Leveraging Beam’s Batch-Mode for Robust Recoveries and Late-Data Processing of Streaming Pipelines

Devon Peticolas - Oden Technologies

I regret how long this talk’s name is...
Devon Peticolas
Principal Engineer
Devon Peticolas
Principal Engineer
Jie Zhang
Sr. Data Engineer
In This Talk

- Why does Oden need batch recoveries for our streaming jobs?
- How we make our streaming jobs also run in batch mode.
- How we orchestrate our streaming pipeline to run in batch-mode daily.
A little about Oden
Oden’s Customers

Medium to large manufacturers in plastics extrusion, injection molding, and pipes, chemical, paper and pulp.

Process and Quality Engineers looking to centralize, analyze, and act on their data.

Plant managers who are looking to optimize logistics, output, and cost.
Interactive Time-series Analysis

- Compare performance across different equipment.
- Visualize hourly uptime and key custom metrics.
- Calculations for analyzing and optimizing factory performance.
Real Time Manufacturing Data

- Streaming second-by-second metrics
- Interactive app that prompts on production state changes and collects user input.
Reporting and Alerting

- Daily summaries on key process metrics from continuous intervals of production work.
- Real-time email and text alerts on target violations and concerning trends.
How we use Apache Beam and misuse
How Oden Uses Apache Beam - Reading Events
How Oden Uses Apache Beam - Acquisition
How Oden Uses Apache Beam - Acquisition
User Has: \texttt{diameter-x} and \texttt{diameter-y}

User Wants: 
\texttt{avg-diameter} = \left(\frac{\texttt{diameter-x} + \texttt{diameter-y}}{2}\right)

- Metrics need to be computed in real-time.
- Components can be read from different devices with different clocks.
- Formulas are stored in postgres.
How Oden Uses Apache Beam - Calculated Metrics

PLC → Acquisition → Raw Events → Metrics-A → Calculated Metrics

Metrics-B → Windowed Calculated Metrics
How Oden Uses Apache Beam - State Change Detection
How Oden Uses Apache Beam - State Change Detection

PLC

Raw Events → Acquisition → Metrics-A → Calculated Metrics

Windowed Calculated Metrics

State Change Detection

Postgres
How Oden Uses Apache Beam - Rollups

- Creates special “rollup” metrics before writing to TSDB
- Optimizes aggregations i.e.
  - $\text{sum([t6...t16])} = \text{sum([t6...t8])} + \text{sum([t8...t12])} + \text{sum([t12...t16])} + \text{val(t16)}$
  - $\text{count([t6...t16])} = \text{count([t6...t8])} + \text{count([t8...t12])} + \text{count([t12...t16])} + 1$
  - $\text{mean([t6...t16])} = \text{sum([t6...t16])} / \text{count([t6...t16])}$
How Oden Uses Apache Beam - Rollups

- PLC
- Acquisition
- Raw Events
- Metrics-A
- Calculated Metrics
- Windowed Calculated Metrics
- Metrics-B
- State Change Detection
- Metrics-C
- Rollups
- TSDB
- Postgres
How Oden Uses Apache Beam - In Summary

PLC

Acquisition

Raw Events

Metrics-A

Calculated Metrics

Windowed Calculated Metrics

Metrics-B

Rollups

State Change Detection

Metrics-C

TSDB

Postgres
How Oden Uses Apache Beam - In Summary

**PLC**

**Acquisition**
- Raw Events
- Flaming Rainbow Bridge to Aegard
- Metrics
- Calculated Metrics
- Windowed Calculated Metrics (60s)
- API that uses regex as a language parser with terrifying effectiveness

**Calculated Metrics**
- We built this to close a deal and now I'm not sure if we can kill it
- Windowed Calculated Metrics
- Postgres
- Definitely a waste of money

**Rollups** (600s)

**New TSDB**

**TSDB**
- Also BigQuery because we don't trust our DS team with either TSDB
- Google-provided Dataflow Template that DROPs DATA when redeployed
- Probably a waste of money
- Also Metrics
- State Change Detection
- Flaming Rainbow Bridge to Asgard

**Reality**
- Everytime I remember this exists I get sad
- We built this to close a deal and now I'm not sure if we can kill it
- Definitely a waste of money
- Also BigQuery because we don't trust our DS team with either TSDB
- New TSDB
- TSDB
● Downstream writes are idempotent.
● LOTS of windowing keyed by the metric.
● Real-time processing is needed for users to make real-time decisions.
● Downstream writes are idempotent.
● LOTS of windowing keyed by the metric.
● Real-time processing is needed for users to make real-time decisions.
Connectivity for factories is hard
Connectivity in factories is hard
How Oden Uses Apache Beam - Disconnects

- PLC
- Raw Events
- Acquisition
- Metrics-A
- Calculated Metrics
- Windowed Calculated Metrics
- Metrics-B
- Rollups
- State Change Detection
- Metrics-C
- TSDB
- Postgres

Undelivered Data Backs Up
How Oden Uses Apache Beam - Disconnects

Downstream jobs are flooded with late events
How Oden Uses Apache Beam - Disconnects

Raw Events → Acquisition → Metrics-A → Calculated Metrics → Metrics-B → Rollups → Metrics-C → TSDB

PLC

One bad apple ruins the bunch
Attempts at Late Data Handling - Complex Triggering Logic

/* The Window described attempts to both be prompt but not needlessly retrigger. It's designed to account for the
following cases...

* <ul><li>All data is coming on-time. The watermark at any given time is roughly the current time.
<li>Data is being backfilled from some subset of metrics and the watermark is ahead of the event time of the windows
for those metrics.
<li>Data is being backfilled for some subset of metrics but the watermark has been stuck to be earlier than than
event time for most metrics.
<li>Any of cases 1, 2, or 3 but where late data has arrived due to some uncontrollable situation (i.e. a single
metric for a pane gets stuck in pubsub for days and then is released).</ul> */

public static <T> Window<T> earlyAndLateFireSlidingWindow(Duration windowSize, Duration windowSlide, Duration earlyFire, Duration allowedLateness, Duration offset) {
  return Window.<T>into(
    SlidingWindows.of(windowSize)
      .every(windowSlide)
      // In sliding windows, with a configurable window size plus a buffer (default at 0) on the end to provide
      // space for calculating the last deltasum value (rollups). We add a offset (default at 0), which moves the
      // window forward [start+offset, end+offset) to align with Heroic's exclusive start and inclusive end.
      // .withOffset(windowSize.minus(deltasumBuffer).plus(offset)))
      .withOffset(offset))
    // This sliding window will fire (materialize the accumulated data) at least once. Each time we do we'll fire
    // with the accumulated data in the window so far (as opposed to just the new data since the last fire).
    .accumulatingFiredPanes()
    .triggering(
      // The primary way that this window will fire is when the watermark (tracked upstream as the estimated
      // minimum of the backlog) exceeds the end of the window. This is the only firing behavior for case 1 and
      // the first firing behavior for cases 2 and 4.
      AfterWatermark.pastEndOfWindow()
      // In case 3, we don't want the user to have to wait until the watermark has caught up to get their data
      // so we have a configurable threshold that will allow the window to fire early based on how much time
      // has passed since the first element we saw in the pane.
      .withEarlyFirings(AfterProcessingTime.pastFirstElementInPane().plusDelayOf(earlyFire))
      // In case 2, all elements are considered "late". And we don't want to excessively fire once for every
      // element that gets added to the pane (i.e. 300 times for a 5 minute window). So, instead, we only late
      // fire when new elements enter and the window's time has passed in process time. The assumption here is
      // that backfilling a pane is, typically, faster than on-time filling. This introduces a small, but
      // acceptable, lag in case 4.
      .withLateFirings(
        AfterAll.of(
          AfterPane.elementCountAtLeast(1),
          AfterProcessingTime.pastFirstElementInPane().plusDelayOf(windowSize))))
    // When accounting for case 3, after the watermark has caught up, the default behavior would be to fire the
    // window again. This changes that behavior to only fire if any new data has arrived between the early fire and
    // the on-time fire.
    .withOnTimeBehavior(OnTimeBehavior.FIRE_IF_NON_EMPTY)
    // This sets the duration we will retain the panes and accept late data in event time.
    .withAllowedLateness(allowedLateness).
  )
}
Attempts at Late Data Handling - Complex Triggering Logic

/* The Window described attempts to both be prompt but not needlessly retrigger. It’s designed to account for the following cases:
   * All data is coming on-time. The watermark at any given time is roughly the current time.
   * Data is being backfilled from some subset of metrics and the watermark is ahead of the event time of the windows for those metrics.
   * Data is being backfilled for some subset of metrics but the watermark has been stuck to be earlier than the event time for most metrics.
   * Any of cases 1, 2, or 3 but where late data has arrived due to some uncontrollable situation (i.e. a single metric for a pane gets stuck in pubsub for days and then is released). */

public static <T> Window<T> earlyAndLateFireSlidingWindow(Duration windowSize, Duration windowSlide, Duration earlyFire, Duration allowedLateness, Duration offset) {
    return Window.<T>into(
        SlidingWindows.of(windowSize)
            .every(windowSlide)
            // In sliding windows, with a configurable window size plus a buffer (default at 0) on the end to provide space for calculating the last delta sum value (rollups). We addColor a offset (default at 0), which moves the window forward [start-offset, end-offset) to align with Heroic’s exclusive start and inclusive end.
            .withOffset(windowSize.minus(deltasumBuffer).plus(offset)))
            .withOffset(offset))
        // This sliding window will fire (materialize the accumulated data) at least once. Each time we do we’ll fire
        // with the accumulated data in the window so far (as opposed to just the new data since the last fire).
        .accumulatingFiredPanes()
        .triggering(
            // The primary way that this window will fire is when the watermark (tracked upstream as the estimated minimum of the backlog) exceeds the end of the window. This is the only firing behavior for case 1 and
            AfterWatermark.pastEndOfWindow()
            // In case 3, we don’t want the user to have to wait until the watermark has caught up to get their data
            // so we have a configurable threshold that will allow us to fire early based on how much time
            // has passed since the first element we saw in the pane.
            .withEarlyFirings(AfterProcessingTime.pastFirstElementInPane().plusDelayOf(earlyFire))
            // In case 2, all elements are considered “late”. And we don’t want to excessively fire once for every element that gets added to the pane (i.e. 300 times for a 5 minute window). So, instead, we only late
            // fire when new elements enter and the window’s time has passed in process time. The assumption here is
            // that backfilling a pane is, typically, faster than on-time filling. This introduces a small, but
            // acceptable, lag in case 4.
            .withLateFirings(
                AfterAll.of(
                    AfterPane.elementCountAtLeast(1),
                    AfterProcessingTime.pastFirstElementInPane().plusDelayOf(windowSize)))))
            // When accounting for case 3, after the watermark has caught up, the default behavior would be to fire the
            // window again. This changes that behavior to only fire if any new data has arrived between the early fire and
            // the on-time fire.
            .withOnTimeBehavior(OnTimeBehavior.FIRE_IF_NON_EMPTY)
            // This sets the duration we will retain the panes and accept late data in event time.
            .withAllowedLateness(allowedLateness));
}
Attempts at Late Data Handling - Complex Triggering Logic

```java
public class GroupByCalcMetricIDandTimestampDoFn extends DoFn<KV<String, Metric>, KV<String, List<Metric>>> {
    @StateId("discardingWindow")
    private final StateSpec<ConcurrentSkipListMap<Long, List<Metric>>> window = StateSpecs.value();

    public void process(
        @Element KV<String, Metric> element,
        @StateId("discardingWindow") ValueState<ConcurrentSkipListMap<Long, List<Metric>>> windowState,
        ProcessContext c) {
        Metric metric = element.getValue();
        Long metricTs = metric.getEventTimestampMillis();
        String calcmetricID = element.getKey();
        HashMap<String, CalcMetricHelper> helperMap = c.sideInput(definitionsView).get(0);
        CalcMetricHelper helper = c.sideInput(definitionsView).get(0).get(calcmetricID);
        if (helper.formulaUUIDToName == null) return;
        if (helper.formulaUUIDToName.size() == 1) {
            c.outputWithTimestamp(KV.of(calcmetricID, Arrays.asList(metric)), c.timestamp());
            return;
        }
        ConcurrentSkipListMap<Long, List<Metric>> currentWindow =
            MoreObjects.firstNonNull(windowState.read(), new ConcurrentSkipListMap<>());
        Map.Entry<Long, List<Metric>> lastPane = currentWindow.lastEntry();
        if (lastPane != null & lastPane.getKey() - metricTs > WINDOW_SIZE_MS)
            return;
        List<Metric> metrics = Optional.ofNullable(currentWindow.get(metricTs)).orElse(new ArrayList<Metric>())
            .stream().map(Metric::getMetricID).collect(Collectors.toSet());
        if (metricIDs.contains(metric.getMetricID()))
            return;
        metrics.add(metric);
        metricIDs.add(metric.getMetricID());
        if (helper.formulaUUIDToName.keySet().equals(metricIDs)) {
            c.outputWithTimestamp(KV.of(calcmetricID, Arrays.asList(metric)), c.timestamp());
            currentWindow.remove(metricTs);
        } else {
            currentWindow.put(metricTs, metrics);
        }
        if (currentWindow.isEmpty()) {
            return;
        }
        lastPane = currentWindow.lastEntry();
        if (lastPane != null)
            currentWindow.headMap(lastPane.getKey() - WINDOW_SIZE_MS, false).clear();
        windowState.write(currentWindow);
    }
}
```

- **ValueState<ConcurrentSkipListMap<Long, List<T>>>** to “hand-roll” window-like behavior
- Manage a window per key and manually garbage collect using the timestamps of the metrics.
- We have to disable autoscaling in Dataflow for this to work.
- Both solutions have big issues...
Attempts at Late Data Handling - Complex Triggering Logic

```java
public class GroupByCalMetricIDandTimestampDoFn extends DoFn<KV<String, Metric>, KV<String, List<Metric>>> {
  @StateId("discardingWindow")
  private final StateSpec<ValueState<ConcurrentSkipListMap<Long, List<Metric>>>> window = StateSpecs.value();

  public void process(
    @Element KV<String, Metric> element,
    @StateId("discardingWindow") ValueState<ConcurrentSkipListMap<Long, List<Metric>>> windowState,
    ProcessContext c) {
    Metric metric = element.getValue();
    Long metricTs = metric.getEventTimestampMillis();
    String calcmetricID = element.getKey();
    HashMap<String, CalcMetricHelper> helperMap = c.sideInput(definitionsView).get(0);
    CalcMetricHelper helper = c.sideInput(definitionsView).get(calcmetricID);
    if (helper.formulaUUIDToName == null)
      return;
    if (helper.formulaUUIDToName.size() == 1) {
      c.outputWithTimestamp(KV.of(calcmetricID, Arrays.asList(metric)), c.timestamp());
      return;
    }
    ConcurrentSkipListMap<Long, List<Metric>> currentWindow =
      MoreObjects.firstNonNull(windowState.read(), new ConcurrentSkipListMap<>());
    Map.Entry<Long, List<Metric>> lastPane = currentWindow.lastEntry();
    if (lastPane != null && lastPane.getKey() - metricTs > WINDOW_SIZE_MS)
      return;
    List<Metric> metrics = Optional.ofNullable(currentWindow.get(metricTs)).orElse(new ArrayList<Metric>());
    if (helper.formulaUUIDToName.keySet().equals(metric.getMetricID()))
      c.outputWithTimestamp(KV.of(calcmetricID, metrics), c.timestamp());
    currentWindow.remove(metricTs);
    else {
      currentWindow.put(metricTs, metrics);
    }
    if (currentWindow.isEmpty())
      return;
    lastPane = currentWindow.lastEntry();
    if (lastPane != null)
      currentWindow.headMap(lastPane.getKey() - WINDOW_SIZE_MS, false).clear();
    windowState.write(currentWindow);
  }
}
```

- **ValueState<ConcurrentSkipListMap<Long, List<T>>> to “hand-roll” window-like behavior**
- **Manage a window per key and manually garbage collect using the timestamps of the metrics.**
- **We have to disable autoscaling in Dataflow for this to work.**
- **Both solutions have big issues...**

No link, this is bad code

Even more crazy
How Oden Uses Apache Beam - It all fell apart

2019-10-18
Pager Duty Incident: 2586

THERE MUST BE A BETTER WAY
What do our users need?
Age of Data

<table>
<thead>
<tr>
<th></th>
<th>Real-time Decision Making</th>
<th>Historical Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>High Access Freq</td>
<td></td>
</tr>
<tr>
<td>Old</td>
<td>Low Access Freq</td>
<td></td>
</tr>
</tbody>
</table>

Access Freq

Access Freq decreases as the age of data increases.
Age of Data

Access Freq

Real-time Decision Making

Historical Analysis

Very Late Data

New

Age of Data

Old
Handling late data in real-time

Handling late data later
How Oden Uses Apache Beam - Late Events Capture
How Oden Uses Apache Beam - Disconnects

One bad apple ruins the bunch
How Oden Uses Apache Beam - Recovery Scripts

- Create a bunch of temporary topics and subscriptions
- Deploy “recovery” versions of the jobs listening to these topics.
- Use Google-provided dataflow templates to get data out of GCS and into PubSub topics
- Well orchestrated but very manual process.
Batch-Mode is Better
# Batch vs Streaming Jobs In Beam

<table>
<thead>
<tr>
<th>Streaming</th>
<th>Batch</th>
</tr>
</thead>
<tbody>
<tr>
<td>● “Unbounded” PCollections</td>
<td>● “Bounded” PCollections</td>
</tr>
<tr>
<td>● Generally talks to real-time queues</td>
<td>● Generally talks to files / databases</td>
</tr>
<tr>
<td>● Processed piece-by-piece</td>
<td>● Processed in stages</td>
</tr>
<tr>
<td>● More wasted worker overhead</td>
<td>● Workers run until they’re done</td>
</tr>
<tr>
<td></td>
<td>● A little cheaper *</td>
</tr>
</tbody>
</table>
Batch vs Streaming Sources/Sinks

**Streaming**
- PubSubIO
- KafkaIO

**Batch**
- JdbcIO
- BigQueryIO
- TextIO
- AvroIO
- GenerateSequence
Batch vs Streaming Sources/Sinks

**Streaming**
- PubSubIO
- KafkaIO
- `TextIO*`
- `AvroIO*`
- `GenerateSequence*`

**Batch**
- JdbcIO
- BigQueryIO
- `TextIO`
- `AvroIO`
- `GenerateSequence`
Batch vs Streaming Jobs In Beam

Streaming

Batch

Topic A

Streaming Job

Topic B

GCS A

Batch Job

GCS B
Streaming Jobs - Simple

```
PubSubIO (Read) -> Transform -> PubSubIO (Write)
```

**Source**

**Sink**
Batch Jobs - Simple

Source

TextIO (Read) → Transform → TextIO (Write)

Sink
Batch Jobs - Simple

Source: TextIO (Read)

Transform

Sink: TextIO (Write)

gs://dataflow.oden-qa.io/metric-b/2021-07-15/

gs://dataflow.oden-qa.io/metric-a/2021-07-15/metrics-*.ndjson
Batch Jobs - Simple

When the late metrics arrived

gs://dataflow.oden-qa.io/metric-a/2021-07-15/metrics-*.ndjson

gs://dataflow.oden-qa.io/metric-b/2021-07-15/
Streaming Jobs - Complex

Source

GenerateSequence.from(0).withRate(1, Duration.standardMinutes(5L)))

Sink

PubSubIO (Write)
Batch Jobs - Complex

- **Source**: GenerateSequence.from(0).to(1)
- **Internal Config API**
- **Map Elements**
- **Config PCollection**
- **Config PCollection View**
- **Side-input**
- **Transform**
- **TextIO (Read)**
- **TextIO (Write)**
Running Streaming Jobs in Batch - With if-statements

- The “core” transform is abstracted into a generic PTransform<String, String>
Pipeline pipeline = Pipeline.create(options);

if (options.getConsumerMode() == "PUBSUB") {
    pipeline.apply("ReadFromPubsub",
            PubsubIO.readStrings()
            .fromSubscription(options.getSourcePubsubSubscription())
            .withTimestampAttribute("ts");
} else {
    pipeline
            .apply("ReadFromFiles", TextIO.read().from(options.getSourceFilePattern())
            .apply("AssignEventTimestamps",
                    WithTimestamps.of((String event) -> new Instant(...))
                    .withAllowedTimestampSkew(new Duration(Long.MAX_VALUE)));
}

pipeline.apply("MakeMetricsIntoRollups",
            new RollupMetrics( // PTransform<String, String>
                    options.getWindowSize(),
                    options.getAllowedLateness(),
                    options.getDeltasumBuffer(),
                    options.getEarlyFire()));

if (options.getProducerMode() == "PUBSUB") {
    pipeline.apply("WriteToPubsub",
            PubsubIO.writeStrings()
            .to(options.getSinkPubsubTopic)
            .withTimestampAttribute("ts");
} else {
    pipeline.apply("WriteToFiles",
            TextIO.write().to(super.filenamePrefix));
}

pipeline.run();
Running Streaming Jobs in Batch - With if-statements

```
Pipeline pipeline = Pipeline.create(options);
if (options.getConsumerMode() == "PUBSUB") {
  pipeline.apply(
    "ReadFromPubsub",
    PubsubIO.readStrings()
      .fromSubscription(options.getSourcePubsubSubscription())
      .withTimestampAttribute("ts"));
} else {
  pipeline
    .apply("ReadFromFiles", TextIO.read().from(options.getSourceFilePattern())
      .apply(
        "AssignEventTimestamps",
        WithTimestamps.of((String event) -> new Instant(...))
          .withAllowedTimestampSkew(new Duration(Long.MAX_VALUE)));
}

pipeline.apply(
  "MakeMetricsIntoRollups",
  new RollupMetrics( // PTransform<String, String>
    options.getWindowSize(),
    options.getAllowedLateness(),
    options.getDeltasumBuffer(),
    options.getEarlyFire())
if (options.getProducerMode() == "PUBSUB") {
  pipeline.apply(
    "WriteToPubsub",
    PubsubIO.writeStrings()
      .to(options.getSinkPubsubTopic)
      .withTimestampAttribute("ts"));
} else {
  pipeline.apply(
    "WriteToFile",
    TextIO.write().to(super.filenamePrefix));
}
pipeline.run();
```
Running Streaming Jobs in Batch - With if-statements

```java
Pipeline pipeline = Pipeline.create(options);
if (options.getConsumerMode() == "PUBSUB") {
    pipeline.apply("ReadFromPubsub",
                  PubsubIO.readStrings()
                      .fromSubscription(options.getSourcePubsubSubscription())
                      .withTimestampAttribute("ts"));
} else {
    pipeline
        .apply("ReadFromFiles", TextIO.read().from(options.getSourceFilePattern())
             .apply("AssignEventTimestamps",
                    WithTimestamps.of((String event) -> new Instant(...))
                          .withAllowedTimestampSkew(new Duration(Long.MAX_VALUE))));
}

pipeline.apply("MakeMetricsIntoRollups",
                new RollupMetrics( // PTransform<String, String>
                                    options.getWindowSize(),
                                    options.getAllowedLateness(),
                                    options.getDeltasumBuffer(),
                                    options.getEarlyFire()));
if (options.getProducerMode() == "PUBSUB") {
    pipeline.apply("WriteToPubsub",
                  PubsubIO.writeStrings()
                      .to(options.getSinkPubsubTopic)
                      .withTimestampAttribute("ts"));
} else {
    pipeline.apply("WriteToFiles",
                TextIO.write().to(super.filenamePrefix));
}
pipeline.run();
```

- The “core” transform is abstracted into a generic PTransform<String, String>
- Control which “mode” we’re in with a build-time (not ValueProvider) option.
- When in “Pubsub Mode” the job runs streaming Pubsub to Pubsub using PubsubIO’s withTimestampAttribute to set event-time for windows and watermark.
- When in “File Mode” the job runs batch from GCS to GCS. We need to use WithTimestamps to manufally set event-time for windows and watermark.
Running Streaming Jobs in Batch - With EventIO

All source/sink handling is moved to shared transforms we call EventIO.
Running Streaming Jobs in Batch - With EventIO

```java
// public interface RollupOptions extends EventIOOptions
Pipeline pipeline = Pipeline.create(options);
pipeline
    .apply(
        "ReadPlainMetricsFromSource",
        EventIO.<Metric>readJsons().of(Metric.class).withOptions(options))
    .apply(
        new RollupMetrics(
            options.getWindowSize(),
            options.getAllowedLateness(),
            options.getDeltasumBuffer(),
            options.getEarlyFire())
    .apply(
        "WriteRollupMetricsToSink",
        EventIO.<RollupMetric>writeJsons().of(RollupMetric.class).withOptions(options));
pipeline.run();
```

- All source/sink handling is moved to shared transforms we call EventIO.

- Manages:
  - Condition “mode” switching
  - JSON serializing/deserializing
  - Event-timestamp managing
  - Source and sink job options

- Modes:
  - PubSub
  - File (GCS / local files)
  - BigQuery
  - Logging (debug)

- Makes developing locally easy!

- Get's complicated for multi-source/sink jobs and managing building templates.
Running Streaming Jobs in Batch - With EventIO

```java
// public interface RollupOptions extends EventIOOptions
Pipeline pipeline = Pipeline.create(options);
pipeline
    .apply(
        "ReadPlainMetricsFromSource",
        EventIO.<Metric>readJsons().of(Metric.class).withOptions(options))
    .apply(
        new RollupMetrics(
            options.getWindowSize(),
            options.getAllowedLateness(),
            options.getDeltasumBuffer(),
            options.getEarlyFire())
        .apply(
            "WriteRollupMetricsToSink",
            EventIO.<RollupMetric>writeJsons().of(RollupMetric.class).withOptions(options));
pipeline.run();
```

- All source/sink handling is moved to shared transforms we call EventIO.

- Manages:
  - Condition "mode" switching
  - JSON serializing/deserializing
  - Event-timestamp managing
  - Source and sink job options

- Modes:
  - PubSub
  - File (GCS / local files)
  - BigQuery
  - Logging (debug)

- **Makes developing locally easy!**

- Get's complicated for multi-source/sink jobs and managing building templates.
Automating Batch Recoveries with Airflow
Airflow - Overview

- Scheduler, orchestrator, and monitor of DAGs of tasks.
- DAGs and dependency behavior are defined in python.
- Built-in “Operators” which let you easily build tasks for popular services.
- Expressive API for common patterns such as backfilling, short-circuiting, timezone management for ETL.
Airflow - Overview

DAG: compute_and_mail_reports_v1

Graph View

Base date: 2021-08-02 14:00:01
Number of runs: 25
Run: scheduled_2021-08-02T14:00:00+00:00
Layout: Left->Right

wait_metrics_import
wait_copy_target_v2_from_postgres_to_gcs
wait_copy_event_state_from_postgres_to_gcs
compute_quality_aggregates_to_tmp_table
upsert_from_tmp_table_to_postgres
deadline_quality_reports_v2
delete_quality_tmp_table
passive_check
dag = DAG("hello_world_v1", schedule_interval="0 * * * *")

task_1 = PythonOperator(  
    task_id="python_hello",  
    python_callable=lambda: print("hello"),  
    dag=dag,  
)  
task_2 = BashOperator(  
    task_id='bash_world',  
    bash_command='echo world',  
)  

task_1 >> task_2
Airflow - Running Batch Dataflow Jobs

```
dag = DAG("late_data_v1", schedule_interval="30 * * * *")

LATE_METRICS = "gs://dataflow.oden-qa.io/metric-a/{{ds}}/
CALC_METRICS = "gs://dataflow.oden-qa.io/metric-b/{{ds}}/
RLUP_METRICS = "gs://dataflow.oden-qa.io/metric-c/{{ds}}/"

calculated_metrics_df_job = DataflowTemplatedJobStartOperator(
    task_id="calculated-metrics-df-job",
    template='gs://oden-dataflow-templates/latest/batch_calculated_metrics,
    parameters={
        'source': LATE_METRICS,
        'sink': CALC_METRICS + "metrics-*.ndjson",
    },
    dag=dag,
)

rollup_metrics_df_job = DataflowTemplatedJobStartOperator(
    task_id="rollup-metrics-df-job",
    template='gs://oden-dataflow-templates/latest/batch_rollup_metrics,
    parameters={
        'source': CALC_METRICS,
        'sink': RLUP_METRICS + "metrics-*.ndjson",
    },
    dag=dag,
)

Calculated_metrics_df_job >> rollup_metrics_df_job
```

- Batch dataflow jobs are built as templates and launched from Airflow DAG tasks
- The DAG structure mirrors the DAG of streaming dataflow jobs.
- GCS buckets are used as intermediaries.
Late Data Airflow DAG - GCS Wildcards

Airflow “execution_date” macro

\[
\begin{align*}
\text{LATE\_METRICS} &= \text{"gs://dataflow.oden-qa.io/metric-a/{{ds}}/metrics-*.ndjson"} \\
\text{CALC\_METRICS} &= \text{"gs://dataflow.oden-qa.io/metric-b/{{ds}}/metrics-*.ndjson"} \\
\text{RLUP\_METRICS} &= \text{"gs://dataflow.oden-qa.io/metric-c/{{ds}}/metrics-*.ndjson"} \\
\text{ALL\_METRICS} &= \text{"gs://dataflow.oden-qa.io/metric-[abc]/{{ds}}/metrics-*.ndjson"}
\end{align*}
\]

GCS wildcard
Streaming Pipeline

1. Raw Events → Acquisition
2. Acquisition → Metrics-A
3. Metrics-A → Calculated Metrics
4. Calculated Metrics → Windowed Calculated Metrics
5. Windowed Calculated Metrics → Metrics-B
6. Metrics-B → Rollups
7. Rollups → Metrics-C
8. Metrics-C → TSDB
9. Metrics-B → State Change Detection
10. State Change Detection → Postgres
Late Data Airflow DAG - Detecting Late Data in GCS

# Only works in >=1.10.15
check_for_late_data = GoogleCloudStoragePrefixSensor(
    dag=dag,
    task_id="check-for-late-data",
    soft_fail=True,
    timeout=60 * 2,
    bucket="gs://dataflow.oden-qa.io/metric-a/{{ds}}/",
    prefix="/metric-a/{{ds}}/"
)

...

check_for_late_data >> ...

Soft-fail causes “skipped” downstream behavior when no late data for that day
Late Data Airflow DAG - Putting it all together

```
schedule_interval="30 0 * * *"
```
Problem Recap

- Oden uses Beam to process metrics for real-time and historical use-cases
- The real-time processing was broken by factory-specific partitions
- This was partly solved by complex windowing and triggering and beam state
- Ultimately, we decided to cap “lateness” and push late data into GCS to preserve our real-time applications
Solution Recap

- All of our dataflow jobs can run in a batch or streaming mode.
- We only use streaming mode for recent data needed by our users in ASAP.
- Late data is handled with batch jobs nightly orchestrated by an Airflow DAG.
- Streaming jobs now run at a smaller deployment, autoscaling happens less frequently.
What’s next?

- Using the late-data Airflow DAG to backfill old customer data.
- Using the late-data Airflow DAG for “low-priority” metrics to save money.
- Replacing the late-data “fork” with continuously written avro-files.
- The “After-Party Cannon” for continuous testing in our QA environment.
Thank You

Let’s talk about beam! devon@oden.io

We’re hiring! oden.io/careers