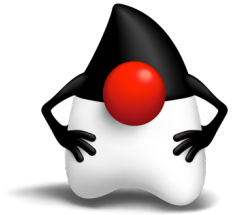
Collectors



- averagingInt/Long/Double
- summingInt/Long/Double
- summarizingInt/Long/Double
- groupingBy
- partitioningBy
- joining
- mapping
- minBy, maxBy
- counting
- reducing
- toCollection/Set/List/Map

What is Oracle Coherence?

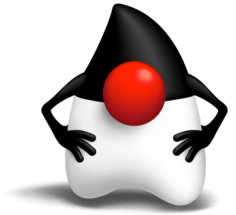
What is Oracle Coherence?



- In-Memory-Data-Grid
  - Clustered, Shared, Data-Structures, RAID-like availability, Dynamic-Live-Scale-Out
  - Pure Java
  - Typically used for Distributed Caching
  - Fully parallel query / aggregation / in-place processing
  - Cluster-wide concurrently control
- Implements Standard Java Platform Data-Structures
  - Eg: NamedCache == java.util.Map (on steroids)

Distributed Streams

Distributed Streams

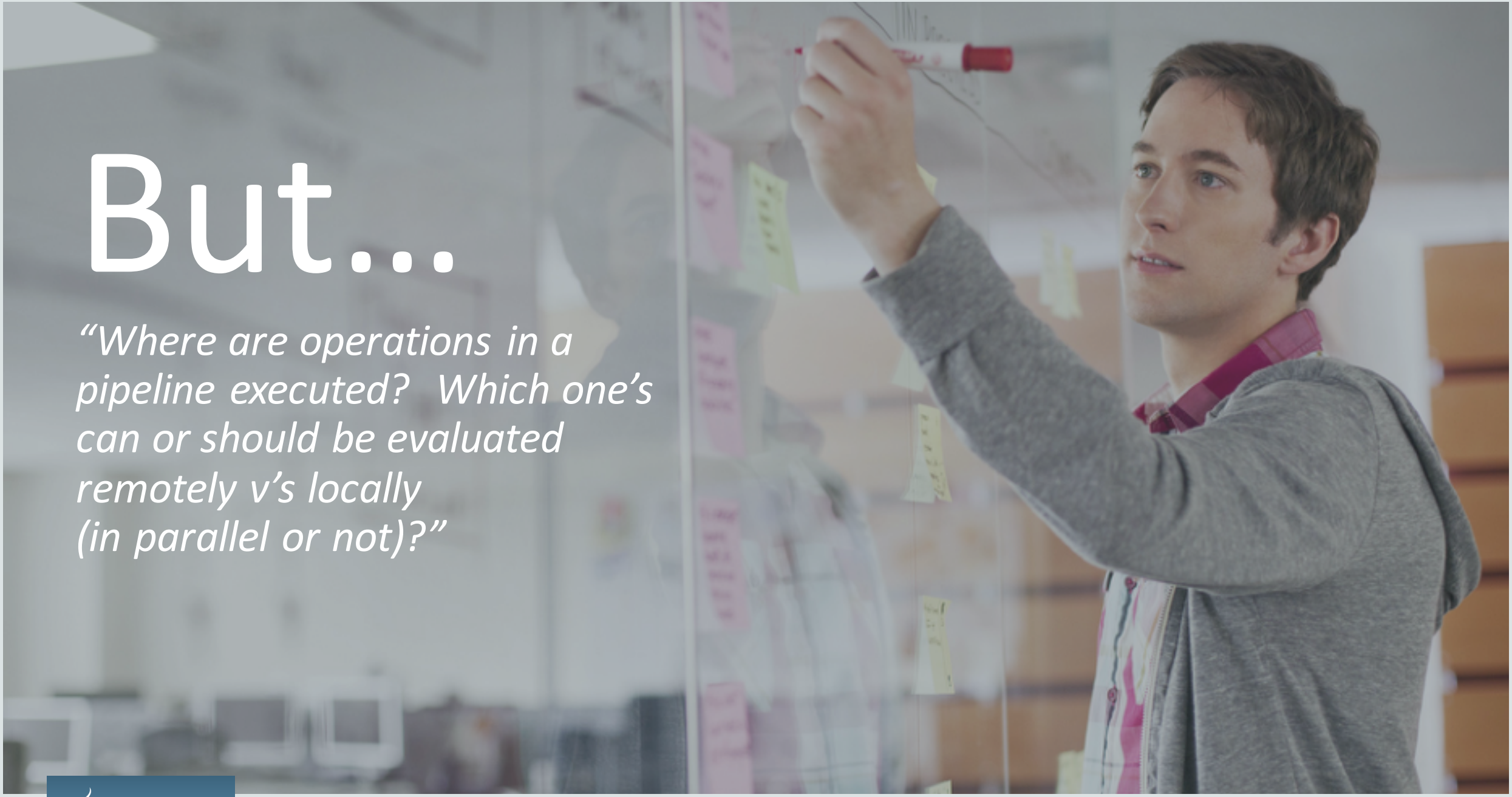


Limitations of the current Java Platform

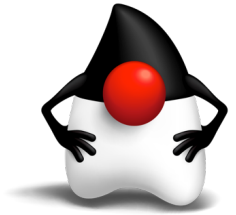
- Query & Aggregation concepts already exist in Coherence…
- Standard Java Design and Implementation is limiting
- Parallel execution only within a single process
  - We want to parallelize it across many processes on many machines
- Design assumes data locality
  - Non-serializable Lambdas used as arguments
  - Built-in Collectors and some of the return types are not serializable
- But… it's an awesome model for distributed computing!
  - Could we make them work in a distributed manner?

But…

"Where are operations in a pipeline executed? Which one's can or should be evaluated remotely v's locally (in parallel or not)?"



Distributed Streams

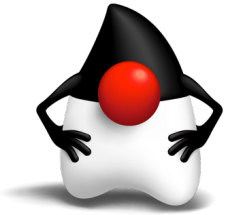


Where can and should operations be executed? What about collectors?

```
Map<Long, Order> orders = new HashMap<>();

Set<Long> customersForProduct = orders.values().stream()
    .flatMap(order -> order.getItems().stream())
    .filter(orderItem -> orderItem.getProductId() == productId)
    .map(OrderItem::getOrder)
    .map(Order::getCustomerId)
    .collect(toSet());
```

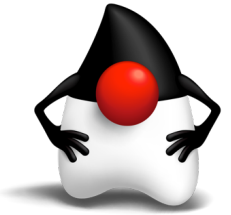
Distributed Streams



Terminal Operations: Evaluate Locally or Remotely (or both?)

- `forEach(Consumer<? super T> action)`
- `reduce(BinaryOperator<T> accumulator)`
- `toArray(IntFunction<A[]> generator)`
- `min(Comparator<? super T> comparator)`
- `max(Comparator<? super T> comparator)`
- `count()`
- `anyMatch(Predicate<? super T> predicate)`
- `allMatch(Predicate<? super T> predicate)`
- `noneMatch(Predicate<? super T> predicate)`
- `findFirst()`
- `findAny()`
- `collect(Collector<? super T,A,R> collector)`

Distributed Streams



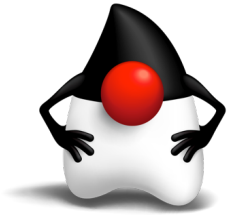
On Coherence

- Re-implemented Stream API
  - Support for serialization of pipeline operations, including partial and final results
  - Evaluate pipeline operations in a distributed manner, in parallel
- Implemented serializable versions of the standard JDK Collectors

```
long customerId = ...;

List<Order> custOrders = orders.stream()
        .map(Map.Entry::getValue)
        .filter(order -> order.getCustomerId() == customerId)
        .collect(RemoteCollectors.toList());
```

Distributed Streams



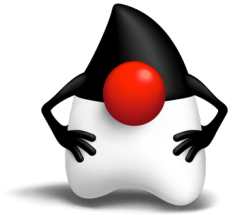
Query Example

- What should we process? Keys? Values? Entries?
  - Eg: Don't Call "stream" on result of values(), entrySet(), keySet()

```
NamedCache<Long, Order> orders = ...;

Set<Long> customersForProduct = orders.stream()
        .map(Map.Entry::getValue)
        .flatMap(order -> order.getItems().stream())
        .filter(orderItem -> orderItem.getProductId() == productId)
        .map(OrderItem::getOrder)
        .map(Order::getCustomerId)
        .collect(RemoteCollectors.toSet());
```

Distributed Streams



Aggregation Example

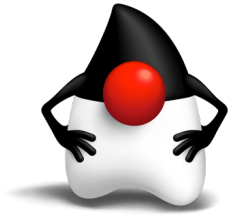
```
import static com.tangosol.util.stream.RemoteCollectors.*;

NamedCache<Long, Order> orders = ...;

Map<Long, Map<String, Double>> salesByCustomerByProduct =
    orders.stream()
        .map(Map.Entry::getValue)
        .flatMap(order -> order.getItems().stream())
        .collect(groupingBy(
                    (item) -> item.getOrder().getCustomerId(),
                    groupingBy(OrderItem::getName,
                                summingDouble(OrderItem::getTotal))));
```

Optimizations

Optimizations



Pre-filter to avoid placing entries in the pipeline (and creating garbage)

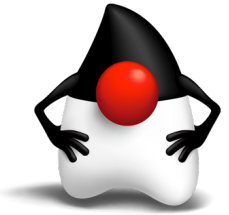
- Don’t do this:

```
orders.stream()
    .map(Map.Entry::getValue)
    .filter(order -> order.getCustomerId() == customerId)
    ...
```

- Do this instead:

```
orders.stream(equal(Order::getCustomerId, customerId))
    .map(Map.Entry::getValue)
    .filter(order -> order.getCustomerId() == customerId)
    ...
```

Optimizations



Pre-map to allow using forward indexes (and avoid deserialization)

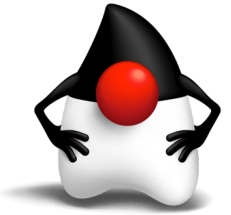
- Don’t do this:

```
orders.stream()
    .map(Map.Entry::getValue)
    .mapToDouble(Order::getTotal)
    .sum();
```

- Do this instead:

```
orders.stream(Order::getTotal)
    .map(Map.Entry::getValue)
    .mapToDouble(Order::getTotalNumber::doubleValue)
    .sum();
```

Optimizations



Combine them (use indexes for getCustomer and getTotal)

- So the following:

```
double total =
    orders.stream()
        .map(Map.Entry::getValue)
        .filter(order -> order.getCustomerId() == customerId)
        .mapToDouble(Order::getTotal)
        .sum();
```

- Becomes:

```
double total =
    orders.stream(equal(Order::getCustomerId, customerId),
                  Order::getTotal)
        .mapToDouble(Number::doubleValue)
        .sum();
```

Summary

Summary

CREATE THE FUTURE

Distributed Streaming Rocks!

- Stream API provides functional API for query and aggregation
- Collectors eliminate the need for custom code
- Coherence provides Distributed Stream API
  - Allows parallel processing of data streams across a cluster “in-place”
  - Scales multiple threads in a single process to multiple threads in multiple processes across machines across the Coherence cluster (with high-availablity)
  - Provides optimizations to avoid deserialization & use indexes
  - Stable results even during cluster failure / recovery
- Next generation? Real-time continuous stream processing

Next Steps

Start Playing!

Coherence for Developers!

CREATE THE FUTURE

- https://www.oracle.com/goto/coherence
- https://coherence.java.net


https://twitter.com/OracleCoherence


https://www.linkedin.com/grp/home?gid=1782166


https://blogs.oracle.com/OracleCoherence

http://www.youtube.com/OracleCoherence

JavaOne ORACLE

CREATE THE FUTURE

Note: The speaker notes for this slide include detailed instructions on how to reuse this Section Header slide in another presentation.

Tip! Remember to remove this text box.

Safe Harbor Statement

The preceding is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Integrated Cloud

Applications & Platform Services

JavaOne™

ORACLE®