Build Beam Source with Splittable DoFn

Boyuan Zhang
boyuanz@apache.org

Content

- BoundedSource and UnboundedSource overview
- Introduction of Splittable DoFn
- Why Splittable DoFn
- Build keyed components of source with Splittable DoFn
  - Size estimation, initial split, dynamic split, checkpoint, progress, watermark, drain, checkpointMark finalization
- Advanced usages

BoundedSource & UnboundedSource

- Beam APIs to enable users to build their own sources
- The root node of the pipeline
- BoundedSource
  - Read from a **finite** data source in a **batch** pipeline
    - FileIO, Redis...
  - Key points: fast
- UnboundedSource
  - Read from an **infinite** data source in a **streaming** pipeline
    - KafkaIO
  - Key points: advance watermark correctly, drain

What is Splittable DoFn

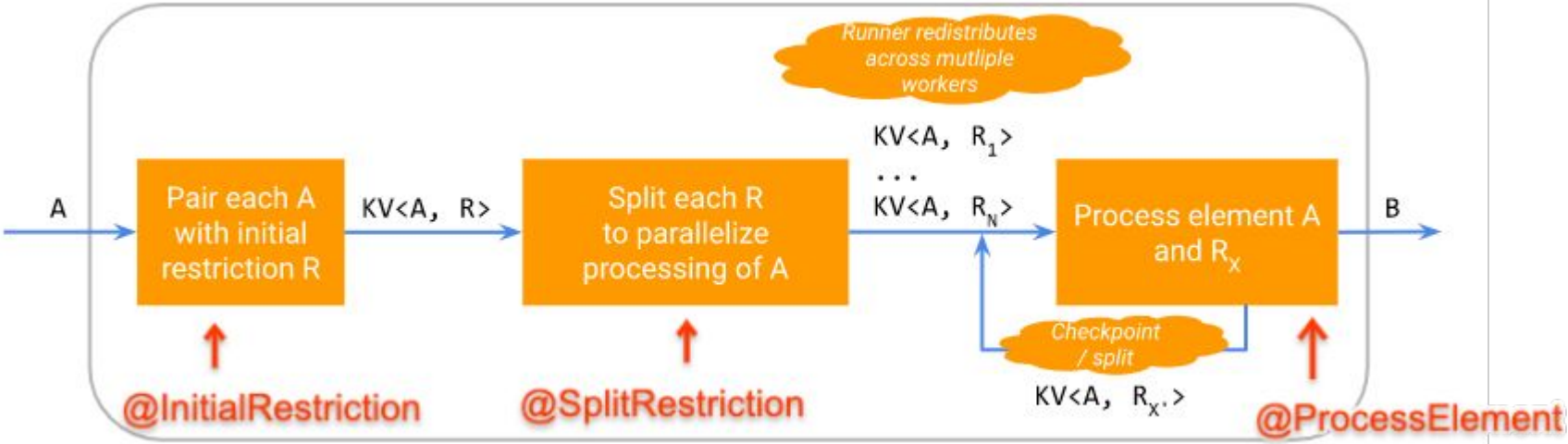
- DoFn
  - Same syntax as DoFn: @StartBundle, @FinishBundle, @ProcessElement…
    - But you **cannot** use user states and timers inside one Splittable DoFn
  - Responsible for processing **element** and **restriction** pairs.
    - Restriction: represents a subset of work that would have been necessary to been done when processing the element.
    - Kafka example: element -> Kafka TopicPartition, restriction -> an OffsetRange [0, Infinity)
- The ability to split **inside** one element
  - (element, restriction) -> (element, restriction_ 1) + (element, restriction_2)

A Simple Java Splittable DoFn

```
@BoundedPerElement
private static class FileToWordsFn extends DoFn<String, Integer> {
  @GetInitialRestriction ← initialize the restriction
  public OffsetRange getInitialRestriction(@Element String fileName) throws IOException {
    return new OffsetRange(0, new File(fileName).length());
  }

  @ProcessElement
  public void processElement(
      @Element String fileName,
      RestrictionTracker<OffsetRange, Long> tracker, ← to track current restriction
      OutputReceiver<Integer> outputReceiver)
      throws IOException {
    RandomAccessFile file = new RandomAccessFile(fileName, "r");
    seekToNextRecordBoundaryInFile(file, tracker.currentRestriction().getFrom());
    while (tracker.tryClaim(file.getFilePointer())) {
      outputReceiver.output(readNextRecord(file));
    }
  }

  // Providing the coder is only necessary if it can not be inferred at runtime.
  @GetRestrictionCoder
  public Coder<OffsetRange> getRestrictionCoder() {
    return OffsetRange.Coder.of();
  }
}
```

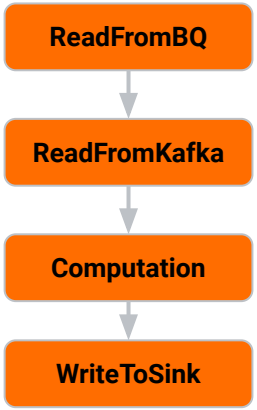
Execution of Splittable DoFn

- Runners run the graph expansion before start processing and execute the expanded graph



Benefits of Using Splittable DoFn

- Splittable DoFn provides parities to current Source APIs
- Splittable DoFn is the only IO framework for portable execution
- Unified model for both batch and streaming
- Having source-like operations at any node of one pipeline
    - For UnboundedSource & BoundedSource, it has to be the root node of the pipeline
    - Brings possibility of building advanced usages on sources



Writing Sources in Splittable DoFn
With KafkaIO example

Size Estimation

- Why estimate size:
  - In batch, the size estimation gives important signals to runners for making split decisions
  - In streaming, this estimation gives the backlog information
- APIs:
  - Splittable DoFn: @GetSize if provided, else RestrictionTracker.getProgress().getWorkCompleted()
  - BoundedSource: BoundedReader.getEstimatedSizeBytes()
  - UnboundedSource: UnboundedReader.getSplitBacklogBytes()

Size Estimation

KafkaIO in Splittable DoFn

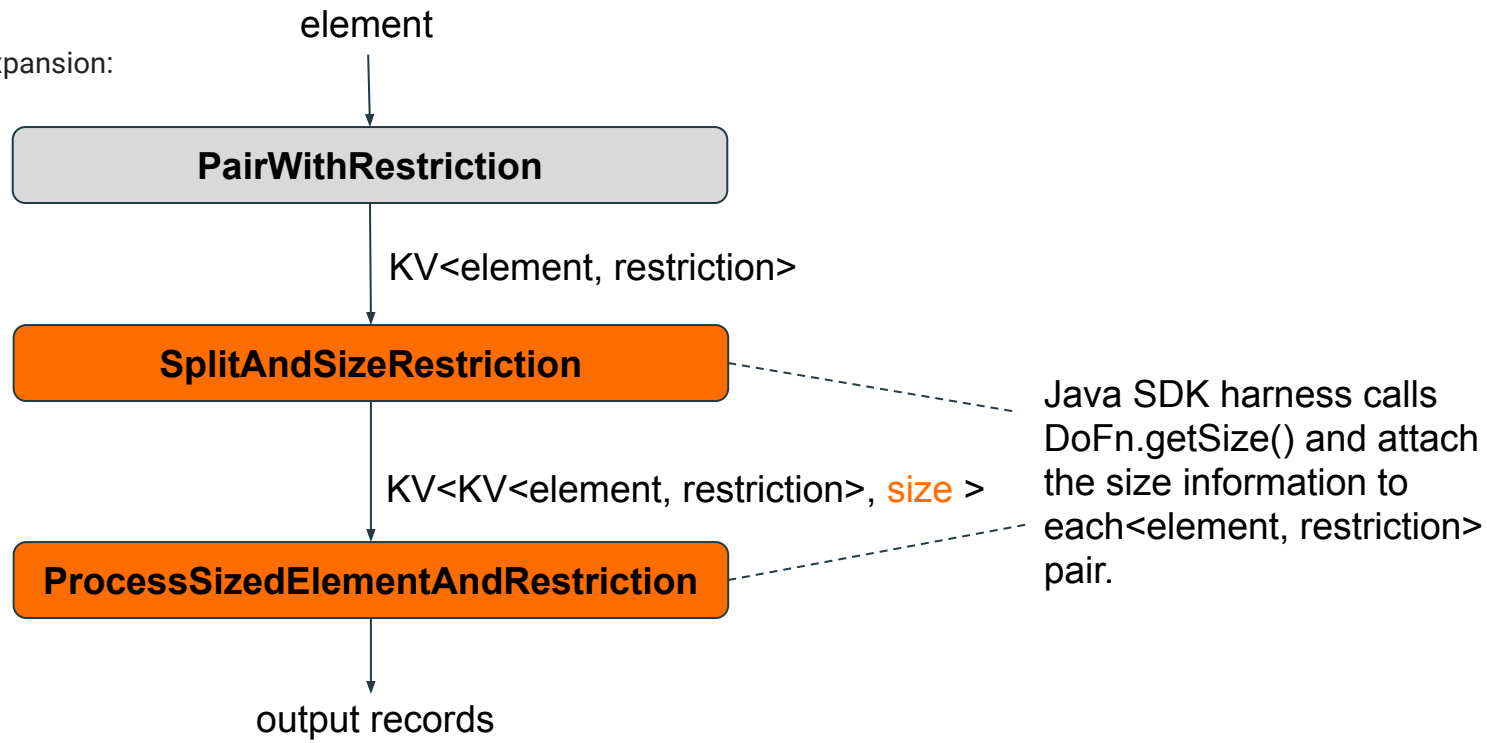
```
@GetSize
public double getSize(
    @Element KafkaSourceDescriptor kafkaSourceDescriptor, @Restriction OffsetRange offsetRange)
    throws Exception {
  double numRecords =
      restrictionTracker(kafkaSourceDescriptor, offsetRange).getProgress().getWorkRemaining();
  // Before processing elements, we don't have a good estimated size of records and offset gap.
  if (!avgRecordSize.asMap().containsKey(kafkaSourceDescriptor.getTopicPartition())) {
    return numRecords;
  }
  return avgRecordSize.get(kafkaSourceDescriptor.getTopicPartition()).getTotalSize(numRecords);
}
```

KafkaIO in UnboundedReader

```
@Override
public long getSplitBacklogBytes() {
  long backlogBytes = 0;

  for (PartitionState p : partitionStates) {
    long pBacklog = p.approxBacklogInBytes();
    if (pBacklog == UnboundedReader.BACKLOG_UNKNOWN) {
      return UnboundedReader.BACKLOG_UNKNOWN;
    }
    backlogBytes += pBacklog;
  }

  return backlogBytes;
}
```

How Size Estimation is invoked by Splittable DoFn

- Graph Expansion:



How runners get this size from SDK

For each KV<KV<element, restriction>, size>

- SDK encodes it into bytes by using KVCoder<KVCoder, DoubleCoder>
  - Both KVCoder and DoubleCoder are well-known coder
- Runners are able to decode bytes into KV<something runner doesn’t care, size> and get size information

Initial Split

- Initial split gives the execution initial parallelisms
- APIs
  - Splittable DoFn: @SplitRestriction
  - BoundedSource: BoundedSource.split()
  - UnboundedSource: UnboundeSource.split()

Initial split

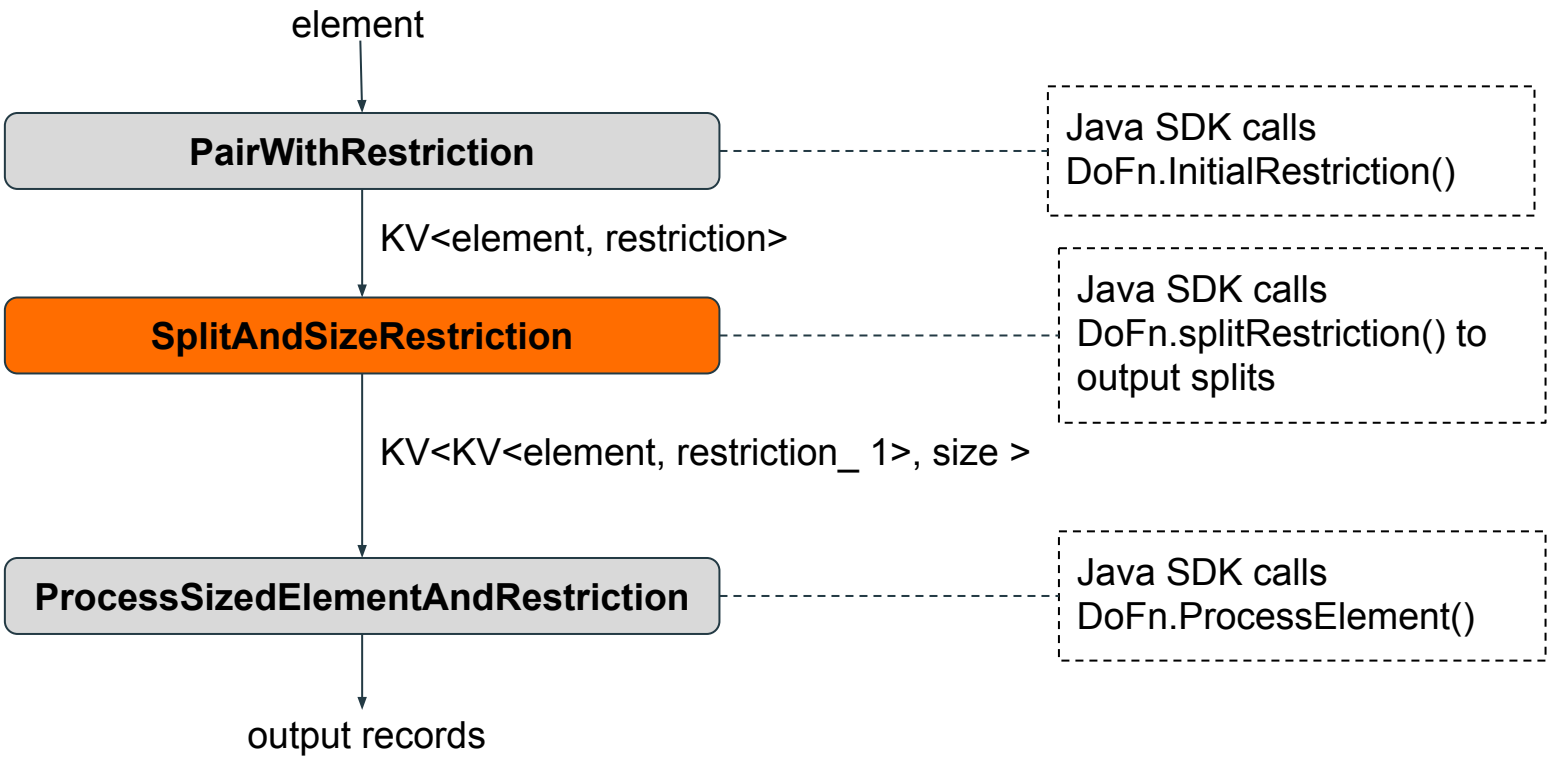
KafkaIO in Splittable DoFn

```java
@SplitRestriction
public void split(@Restriction OffsetRange restriction, OutputReceiver<OffsetRange> out) {
  out.output(restriction);
}
```

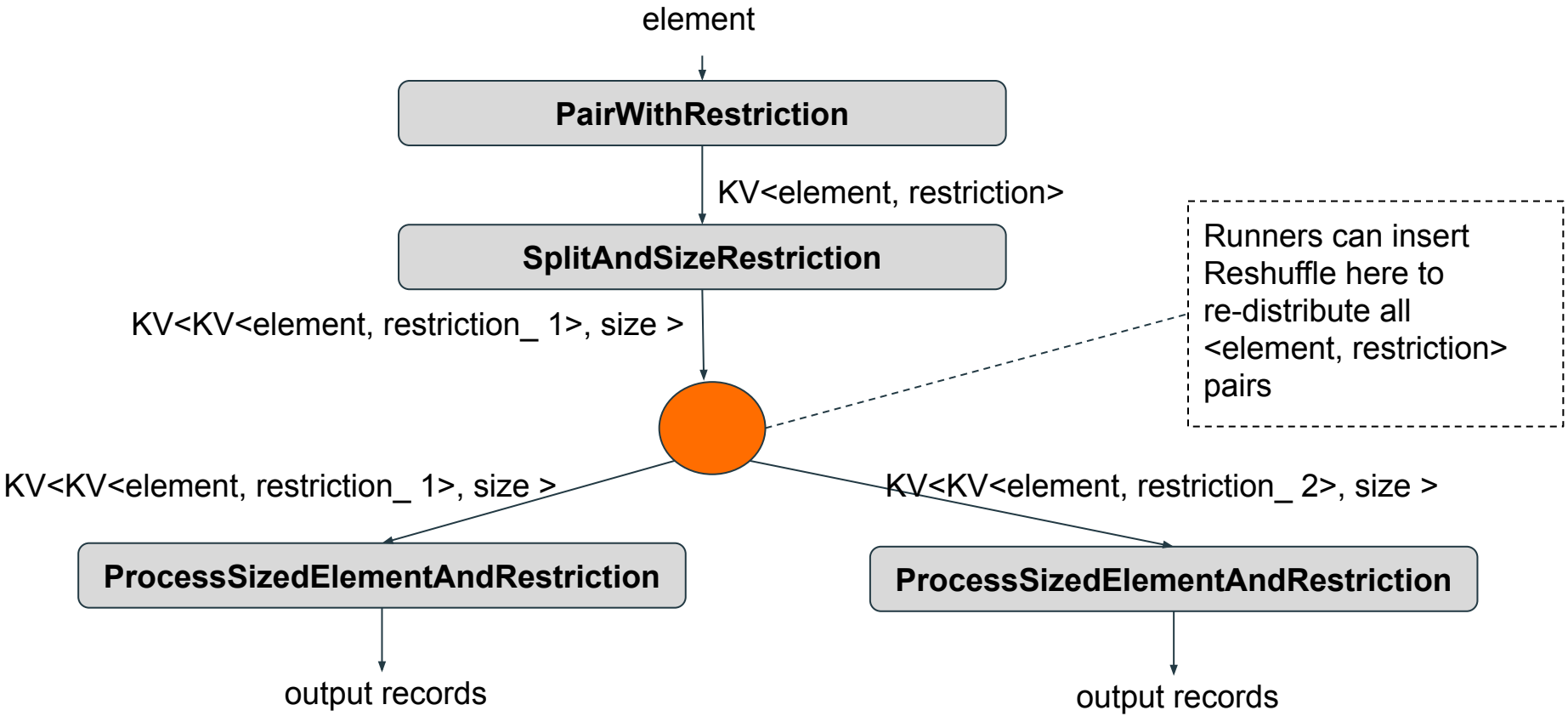
KafkaIO in UnboundedSource

```java
@Override
public List<KafkaUnboundedSource<K, V>> split(int desiredNumSplits, PipelineOptions options)
    throws Exception {

  List<TopicPartition> partitions = new ArrayList<>(spec.getTopicPartitions());

  partitions.sort(
      Comparator.comparing(TopicPartition::topic)
          .thenComparing(Comparator.comparingInt(TopicPartition::partition)));

  int numSplits = Math.min(desiredNumSplits, partitions.size());
  // XXX make all splits have the same # of partitions
  while (partitions.size() % numSplits > 0) {
    ++numSplits;
  }
  List<List<TopicPartition>> assignments = new ArrayList<>(numSplits);

  for (int i = 0; i < numSplits; i++) {
    assignments.add(new ArrayList<>());
  }
  for (int i = 0; i < partitions.size(); i++) {
    assignments.get(i % numSplits).add(partitions.get(i));
  }

  List<KafkaUnboundedSource<K, V>> result = new ArrayList<>(numSplits);

  for (int i = 0; i < numSplits; i++) {
    List<TopicPartition> assignedToSplit = assignments.get(i);
    result.add(
        new KafkaUnboundedSource<>(
            spec.toBuilder()
                .setTopics(Collections.emptyList())
                .setTopicPartitions(assignedToSplit)
                .build(),
            i));
  }

  return result;
}
```

How Initial Split is invoked by Splittable DoFn



How runners distribute these splits



Progress

- While SDK executing an Splittable DoFn, runners want to know the current progress for:
  - Providing metrics to UI
  - Making dynamic decisions
- Progress includes:
  - Work that has been done
  - Work that is remaining
- APIs
  - Splittable DoFn: HasProgress Interface for RestrictionTracker
  - BoundedSource: BoundedReader.getFractionConsumed()
  - UnboundedSource: UnboundedReader.getSplitBacklogBytes()

Progress

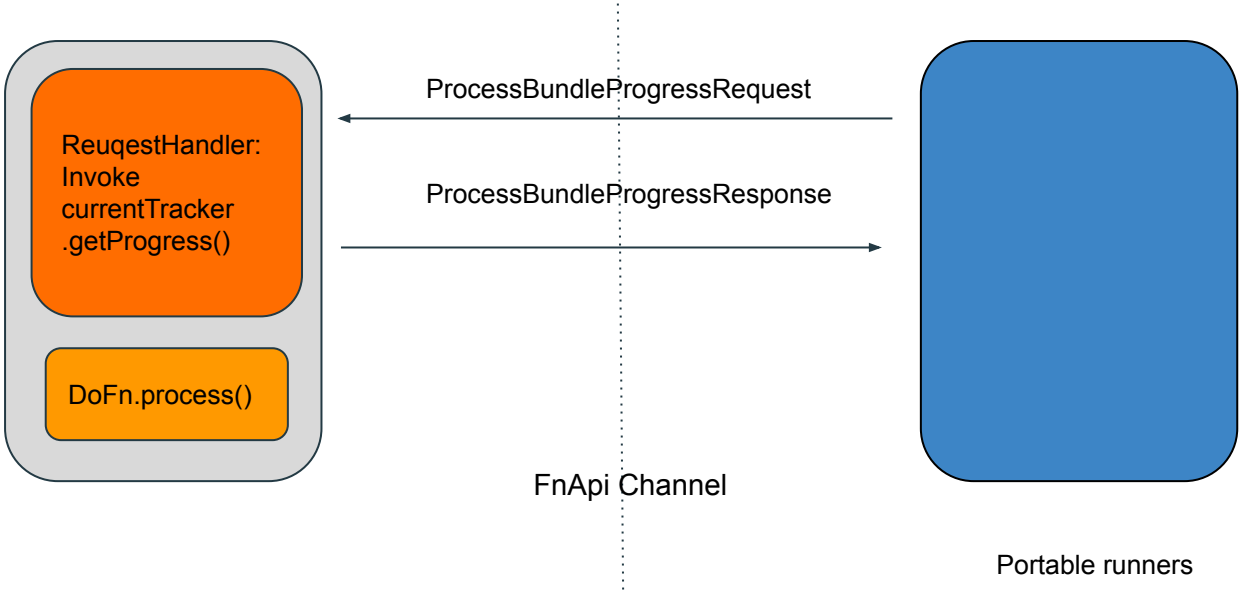
KafkaIO in Splittable DoFn: GrowableOffsetRangeTracker

```java
@Override
public Progress getProgress() {
  // If current tracking range is no longer growable, get progress as a normal range.
  if (range.getTo() != Long.MAX_VALUE || range.getTo() == range.getFrom()) {
    return super.getProgress();
  }

  // Convert to BigDecimal in computation to prevent overflow, which may result in lost of
  // precision.
  BigDecimal estimateRangeEnd = BigDecimal.valueOf(rangeEndEstimator.estimate());

  if (lastAttemptedOffset == null) {
    return Progress.from(
        0,
        estimateRangeEnd
            .subtract(BigDecimal.valueOf(range.getFrom()), MathContext.DECIMAL128)
            .max(BigDecimal.ZERO)
            .doubleValue());
  }

  BigDecimal workRemaining =
      estimateRangeEnd
          .subtract(BigDecimal.valueOf(lastAttemptedOffset), MathContext.DECIMAL128)
          .max(BigDecimal.ZERO);
  BigDecimal totalWork =
      estimateRangeEnd
          .max(BigDecimal.valueOf(lastAttemptedOffset))
          .subtract(BigDecimal.valueOf(range.getFrom()), MathContext.DECIMAL128);
  return Progress.from(
      totalWork.subtract(workRemaining, MathContext.DECIMAL128).doubleValue(),
      workRemaining.doubleValue());
}
```

```java
@Override
public long getSplitBacklogBytes() {
  long backlogBytes = 0;

  for (PartitionState p : partitionStates) {
    long pBacklog = p.approxBacklogInBytes();
    if (pBacklog == UnboundedReader.BACKLOG_UNKNOWN) {
      return UnboundedReader.BACKLOG_UNKNOWN;
    }
    backlogBytes += pBacklog;
  }

  return backlogBytes;
}
```

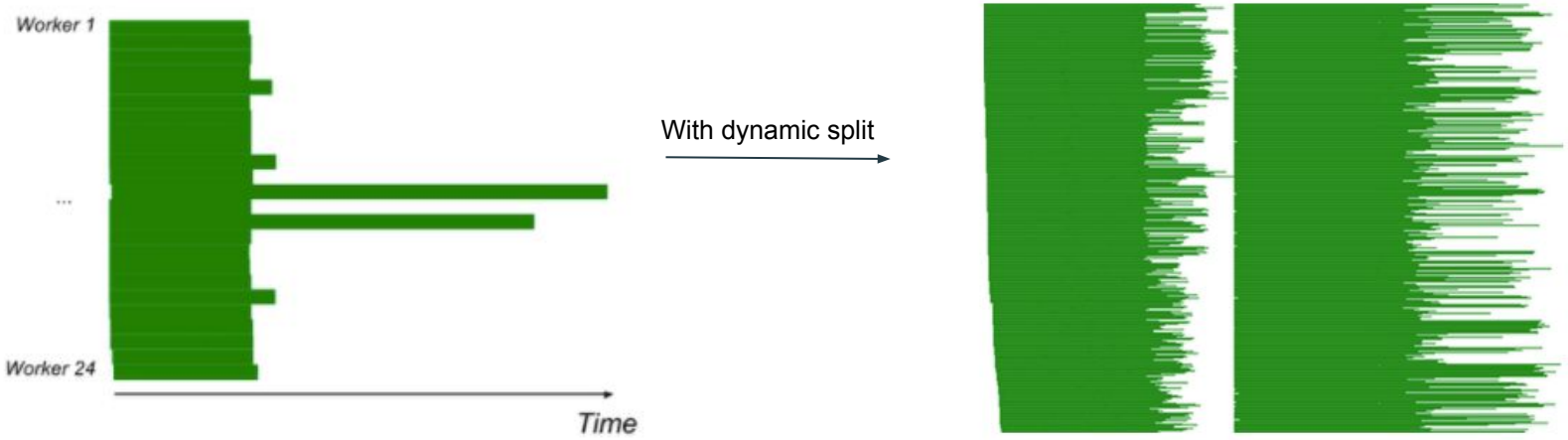
How Splittable DoFn reports progress

- Splittable DoFn reports progress over FnApi



Batch: Dynamic Split

- Dynamic split means:
  - Split current element into primary and residual while processing this element
  - Runners reschedule another instance to process this residual

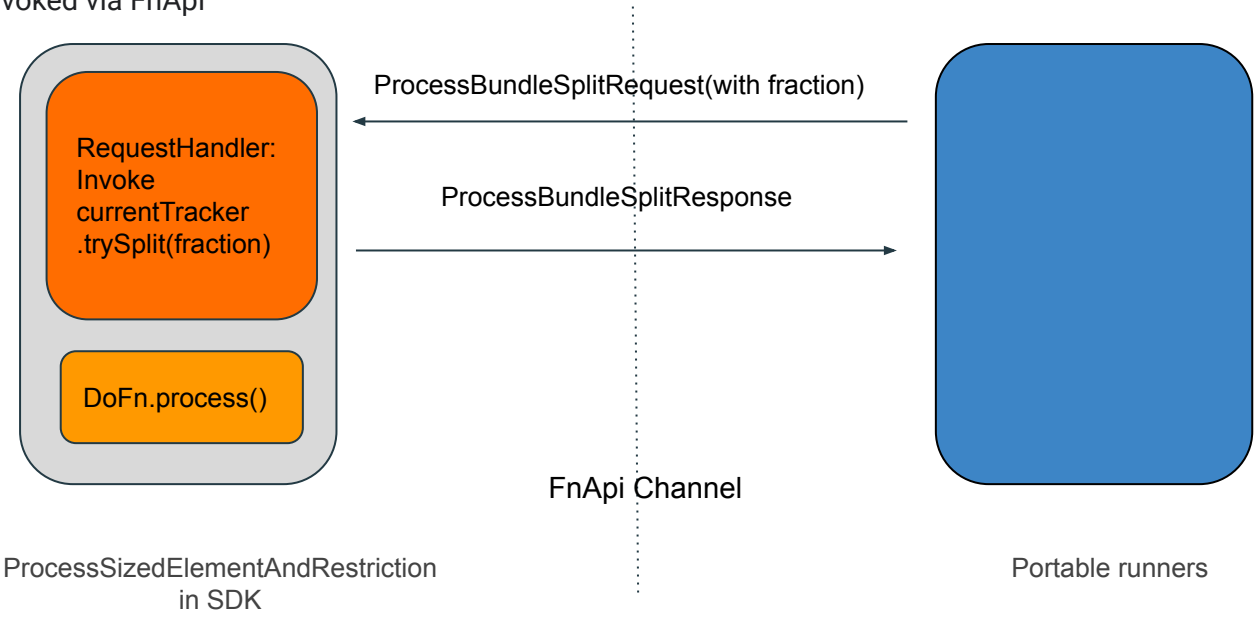


Batch: Dynamic Split

- APIs
  - Splittable DoFn: RestrictionTracker.trySplit(fractionOdRemainder)
  - BoundedSource: BoundedReader.splitAtFraction(fraction)

How Dynamic Split is invoked by Splittable DoFn

- Split is invoked via FnApi



Streaming: Checkpoint

- Streaming checkpoint follow the same code path as Batch dynamic split in Splittable DoFn
  - fractionOfRemainder is always 0.0
- APIs
  - Splittable DoFn: RestrictionTracker.trySplit(fractionOfRemainder = 0)
  - UnboundedSource: UnboundedSource.getCheckpointMark()

Streaming: Checkpoint

KafkaIO in Splittable DoFn:

```
@Override
public SplitResult<OffsetRange> trySplit(double fractionOfRemainder) {
  // If current tracking range is no longer growable, split it as a normal range.
  if (range.getTo() != Long.MAX_VALUE || range.getTo() == range.getFrom()) {
    return super.trySplit(fractionOfRemainder);
  }
  // If current range has been done, there is no more space to split.
  if (lastAttemptedOffset != null && lastAttemptedOffset == Long.MAX_VALUE) {
    return null;
  }
  BigDecimal cur =
      (lastAttemptedOffset == null)
          ? BigDecimal.valueOf(range.getFrom()).subtract(BigDecimal.ONE, MathContext.DECIMAL128)
          : BigDecimal.valueOf(lastAttemptedOffset);

  // Fetch the estimated end offset. If the estimated end is smaller than the next offset, use
  // the next offset as end.
  BigDecimal estimateRangeEnd =
      BigDecimal.valueOf(rangeEndEstimator.estimate())
          .max(cur.add(BigDecimal.ONE, MathContext.DECIMAL128));

  // Convert to BigDecimal in computation to prevent overflow, which may result in loss of
  // precision.
  // split = cur + max(1, (estimateRangeEnd - cur) * fractionOfRemainder)
  BigDecimal splitPos =
      cur.add(
          estimateRangeEnd
              .subtract(cur, MathContext.DECIMAL128)
              .multiply(BigDecimal.valueOf(fractionOfRemainder), MathContext.DECIMAL128)
              .max(BigDecimal.ONE),
          MathContext.DECIMAL128);
  long split = splitPos.longValue();
  if (split > estimateRangeEnd.longValue()) {
    return null;
  }
  OffsetRange res = new OffsetRange(split, range.getTo());
  this.range = new OffsetRange(range.getFrom(), split);
  return SplitResult.of(range, res);
}
```

KafkaIO in UnboundedSource

```
@Override
public CheckpointMark getCheckpointMark() {
  reportBacklog();
  return new KafkaCheckpointMark(
      partitionStates.stream()
          .map(
              p ->
                  new PartitionMark(
                      p.topicPartition.topic(),
                      p.topicPartition.partition(),
                      p.nextOffset,
                      p.lastWatermark.getMillis()))
          .collect(Collectors.toList()),
      source
          .getSpec()
          .isCommitOffsetsInFinalizeEnabled() ? Optional.of(this) : Optional.empty());
}
```

Streaming: Finalize Checkpoint

- For some sources used in streaming, after outputting certain amount of records, we want to ack to that source to perform certain clean up
  - Kafka, Pubsub
- APIs:
  - UnboundedSource: CheckpointMark.finalizeCheckpointMark()
  - Splittable DoFn: Bundle Finalization, or you can build transforms for such purpose

Streaming: Finalize Checkpoint

KafkaIO in Splittable DoFn: KafkaCommitOffset

```java
@Override
public PCollection<Void> expand(PCollection<KV<KafkaSourceDescriptor, KafkaRecord<K, V>>> input) {
  try {
    return input
        .apply(
            MapElements.into(new TypeDescriptor<KV<KafkaSourceDescriptor, Long>>() {})
                .via(element -> KV.of(element.getKey(), element.getValue().getOffset())))
        .setCoder(
            KvCoder.of(
                input
                    .getPipeline()
                    .getSchemaRegistry()
                    .getSchemaCoder(KafkaSourceDescriptor.class),
                VarLongCoder.of()))
        .apply(Window.into(FixedWindows.of(Duration.standardMinutes(5))))
        .apply(Max.longsPerKey())
        .apply(ParDo.of(new CommitOffsetDoFn(readSourceDescriptors)))
        .setCoder(VoidCoder.of());
  } catch (NoSuchSchemaException e) {
    throw new RuntimeException(e.getMessage());
  }
}
}
```

Splittable DoFn in UnboundedSource

```java
@Override
public void finalizeCheckpoint() {
  reader.ifPresent(r -> r.finalizeCheckpointMarkAsync(this));
  // Is it ok to commit asynchronously, or should we wait till this (or newer) is committed?
  // Often multiple marks would be finalized at once, since we only need to finalize the latest,
  // it is better to wait a little while. Currently maximum delay is same as KAFKA_POLL_TIMEOUT
  // in the reader (1 second).
}
```

Streaming: Finalize Checkpoint

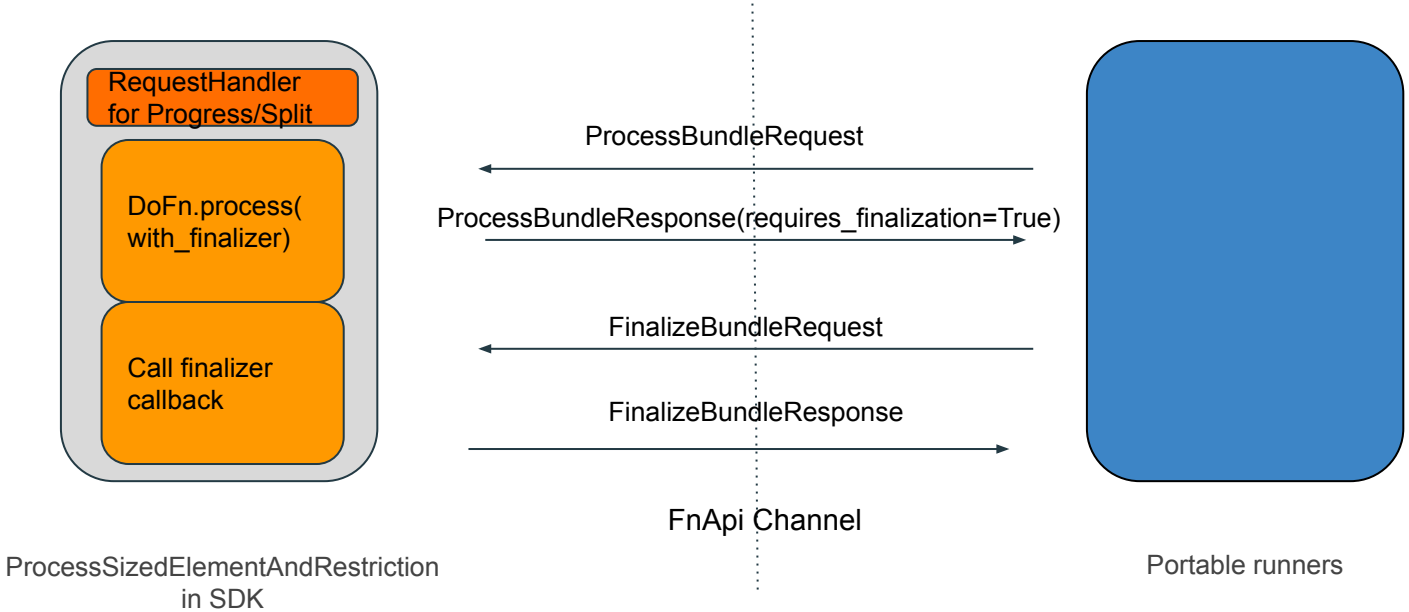
- An Example of using Bundle Finalization

```
@ProcessElement
public ProcessContinuation processElement(
    RestrictionTracker<OffsetRange, OffsetByteProgress> tracker,
    @Element SubscriptionPartition subscriptionPartition,
    OutputReceiver<SequencedMessage> receiver,
    BundleFinalizer finalizer)
    throws Exception {
  try (SubscriptionPartitionProcessor processor =
      processorFactory.newProcessor(subscriptionPartition, tracker, receiver)) {
    processor.start();
    ProcessContinuation result = processor.waitForCompletion(maxSleepTime);
    processor
        .lastClaimed()
        .ifPresent(
            lastClaimedOffset ->
                finalizer.afterBundleCommit(
                    Instant.ofEpochMilli(Long.MAX_VALUE),
                    () -> {
                      Committer committer = committerFactory.apply(subscriptionPartition);
                      committer.startAsync().awaitRunning();
                      // Commit the next-to-deliver offset.
                      committer.commitOffset(Offset.of(lastClaimedOffset.value() + 1)).get();
                      committer.stopAsync().awaitTerminated();
                    }));
    return result;
  }
}
```

Specify using Bundle Finalization

Give the callback

How finalization is invoked



Splittable DoFn self-checkpoint

- This is a new way of resume processing introduced by Splittable DoFn
  - BoundedSource/UnboundedSource don't have such ability
  - It's extremely useful when in streaming processing
    - For example: Reading records from Kafka TopicPartition
- APIs:
  - Return ProcessContinuation.resume() in @ProcessElement function

Splittable DoFn self-checkpoint

KafkaIO in Splittable DoFn

```
@ProcessElement
public ProcessContinuation processElement(
    @Element KafkaSourceDescriptor kafkaSourceDescriptor,
    RestrictionTracker<OffsetRange, Long> tracker,
    WatermarkEstimator watermarkEstimator,
    OutputReceiver<KV<KafkaSourceDescriptor, KafkaRecord<K, V>>> receiver) {
    while (true) {
      rawRecords = consumer.poll(KAFKA_POLL_TIMEOUT.getMillis());
      // When there are no records available for the current TopicPartition, self-checkpoint
      // and move to process the next element.
      if (rawRecords.isEmpty()) {
        return ProcessContinuation.resume();
      }
      for (ConsumerRecord<byte[], byte[]> rawRecord : rawRecords) {
        if (!tracker.tryClaim(rawRecord.offset())) {
          return ProcessContinuation.stop();
        }
        KafkaRecord<K, V> kafkaRecord =
            new KafkaRecord<>(
                rawRecord.topic(),
                rawRecord.partition(),
                rawRecord.offset(),
                ConsumerSpEL.getRecordTimestamp(rawRecord),
                ConsumerSpEL.getRecordTimestampType(rawRecord),
                ConsumerSpEL.hasHeaders() ? rawRecord.headers() : null,
                ConsumerSpEL.deserializeKey(keyDeserializerInstance, rawRecord),
                ConsumerSpEL.deserializeValue(valueDeserializerInstance, rawRecord));
        expectedOffset = rawRecord.offset() + 1;
        receiver.outputWithTimestamp(KV.of(kafkaSourceDescriptor, kafkaRecord), outputTimestamp);
      }
    }
  }
}
```

Resume processing current TopicPartition

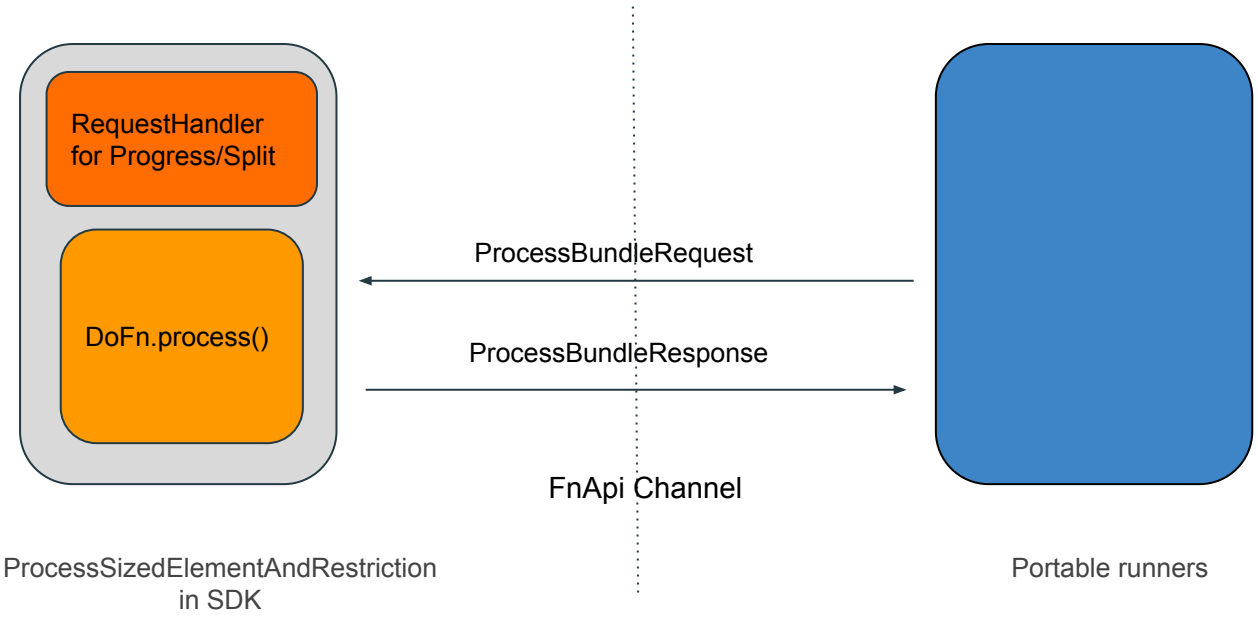
How runners get self-checkpoint results

- Residuals fl FnApi by SDK harness

```
message ProcessBundleResponse {
  // (Optional) Specifies that the bundle has not been completed and the
  // following applications need to be scheduled and executed in the future.
  // A runner that does not yet support residual roots MUST still check that
  // this is empty for correctness.
  repeated DelayedBundleApplication residual_roots = 2;

  // DEPRECATED (Required) The list of metrics or other MonitoredState
  // collected while processing this bundle.
  repeated org.apache.beam.model.pipeline.v1.MonitoringInfo monitoring_infos = 3;

  // (Optional) Specifies that the runner must callback to this worker
  // once the output of the bundle is committed. The Runner must send a
  // FinalizeBundleRequest with the instruction id of the ProcessBundleRequest
  // that is related to this ProcessBundleResponse.
  bool requires_finalization = 4;

  // An identifier to MonitoringInfo.payload mapping.
  //
  // An SDK can report metrics using an identifier that only contains the
  // associated payload. A runner who wants to receive the full metrics
  // information can request all the monitoring metadata via a
  // MonitoringInfosMetadataRequest providing a list of ids as necessary.
  //
  // The SDK is allowed to reuse the identifiers across multiple bundles as long
  // as the MonitoringInfo could be reconstructed fully by overwriting its
  // payload field with the bytes specified here.
  map<string, bytes> monitoring_data = 5;

  reserved 1;
}
```

Encoded residuals

How runners get self-checkpoint results



Streaming: Watermark

- APIs:
  - Splittable DoFn
    - WatermarkEstimator
      - WallTime
      - MonotonicallyIncreasing
      - Manual
    - @NewWatermarkEstimator to specify using which WatermarkEstimator
  - UnboundedSource: UnboundedReader.getWatermark()

Streaming: Watermark

```
@Override
public Instant getWatermark() {

  if (source.getSpec().getWatermarkFn() != null) {
    // Support old API which requires a KafkaRecord to invoke watermarkFn.
    if (curRecord == null) {
      LOG.debug("{}: getWatermark() : no records have been read yet.", name);
      return initialWatermark;
    }
    return source.getSpec().getWatermarkFn().apply(curRecord);
  }

  // Return minimum watermark among partitions.
  return partitionStates
      .stream()
      .map(PartitionState::updateAndGetWatermark)
      .min(Comparator.naturalOrder())
      .get();
}
```

Streaming: Watermark

KafkaIO in Splittable DoFn

- Create WatermarkEstimator
  - We are using both ManualWatermarkEstimator and MonitonicallyIncreasing WatermarkEstimator here

```
@NewWatermarkEstimator
public WatermarkEstimator<Instant> newWatermarkEstimator(
    @WatermarkEstimatorState Instant watermarkEstimatorState) {
  return createWatermarkEstimatorFn.apply(ensureTimestampWithinBounds(watermarkEstimatorState));
}
```

Streaming: Watermark

KafkaIO in Splittable DoFn

- Update WatermarkEstimator

```java
@ProcessElement
public ProcessContinuation processElement(
    @Element KafkaSourceDescriptor kafkaSourceDescriptor,
    RestrictionTracker<OffsetRange, Long> tracker,
    WatermarkEstimator watermarkEstimator,
    OutputReceiver<KV<KafkaSourceDescriptor, KafkaRecord<K, V>>> receiver) {
    // If there is a timestampPolicyFactory, create the TimestampPolicy for current
    // TopicPartition.
    TimestampPolicy timestampPolicy = null;
    if (timestampPolicyFactory != null) {
      timestampPolicy =
          timestampPolicyFactory.createTimestampPolicy(
              kafkaSourceDescriptor.getTopicPartition(),
              Optional.ofNullable(watermarkEstimator.currentWatermark()));
    }
    ConsumerSpEL.evaluateAssign(
        consumer, ImmutableList.of(kafkaSourceDescriptor.getTopicPartition()));
    long startOffset = tracker.currentRestriction().getFrom();
    consumer.seek(kafkaSourceDescriptor.getTopicPartition(), startOffset);
    ConsumerRecords<byte[], byte[]> rawRecords = ConsumerRecords.empty();
    while (true) {
      rawRecords = consumer.poll(KAFKA_POLL_TIMEOUT.getMillis());
      for (ConsumerRecord<byte[], byte[]> rawRecord : rawRecords) {
        if (!tracker.tryClaim(rawRecord.offset())) {
          return ProcessContinuation.stop();
        }
        KafkaRecord<K, V> kafkaRecord = rawRecordsToKafkaRecord(rawRecords);
        Instant outputTimestamp;
        // The outputTimestamp and watermark will be computed by timestampPolicy, where the
        // WatermarkEstimator should be a manual one.
        if (timestampPolicy != null) {
          checkState(watermarkEstimator instanceof ManualWatermarkEstimator);
          TimestampPolicyContext context =
              new TimestampPolicyContext(
                  (long) ((HasProgress) tracker).getProgress().getWorkRemaining(), Instant.now());
          outputTimestamp = timestampPolicy.getTimestampForRecord(context, kafkaRecord);
          ((ManualWatermarkEstimator) watermarkEstimator)
              .setWatermark(ensureTimestampWithinBounds(timestampPolicy.getWatermark(context)));
        } else {
          outputTimestamp = extractOutputTimestampFn.apply(kafkaRecord);
        }
        receiver.outputWithTimestamp(KV.of(kafkaSourceDescriptor, kafkaRecord), outputTimestamp);
      }
    }
  }
}
```

Update ManualWatermarkEstimator

Otherwise it's using MonotonicallyIncreasing WatermarkEstimator SDK harness will update it with outputTimestamp

How Splittable DoFn reports watermark

- Splittable DoFn reports watermark from
  - ProcessBundleResponse.DelayedBundleApplication if self-checkpoint happens
  - ProcessBundleSplitResponse.DelayedBundleApplication

```
// An Application should be scheduled for execution after a delay.
// Either an absolute timestamp or a relative timestamp can represent a
// scheduled execution time.
message DelayedBundleApplication {
  // (Required) The application that should be scheduled.
  BundleApplication application = 1;

  // Recommended time delay at which the application should be scheduled to
  // execute by the runner. Time delay that equals 0 may be scheduled to execute
  // immediately. The unit of time delay should be microsecond.
  google.protobuf.Duration requested_time_delay = 2;
}
```

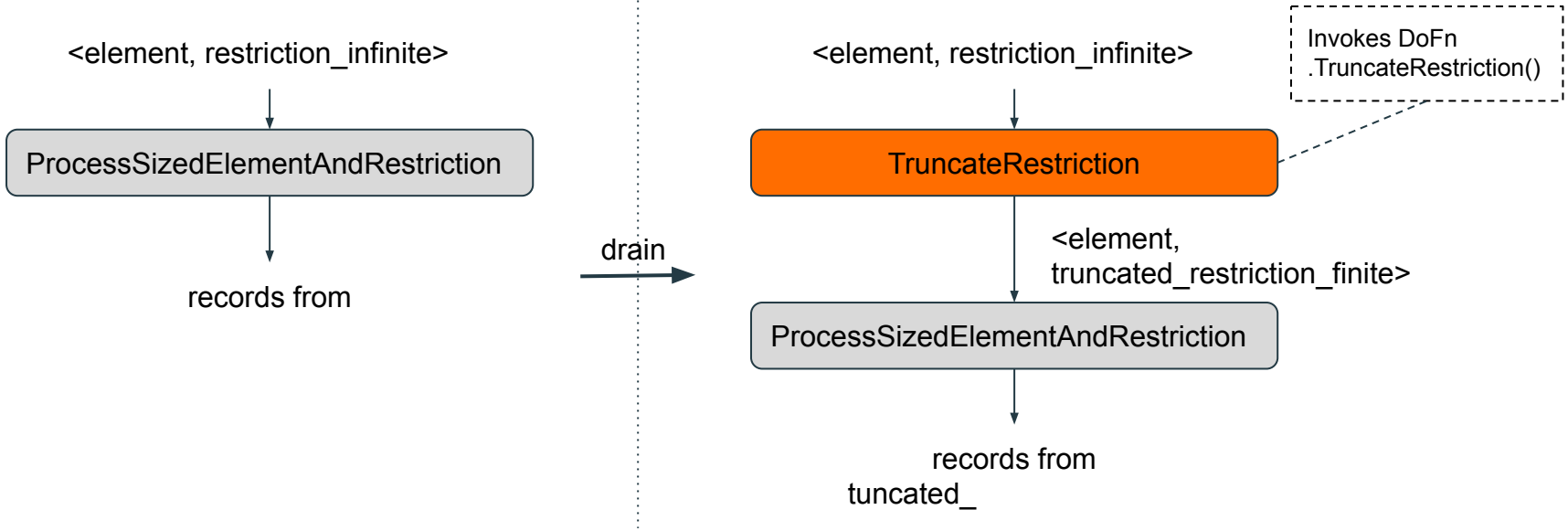
```
message BundleApplication {
  // (Required) The transform to which to pass the element
  string transform_id = 1;

  // (Required) Name of the transform's input to which to pass the element.
  string input_id = 2;

  // (Required) The encoded element to pass to the transform.
  bytes element = 3;  ← Encoded <<element, restriction>, size>

  // The map is keyed by the local output name of the PTransform. Each
  // value represents a lower bound on the timestamps of elements that
  // are produced by this PTransform into each of its output PCollections
  // when invoked with this application.
  //
  // If there is no watermark reported from RestrictionTracker, the runner will
  // use MIN_TIMESTAMP by default.
  map<string, google.protobuf.Timestamp> output_watermarks = 4;  ← watermark

  // Whether this application potentially produces an unbounded
  // amount of data. Note that this should only be set to BOUNDED if and
  // only if the application is known to produce a finite amount of output.
  org.apache.beam.model.pipeline.v1.IsBounded.Enum is_bounded = 5;
}
```

Streaming: Drain

- Dataflow supports drain the streaming pipeline
  - Different from cancel directly, drain allows to continue processing in-flight data till the end.
- APIs
  - Splittable DoFn: @TruncateRestriction
    - Truncate the infinite restriction into finite one
    - Given the source author the ability to configure what the source should do when draining
  - UnboundedSource: Runners stop reading immediately once drain starts

How Drain is performed on Splittable DoFn

- Runners run a new expansion when it's time to drain



Advanced usages

- Deduplication in streaming
- Take advantages of Splittable DoFn bassed source
    - Dynamic Read
- Cross-language Transform

Resources and Welcome to Beam Community!

- Splittable DoFn programming guide: https://beam.apache.org/documentation/programming-guide/#splittable-dofns
- Splittable DoFn x Runners capability matrix: https://beam.apache.org/documentation/runners/capability-matrix/
- Dynamic split: https://cloud.google.com/blog/products/gcp/no-shard-left-behind-dynamic-work-rebalancing-in-google-cloud-dataflow
- Beam FnApi: https://s.apache.org/beam-fn-api
- Beam I/O guide: https://beam.apache.org/documentation/io/developing-io-overview/
- Beam design documentations: https://cwiki.apache.org/confluence/display/BEAM/Design+Documents
- Contribution guide: https://beam.apache.org/contribute/
- Contact us: dev@apache.beam.org