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 needs 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.

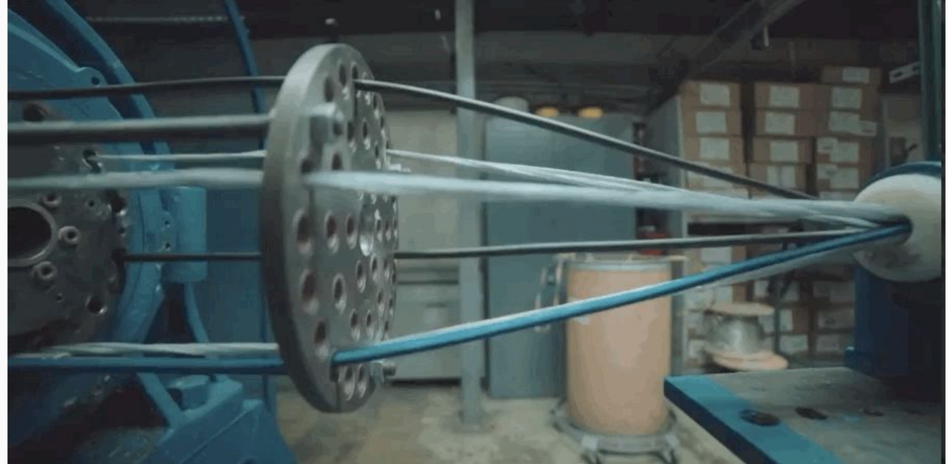
A little about Oden

Oden's Customers

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

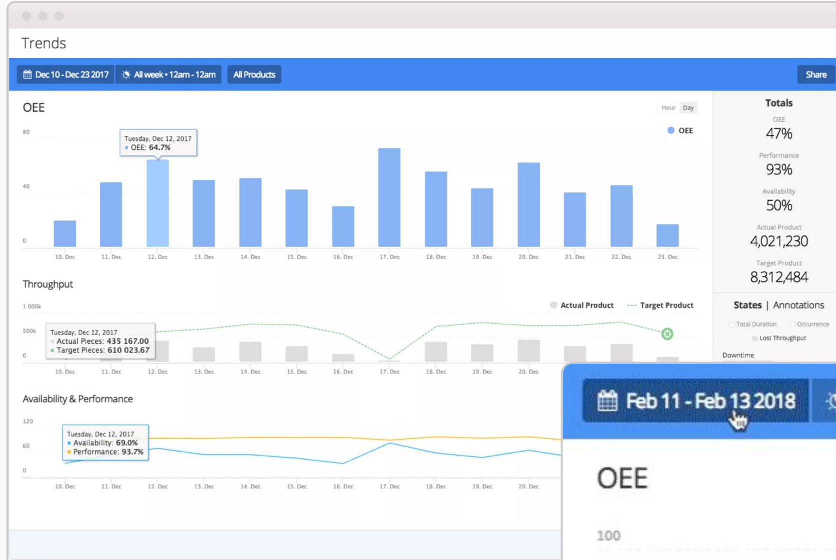
Process and Quality Engineers looking to centralize, analyze, and act on their data.

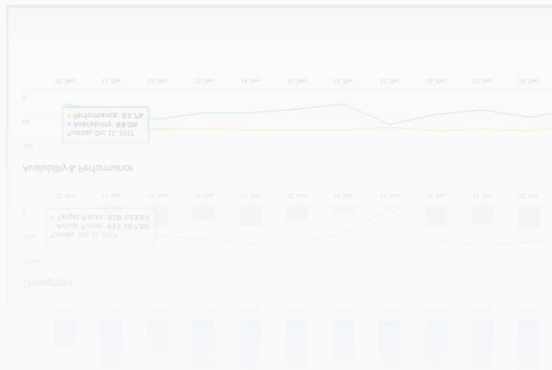
Plant managers who are looking to optimize logistics, output, and cost.

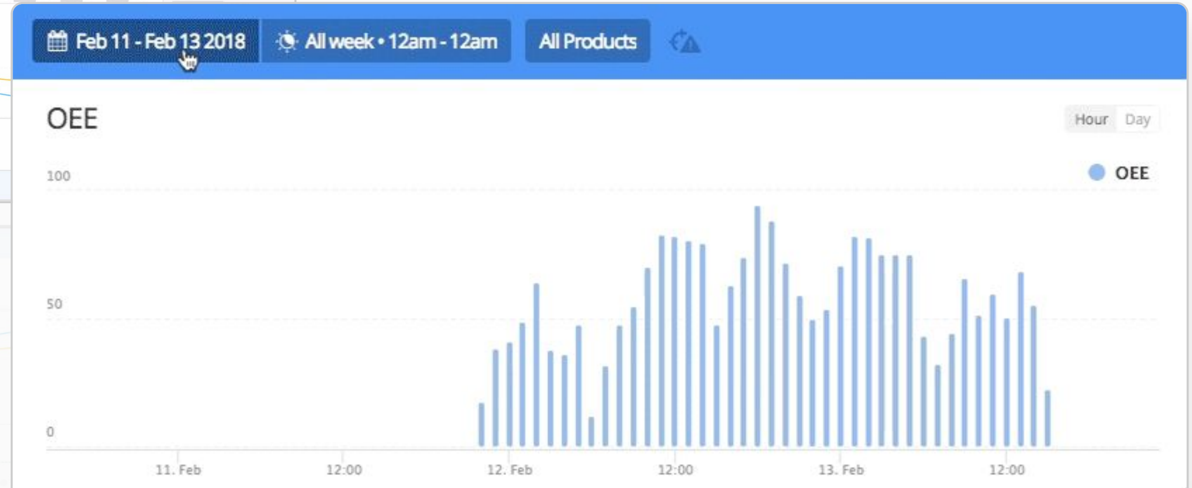


Interactive Time-series Analysis

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

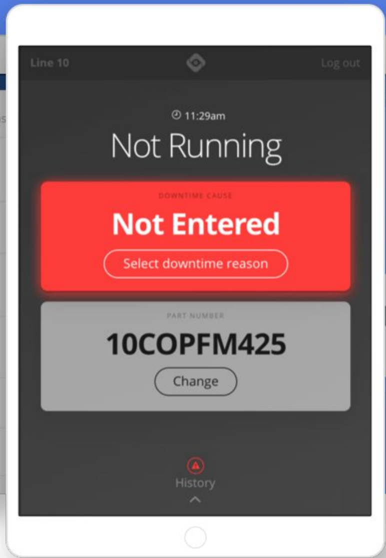


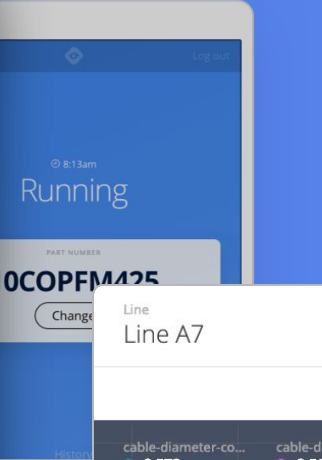




Real Time Manufacturing Data

- 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.

ALERT

Downtime violation on **Line 1**

As of 12:55pm, Line 1 has been in Downtime for more than 15 minutes.

View line

Snooze this alert for: 30m 2h 8h 24h

Powered by Oden Technologies

Is this alert useful? Let us know!

Daily Run Report

Runs completed 9:00am EST February 12, 2019 – 9:00am EST February 13, 2019

Runs sorted by worst Cpk for Cold OD Avg

SWJNG519-LQ8 View run →

Line 10 · | 0 Reels · 06:11 2/12 – 11:42 2/12 · 3h 16m uptime

METRIC	MEAN	STD DEV	TARGET	NON CON*	Cpk
Cold OD Avg	0.403	0.010	0.391 - 0.411	4.235%	**0.274**
Feet per min	274.794	194.059	-	-	-

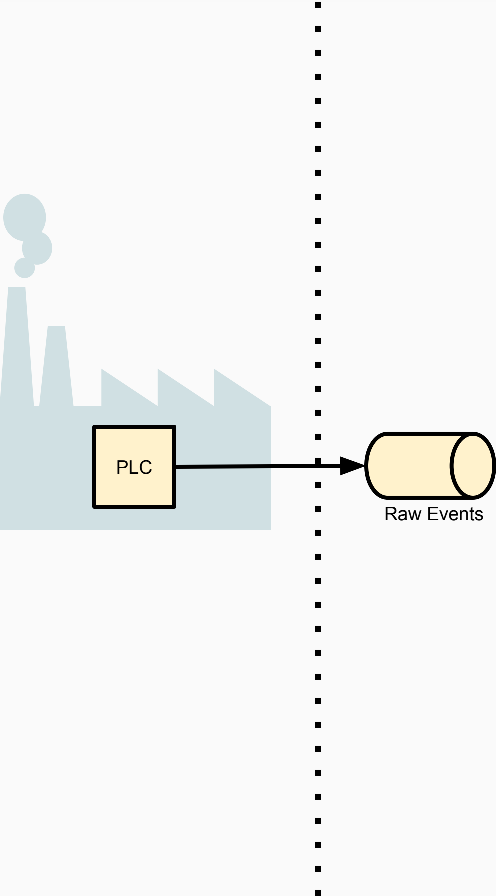
SWHD72Y-R4 View run →

Line 10 · | 0 Reels · 10:08 2/12 – 12:34 2/13 · 1h 35m uptime

METRIC	MEAN	STD DEV	TARGET	NON CON*	Cpk
Cold OD Avg	0.141	0.002	0.135 - 0.145	0.242%	**0.782**
Feet per min	829.680	492.109	-	-	-

How we use *and misuse* Apache Beam

How Oden Uses Apache Beam - Reading Events



How Oden Uses Apache Beam - Acquisition

PLC
Raw Events
Acquisition
Metrics-A

How Oden Uses Apache Beam - Acquisition

ReadFromPubsub
Running
23 hr 22 min 44 sec
2 stages

Partition
5,235 elements/s
3 days 20 hr 55 min 11 sec
1 stage

LateToMetrics
Running
1 hr 33 min 55 sec
1 stage

Runs
Running
2 hr 48 min 49 sec
1 stage

DeadLetters
358 elements/s
10 hr 34 min 14 sec
1 stage

Intervals
6 elements/s
1 hr 39 min 35 sec
1 stage

LateToLateMetrics
Running
1 hr 40 min 38 sec
1 stage

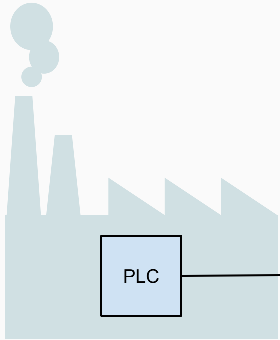
Metrics
175 elements/s
7 hr 57 min 32 sec
1 stage

RenameBatchName
Running
38 sec
1 stage

BlockedDevices
Running
1 hr 57 min 56 sec
1 stage

Multiline
4,697 elements/s
5 days 12 hr 4 min 59 sec
1 stage

Batches
Running
1 hr 9 min 40 sec
1 stage

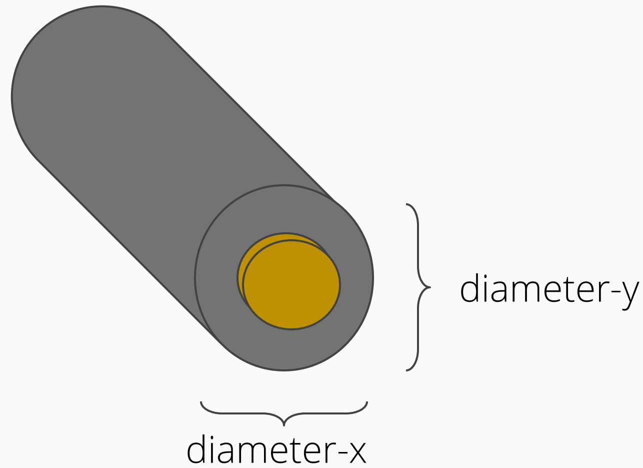


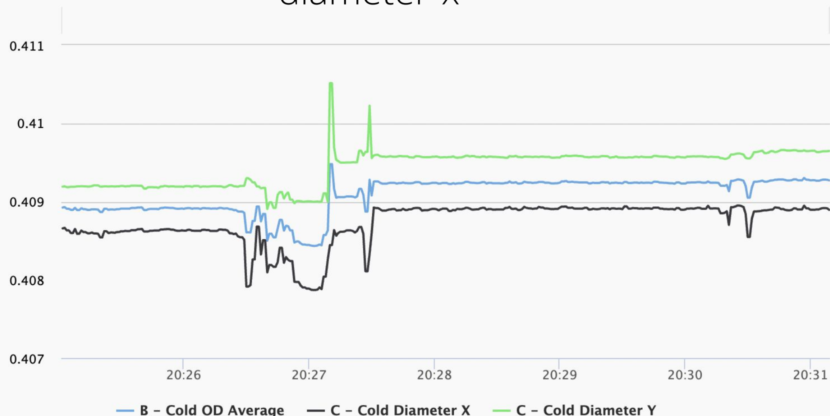
Raw Events

Acquisition

Metrics-A

How Oden Uses Apache Beam - Calculated Metrics





User Has:
diameter-x and **diameter-y**

User Wants:
avg-diameter = (diameter-x + diameter-y) / 2

- 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

Raw Events

Acquisition

Metrics-A

Calculated Metrics

Windowed Calculated Metrics

Metrics-B

How Oden Uses Apache Beam - State Change Detection

Maintenance Issue

Uptime

60
50
40
30
20
10
0

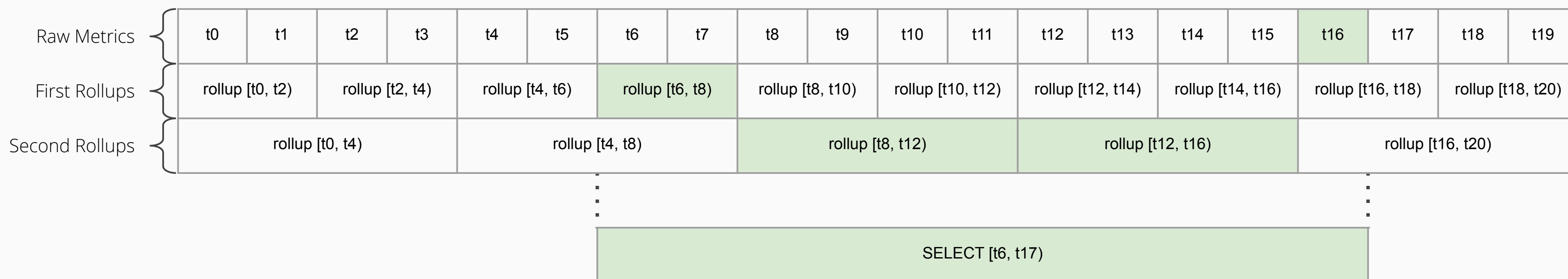
11:45 12:00 12:15 12:30 12:45 13:00 13:15 13:30 13:45

A – Line Speed (fpm)

How Oden Uses Apache Beam - State Change Detection

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

How Oden Uses Apache Beam - Rollups



- Creates special “rollup” metrics before writing to TSDB
- Optimizes aggregations i.e.
  - sum([t6...t16)) = sum([t6...t8)) + sum([t8...t12)) + sum([t12...t16)) + val(t16)
  - count([t6...t16)) = count([t6...t8)) + count([t8...t12)) + count([t12...t16)) +1
  - mean([t6...t16)) = sum([t6...t16)) / count([t6...t16))

How Oden Uses Apache Beam - Rollups

PLC

Raw Events

Acquisition

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

Raw Events

Acquisition

Metrics-A

Calculated Metrics

Windowed Calculated Metrics

Metrics-B

Rollups

State Change Detection

Metrics-C

TSDB

Postgres

How Oden Uses Apache Beam - In ~~Summary~~ Reality

PLC

Raw Events

Flaming Rainbow Bridge to Asgard

Acquisition

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

Metrics

Calculated Metrics

Windowed Calculated Metrics (60s)

Google-provided Dataflow Template that DROPS DATA when redeployed

Also BigQuery because we don't trust our DS team with either TSDB

Rollups (600s)

new TSDB

Probably a waste of money

Also Metrics

TSDB

State Change Detection

Postgres

API that uses regex as a language parser with terrifying effectiveness

Definitely a waste of money

How Oden Uses Apache Beam - Common Trends

- Downstream writes are idempotent.
- LOTS of windowing keyed by the metric.
- Real-time processing is needed for users to make real-time decisions.

How Oden Uses Apache Beam - Common Trends

- 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

Search

- Provider Counts
- Min. Speed
- Max. Speed
- Min. Price
- [x] DSL
- [x] Cable
- [x] Fiber
- [ ] Fixed Wireless

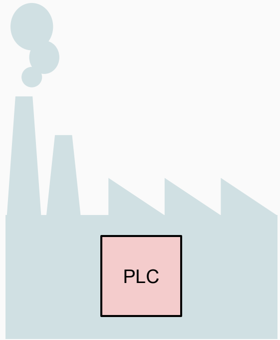
United States

Vancouver
Seattle
WASH.
MONT.
N.D.
MINN.
Minneapolis
S.D.
WIS.
MICH.
IDAHO
ORE.
WYO.
NEBR.
IOWA
Chicago
NEV.
UTAH
Denver
COLO.
KANS.
MO.
St. Louis
ILL.
IND.
OHIO
San Francisco
CALIF.
OKLA.
ARK.
TENN.
KY.
W.VA.
VA.
N.C.
S.C.
ARIZ.
N.M.
Phoenix
San Diego
Dallas
TEX.
LA.
MISS.
ALA.
GA.
Atlanta
Houston
FLA.
Miami
PA.
N.Y.
N.J.
MD.
DEL.
CONN.
MASS.
VT.
MAINE
QUE.
ONT.
Ottawa
Toronto
N.B.
P.E.I.
N.S.
Monterrey
Mexico
Gulf of Mexico
Bahamas
Cuba

Connectivity in factories is hard

2.4 GHz Coverage – All 1st Floor

How Oden Uses Apache Beam - Disconnects



Undelivered
Data Backs Up

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

Downstream jobs are flooded with late events

How Oden Uses Apache Beam - Disconnects

PLC

PLC

PLC

Raw Events

Acquisition

Metrics-A

Calculated Metrics

Windowed Calculated Metrics

Metrics-B

Rollups

State Change Detection

Metrics-C

TSDB

Postgres

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 watermarket 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);
}
```

- Trigger normally with watermark when things are on-time.
- Use early-fire plus process-time delay past first element in pane when watermark is “stuck”
- Use late-firings plus process-time delay to control for the rate of firing for “late” metrics.
- Creates a “deprecated” behavior when a client is backfilling data where triggering is *less realtime*.
- Dataflow jobs can get stuck when high allowed lateness is allowed.

https://git.io/JBSxl

Attempts at Late Data Handling - Complex Triggering Logic

```java
/* 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 [illegible] will 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 watermarket 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);
}
```

THIS IS CRAZY

- Trigger normally with watermark when things are on-time.
- Use early-fire plus process-time delay past first element in pane when watermark is “stuck”
- Use late-firings plus process-time delay to control for the rate of firing for “late” metrics.
- Creates a “deprecated” behavior when a client is backfilling data where triggering is *less realtime*.
- Dataflow jobs can get stuck when high allowed lateness is allowed.

https://git.io/JBSxl

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<ValueState<ConcurrentSkipListMap<Long, List<Metric>>>> window = StateSpecs.value();
  @ProcessElement
  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>());
    Set<String> metricIDs = metrics.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, 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

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<ValueState<ConcurrentSkipListMap<Long, List<Metric>>>> window = StateSpecs.value();
  @ProcessElement
  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>());
    Set<String> metricIDs = metrics.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, 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);
  }
}
```

EVEN MORE CRAZY

- `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

How Oden Uses Apache Beam - It all fell apart

2019-10-18
Pager Duty Incident: 2586

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?

Access Freq

New

Age of Data

Old

Real-time
Decision Making

Historical
Analysis

Access
Freq

New

Age of Data

Old

Real-time
Decision Making

Historical
Analysis

Access
Freq

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

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

How Oden Uses Apache Beam - Disconnects

PLC

PLC

PLC

Raw Events

Acquisition

Late-Events

PubSub To GCS

GCS

Metrics-A

Calculated Metrics

Windowed Calculated Metrics

Metrics-B

Rollups

State Change Detection

Metrics-C

TSDB

Postgres

One bad apple ~~ruins the bunch~~ is OK

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

Streaming

- “Unbounded” PCollections
- Generally talks to real-time queues
- Processed piece-by-piece
- More wasted worker overhead

Batch

- “Bounded” PCollections
- Generally talks to files / databases
- Processed in stages
- Workers run until they’re done
- A little cheaper *

Batch vs Streaming Sources/Sinks

Streaming

- PubSubIO
- KafkaIO

Batch

- JdbcIO
- BigQueryIO
- TextIO
- AvroIO
- GenerateSequence

Batch vs Streaming Sources/Sinks

Streaming

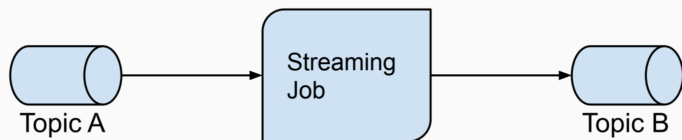
- PubSubIO
- KafkaIO
- *TextIO**
- *AvroIO**
- *GenerateSequence**

Batch

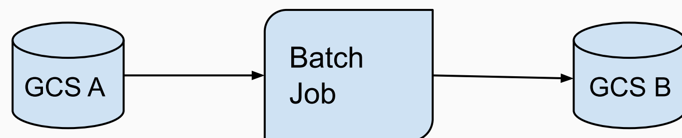
- JdbcIO
- BigQueryIO
- TextIO
- AvroIO
- GenerateSequence

Batch vs Streaming Jobs In Beam

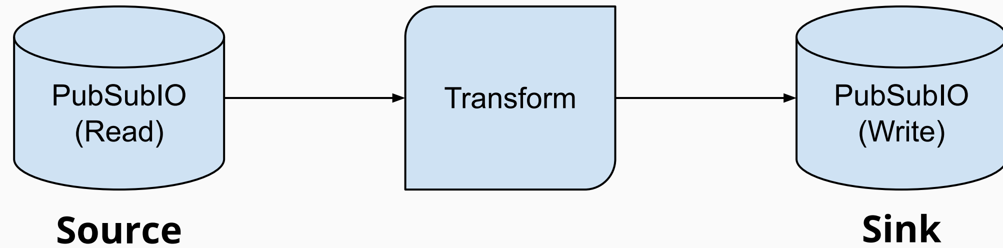
Streaming



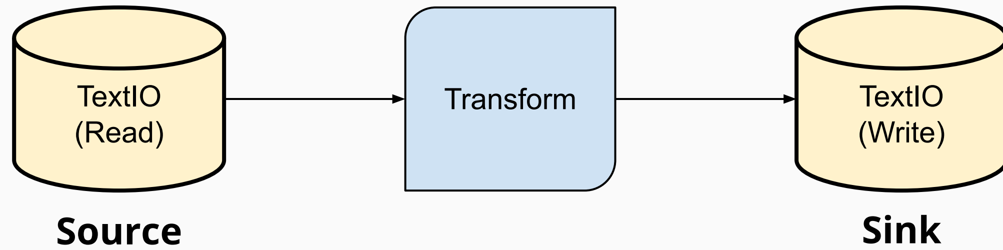
Batch



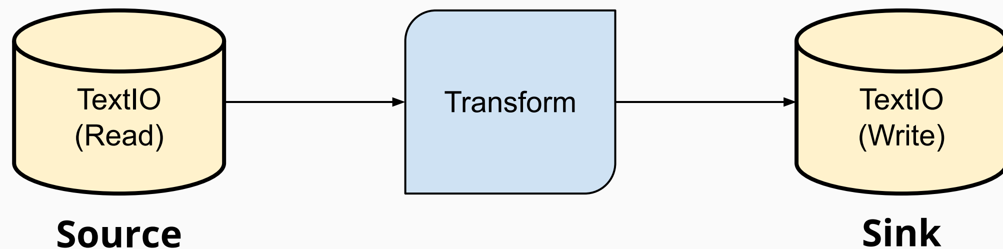
Streaming Jobs - Simple



Batch Jobs - Simple



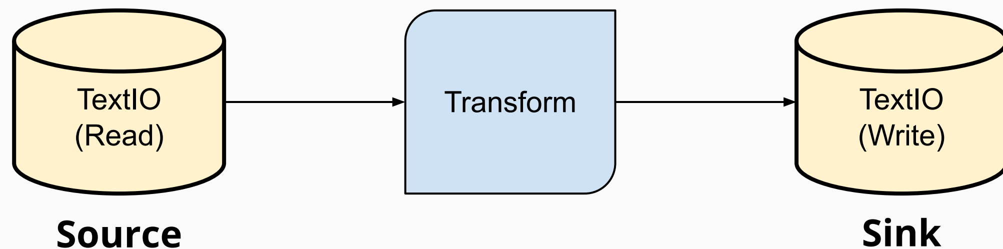
Batch Jobs - Simple



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



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

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

When the late metrics arrived

Streaming Jobs - Complex

Internal
Config API

Generate
Sequence
(5 min)

Map
Elements

Config
PCollection

Source

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

Config
PCollection
View

Side-input

PubSubIO
(Read)

Transform

PubSubIO
(Write)

Source

Sink

Batch Jobs - Complex

Internal
Config API

Generate
Sequence
(1 element)

Map
Elements

Config
PCollection

Source

`GenerateSequence.from(0).to(1)`

Config
PCollection
View

Side-input

TextIO
(Read)

Transform

TextIO
(Write)

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(
      "WriteToFiles",
      TextIO.write().to(super.filenamePrefix));
}
pipeline.run();
```

- The “core” transform is abstracted into a generic PTransform<String, String>

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(
      "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.

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(
      "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.

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(
      "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

```
// 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.

Running Streaming Jobs in Batch - With EventIO

```
// 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

```
// 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

Apache Airflow

- 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

Airflow DAGs Data Profiling Browse Admin Docs About 2021-08-03 15:03:16 UTC

On **DAG: compute_and_mail_reports_v1**

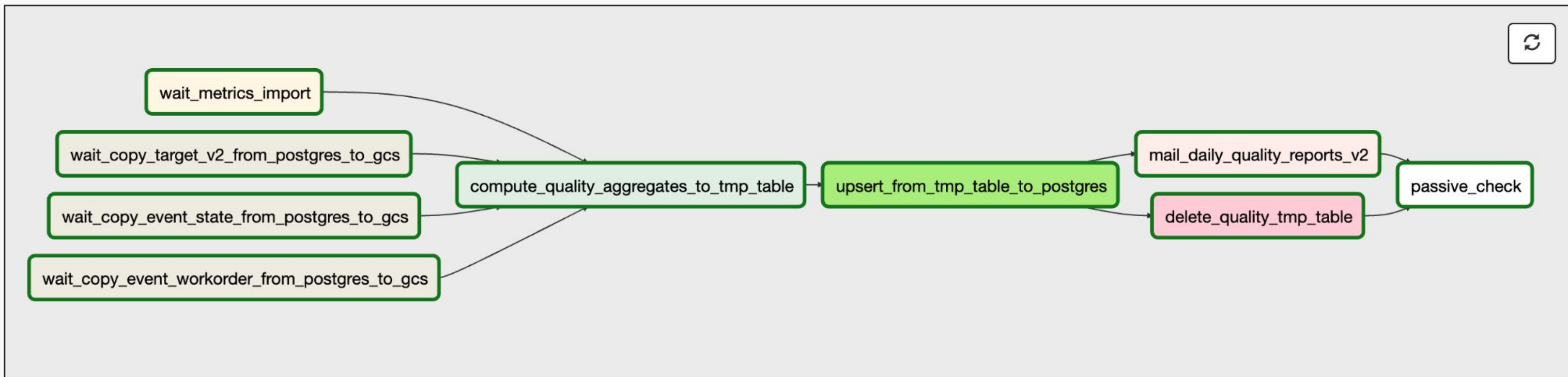
schedule: 0 14 * * *

Graph View | Tree View | Task Duration | Task Tries | Landing Times | Gantt | Details | Code | Trigger DAG | Refresh | Delete

success Base date: 2021-08-02 14:00:01 Number of runs: 25 Run: scheduled__2021-08-02T14:00:00+00:00 Layout: Left->Right Go Search for...

BigQueryCheckOperator BigQueryOperator BigQueryTableDeleteOperator GenericTransfer GoogleCloudStorageObjectUpdatedSensor PassiveCheckOperator PythonOperator

up_for_retry failed success running queued no_status scheduled skipped upstream_failed up_for_reschedule

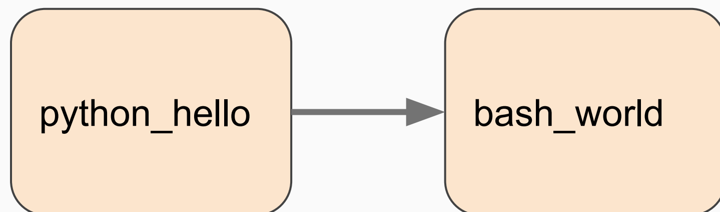


Airflow - Running Batch Dataflow Jobs

```
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

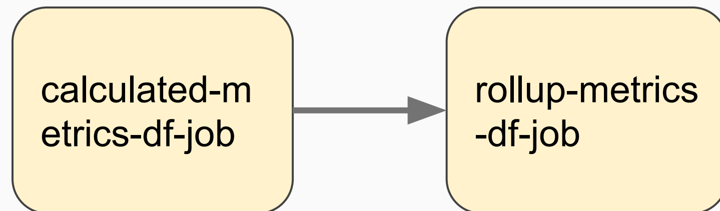
```
dag = DAG("late_data_v1", schedule_interval="30 0 * * *")

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

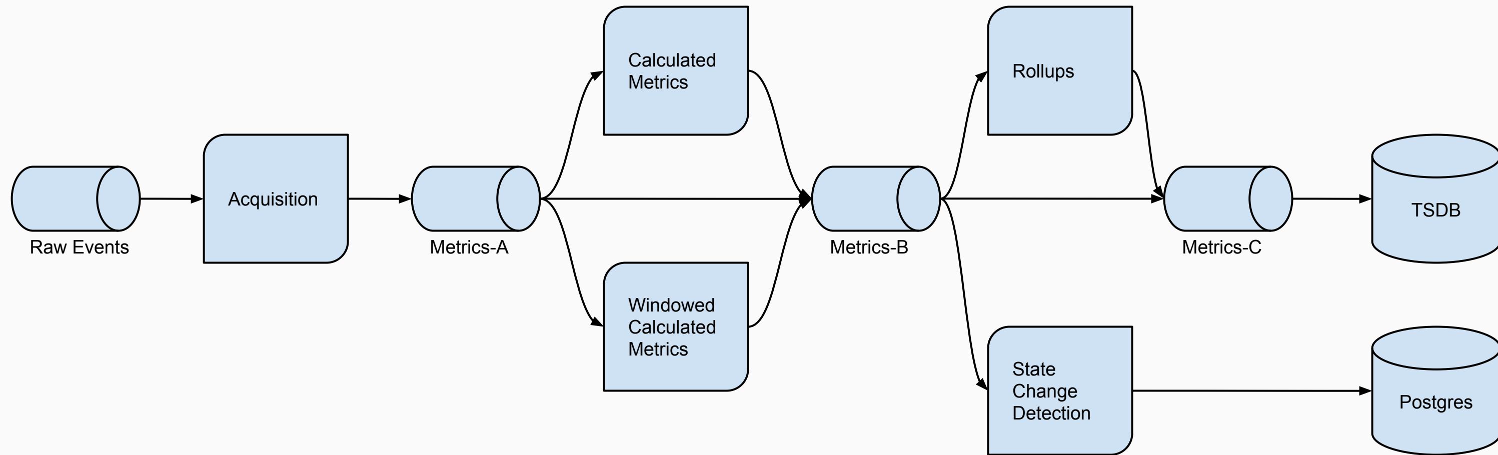
Airflow “execution_date” macro

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

ALL_METRICS  = "gs://dataflow.oden-qa.io/metric-[abc]/{{ds}}/metrics-*.ndjson"
```

GCS wildcard

Streaming Pipeline



Late Data Airflow DAG

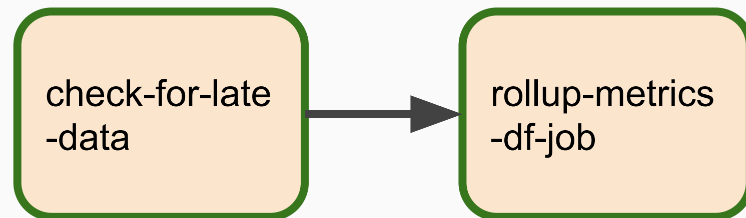
Late Events
Acquisition
Metrics-A
Calculated Metrics
Windowed Calculated Metrics
Metrics-B
Rollups
State Change Detection
Metrics-C
ToTSDB
TSDB
Postgres

Late Data Airflow DAG - Detecting Late Data in GCS

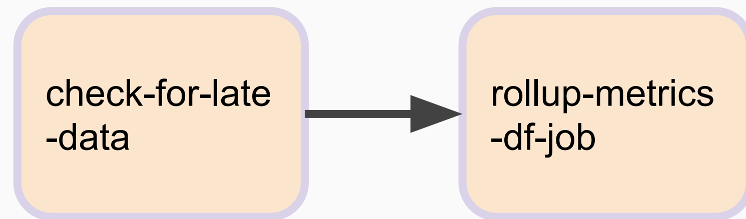
```
# Only works in >=1.10.15
check_for_late_data = GoogleCloudStoragePrefixSensor(
    dag=dag,
    task_id="check-for-late-data",
    soft_fail=True,
    timeout=60 * 2,
    bucket="gs://dataflow.oden-qa.io/metric-a/{{ds}}/",
    prefix="/metric-a/{{ds}}/",
)

...

check_for_late_data >> ...
```

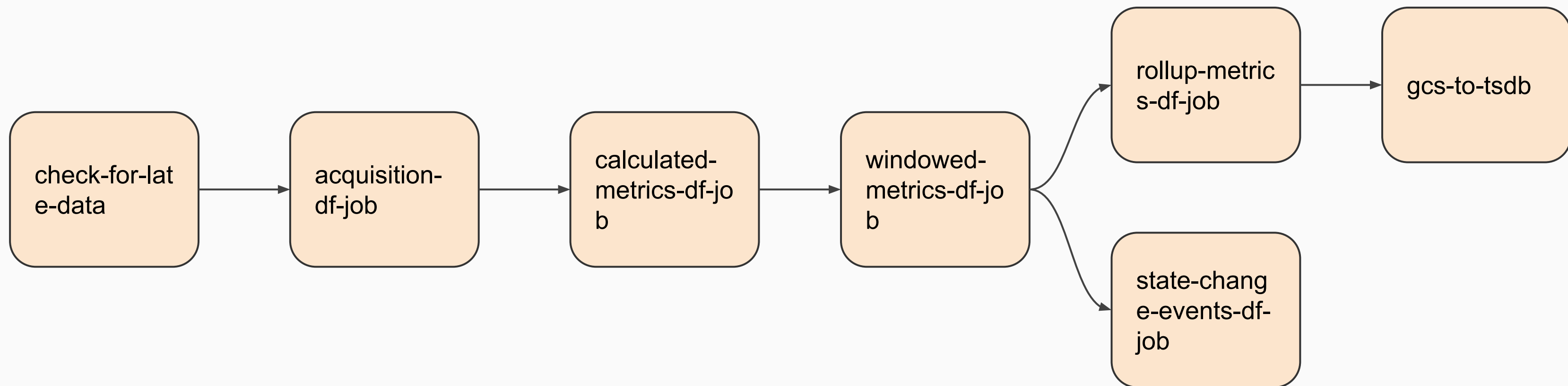


Soft-fail causes "skipped" downstream behavior when no late data for that day



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