

Dataflow Streaming Innovations

Apache Beam Summit

3:45 PM Tuesday, 8 July 2025

Tom Stepp & Ryan Wigglesworth, Google

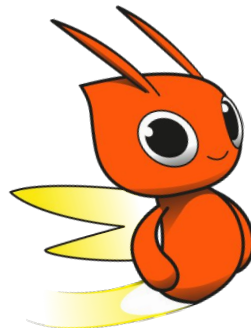


Table of Contents

Fast Job Updates	01
Direct Path	02
Kafka Integration	03
High Key Cardinality	04
ML Aware Streaming	05
Observability	06

01

Fast Job Updates

Fast Job Updates: Background

- Current options for job update:
 - Perform in-flight updates (autoscaling changes)
 - Launch a replacement job (--update).
- Work around for updating incompatible jobs:
 - Stop and replace
 - Run parallel pipelines

Fast Job Updates: Problem Statement

- Problem: Dataflow update is too slow (~15m)
 - Waiting for original job workers to shutdown
 - Waiting for new job workers to startup
- Solution: Reduce downtime with new Dataflow update options

Fast Job Updates: To The Rescue

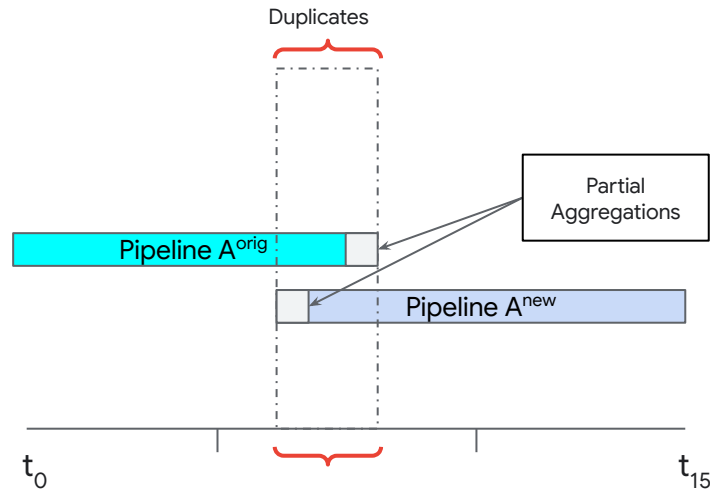
- What
 - A new version of the pipeline is created and swapped out with the original pipeline.
 - When a small period of duplicates / partial aggregations are acceptable and state does not need to be transferred.
- Benefits
 - Eliminate pause in processing for streaming pipelines during updates.
 - Reducing operational toil in updating incompatible pipelines.

Fast Job Updates: Key changes

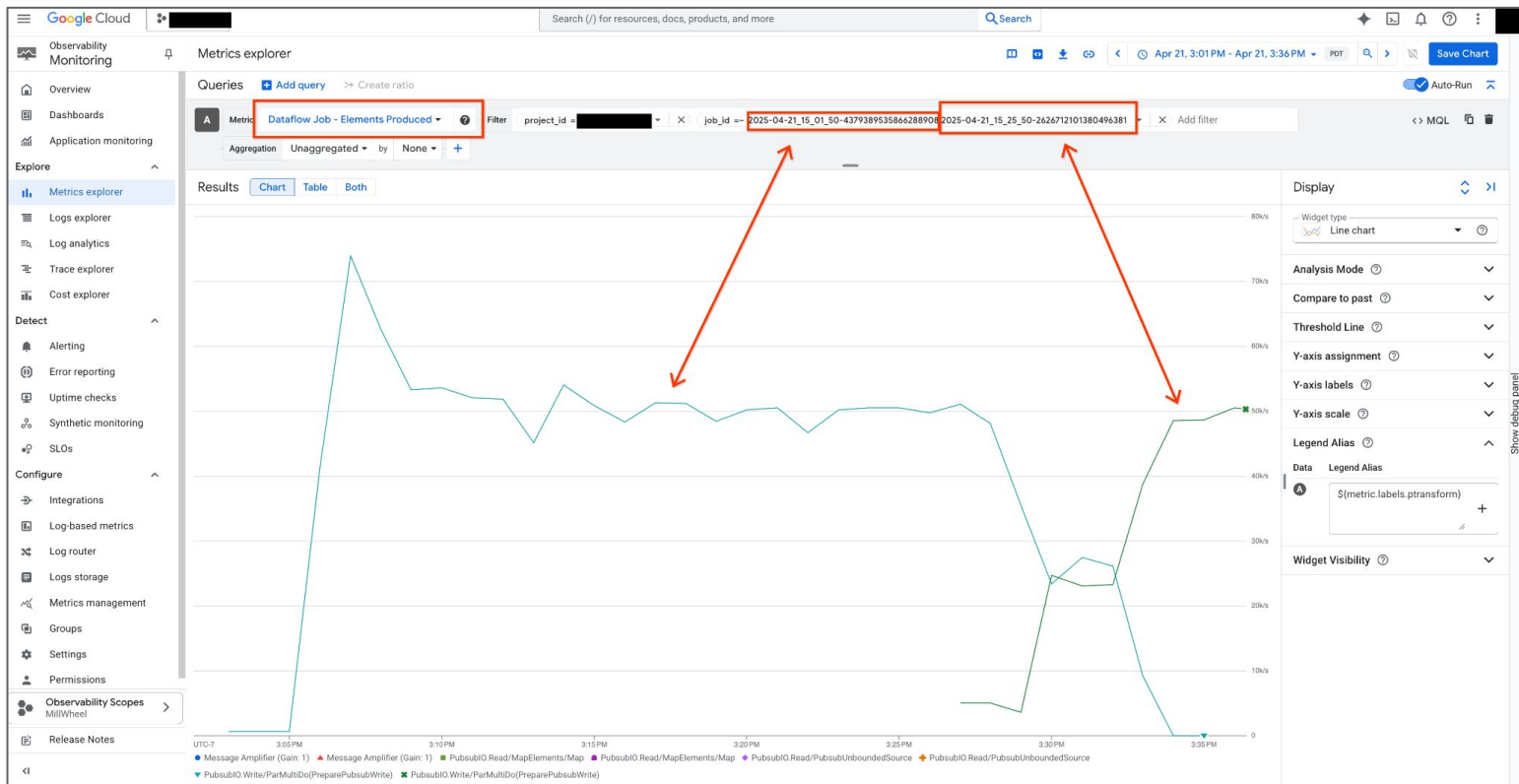
- Start the new jobs worker pool while the old job is still running
- New job initializes to same number of workers as old job

Fast Job Updates: Workflow

1. A^{new} pipeline is started.
2. When A^{new} workers are ready, new messages will be read from sources.
3. A^{orig} will be set to draining after configured max overlap duration elapses.
4. When A^{orig} drain completes, no more duplicates or partial aggregations.



Fast Job Updates: Processing Transfer



Fast Job Updates: How to use

- Options when launching new job, using Java syntax:
 - `--dataflowServiceOptions=parallel_replace_job_name=<name>`
 - `--dataflowServiceOptions=parallel_replace_job_min_parallel_pipelines_duration=4m`

Fast Job Updates: Future ideas

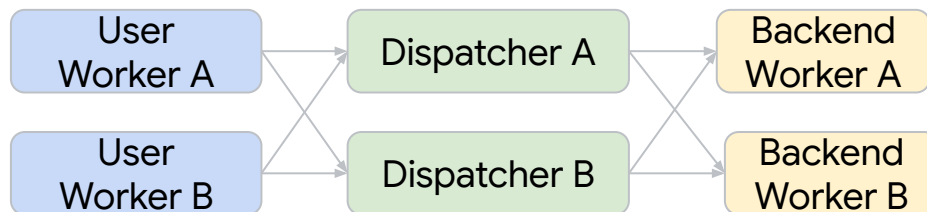
- Allow jobs to retain the same name.
- Stop and Replace flows, where A^{orig} is stopped before A^{new} is started.
- Reduce down time when issuing `--update` with a compatible job.

02

Direct Path

Direct Path: Original architecture

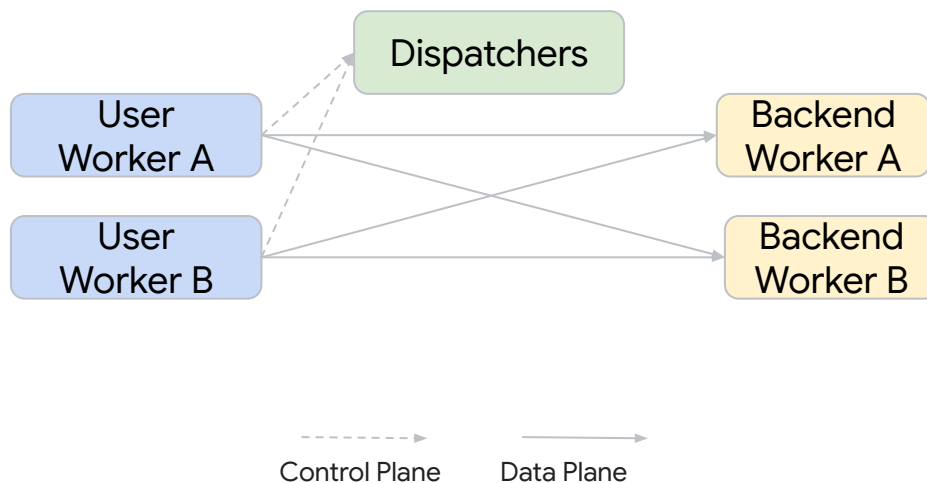
- A design limited by available tech for GCP services at the time
- Couldn't expose address to single backend worker task directly



→
Data Plane

Direct Path: Updated architecture

- An improved design, where dispatcher can share the address of backend tasks instead of acting as proxy.
- Eliminates issue of high fanout



Direct Path: Key benefits

- Simplification
- Reduce “noisy neighbor” problem
- Improve latency on data path

Direct Path: Impact

- Reduced backlog and latency on KafkaToBigQuery job

	Backlog Mean	Backlog P99	Data Lag Mean	Data Lag P99
Directpath	2.6 MB	16.5 MB	3.6 sec	16.4 sec
Cloudpath	4.1 MB	219.1 MB	4.3 sec	31.8 sec
Difference	-35.3%	-92.4%	-16.6%	-48.3%

03

Kafka Integration

Managed IO

Managed IO

Simplifies pipeline management for supported sources and sinks.

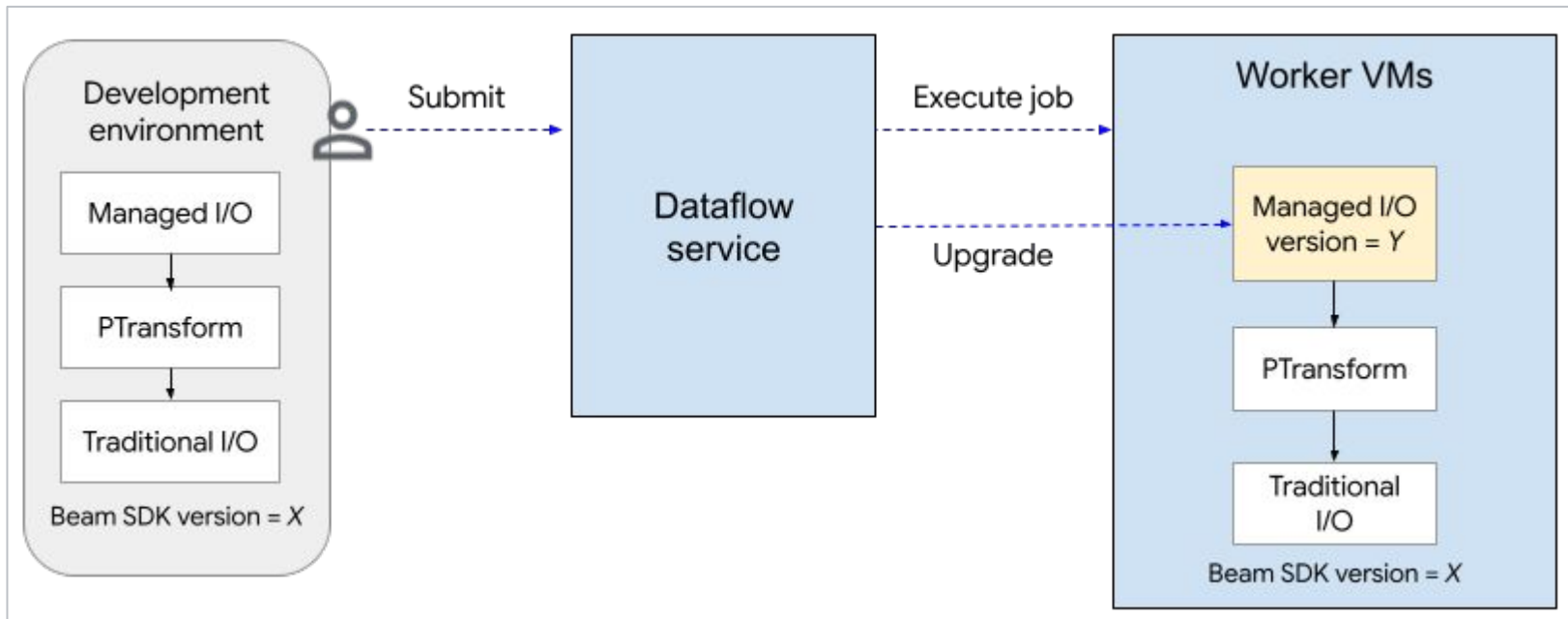
Consists of:

- A Beam transform that provides a common API for creating I/O connectors.
- A service that provides IO upgrades independent of the Beam version.

Advantages of Managed IO

- Dataflow automatically upgrades the managed I/O connectors in your pipeline.
- A single configuration API, resulting in simple and consistent pipeline code.

Managed IO



Managed IO for Kafka

It's simple!

- `Managed.read(Managed.KAFKA).withConfig(readConfigMap);`
- `Managed.write(Managed.KAFKA).withConfig(writeConfigMap);`

Managed IO for Kafka

1. `ImmutableMap<String, Object> config = ImmutableMap.<String, Object>builder()`
2. `.put("bootstrap_servers", options.getBootstrapServer())`
3. `.put("topic", options.getTopic())`
4. `.put("max_read_time_seconds", "1")`
5. `.build();`
6. `Managed.read(Managed.KAFKA).withConfig(readConfig);`

Redistribute Transform

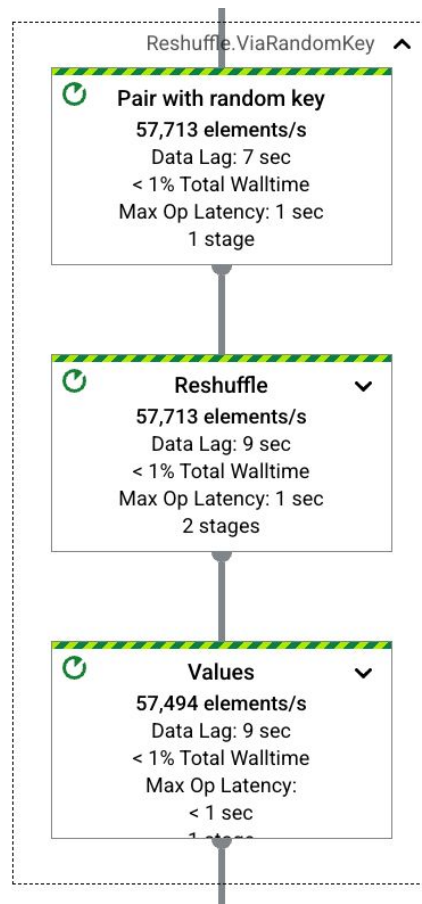
Partition-Limited Parallelism

- Problem: Limited Kafka partitions cause limited parallelism (keys)
- Solution: Add parallelism by rekeying inputs via
`read.withRedistribute().withRedistributeNumKeys(N)`

Pipeline Translation

Translates to:

- Add key: KV<Key, Record>
- Reshuffle (GroupByKey)
- Remove key, extract values: Record



At Least Once Optimization

- For at-least-once processing, rather than exactly-once.
- Specify: `kafkaRead.withRedistribute().withAllowDuplicates()`
- Why? Runner optimization is much cheaper than regular redistribute.

Pipeline (Kafka To BigQuery)	Throughput (elements/sec)	Hourly cost	Throughput / Cost
Standard	70 K	\$1.29	54.3
Redistribute	270 K	\$5.89	45.8
Redistribute + Allow Duplicates	280 K	\$1.86	150.5

Offset Deduplication

Offset Deduplication

- Problem:
 - Redistribute is expensive due to exactly-once cost and latency.
 - Can't use allow duplicates optimization because you need exactly-once.
- Solution:
 - Utilize source metadata, such as message offsets, to make shuffle cheaper.
 - Add `withOffsetDeduplication()` to your Kafka read step with Redistribute.

04

High Key Cardinality

High Key Cardinality: Background

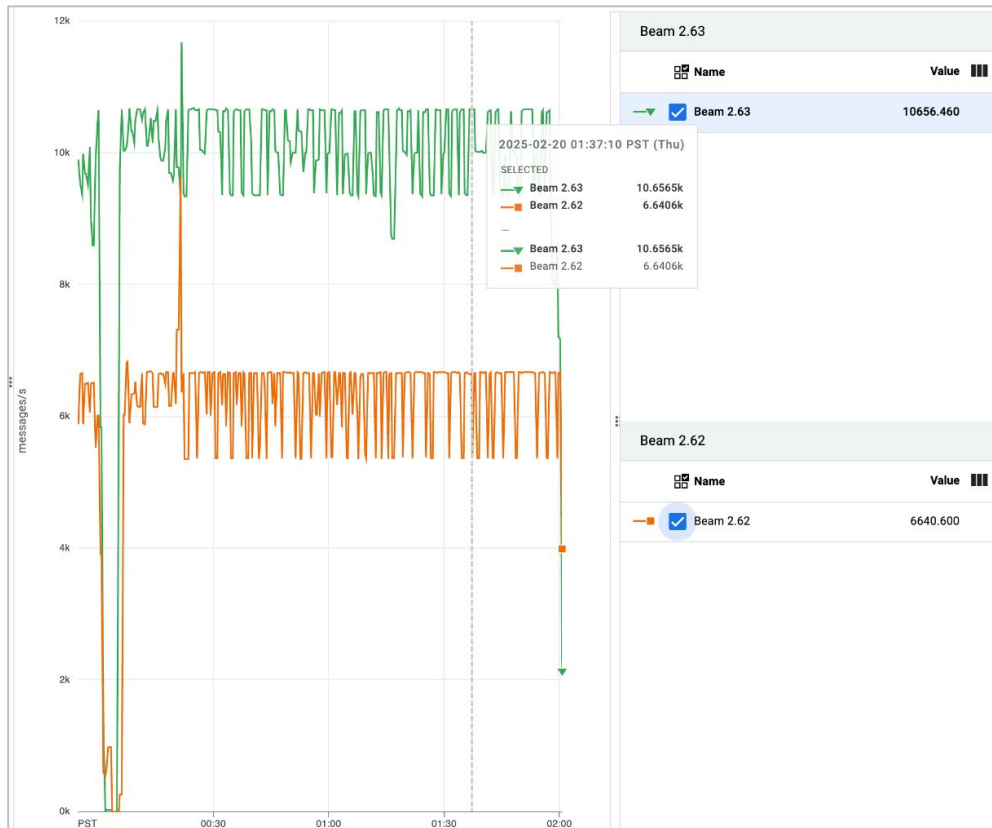
- Jobs with 1M+ keys typically perform much worse than jobs using smaller key space
- Example: sample job with 50 keys processed 20x msg/sec than job with 1M keys
- Some Dataflow customers have 1M+ keys that could be better supported

High Key Cardinality: Details

- github.com/apache/beam/issues/33578 – Dataflow Streaming Micro Optimizations
- Beam versions 2.63+ contain related improvements which help with many keys

High Key Cardinality: Results with Kafka

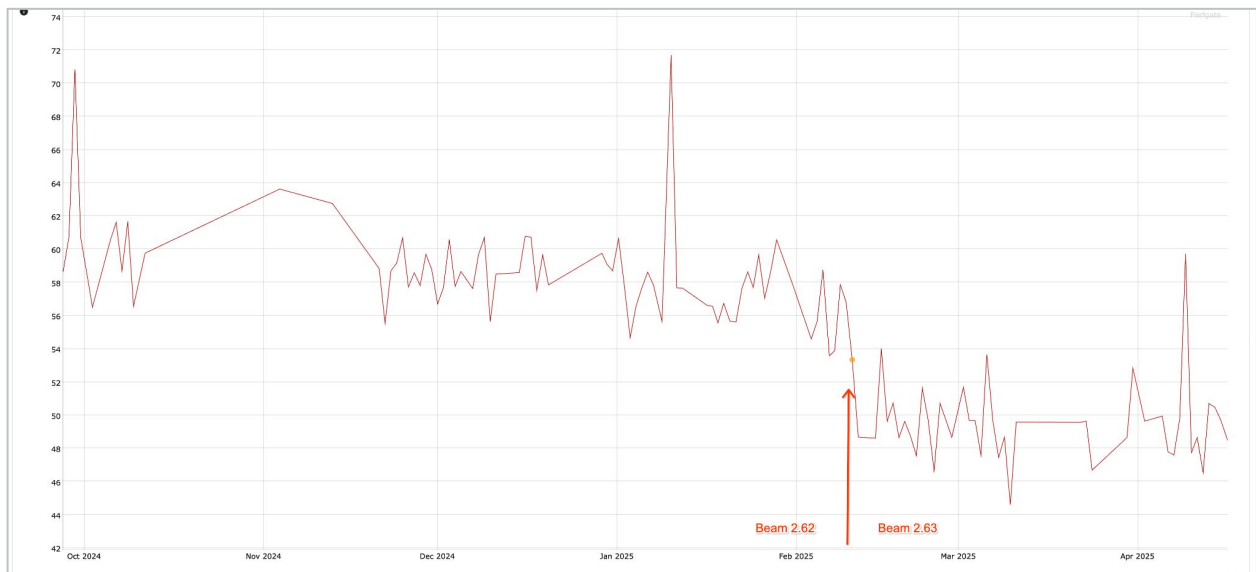
60% high throughput on a
simple Kafka source job



High Key Cardinality: Results with Nexmark Queries

Nexmark queries benchmark streaming processing.

15% runtime reduction



05

ML Aware Streaming

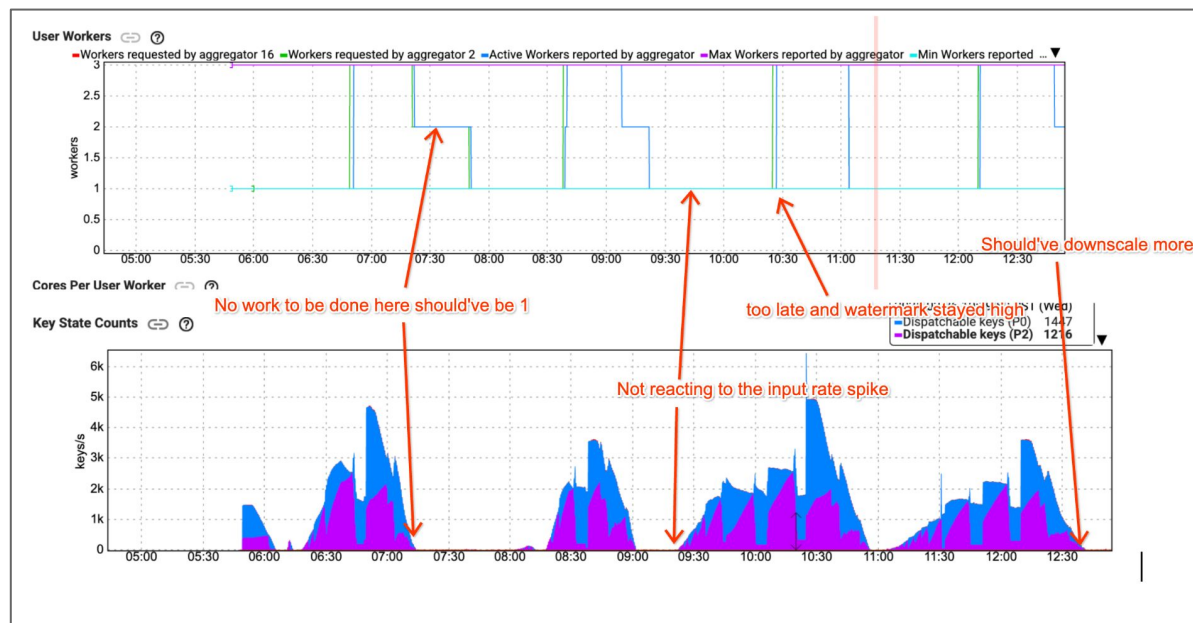
GPU Intensive Autoscaling

ML Aware Streaming: GPU Intensive Autoscaling

- Standard autoscaling policy is designed for traditional transforms that take seconds to compute.
- This leads to issues when dealing with ML workloads which can take minutes or more to process.

ML Aware Streaming: GPU Intensive Autoscaling

Example job showing poor performance with standard autoscaling

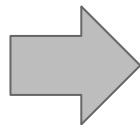
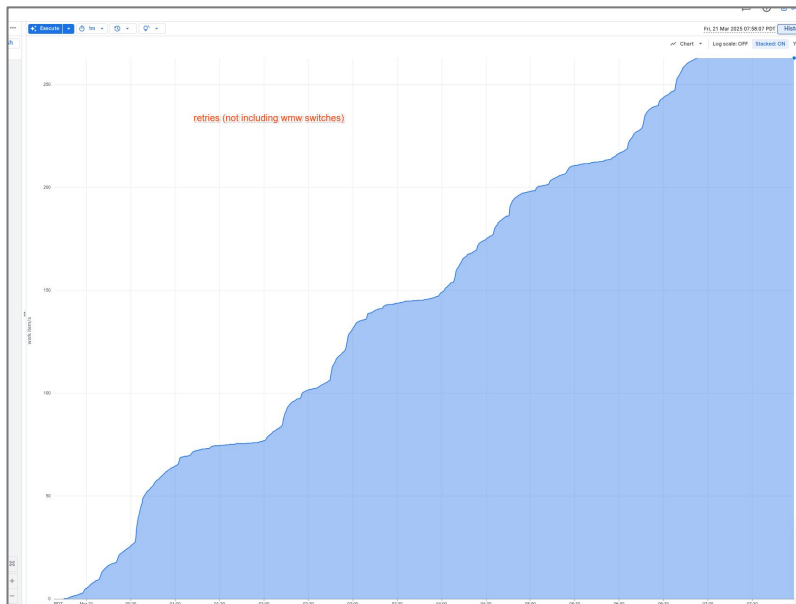


Async Dofn

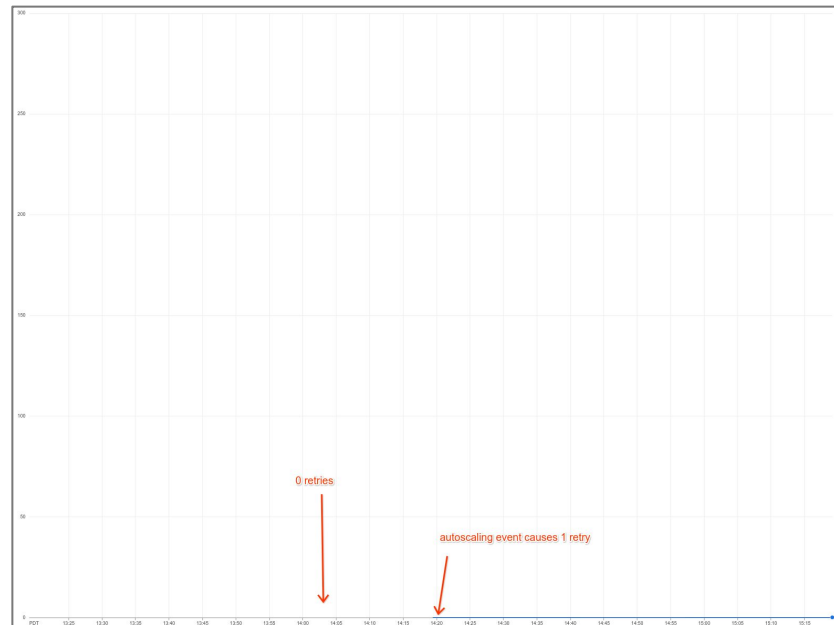
- Convert synchronous GetWork into an Asynchronous process.
 - Allows committing some messages in a bundle
 - Avoids retries in case of range invalidations
 - Much faster for long running processing but slower for quick message processing since an additional RPC to Dataflow is required.

ML Aware Streaming: Reduced retries

Defaults: Job processes 650 messages and retries 250 of them!!



GPU Autoscaling Algorithm and Retry Optimizations



ML Aware Streaming: Performance Results

	Max Backlog Bytes	Avg Worker Count*
Baseline	594392	54.7313
Experiment	124715	35.0562
Diff %	-79.01%	-35.94%

Beam Support

ML Aware Streaming: Async DoFn

- Before: `result = messages | beam.ParDo(RegularDofn())`
- After: `result = messages | beam.ParDo(AsyncWrapper(RegularDofn()))`

ML Aware Streaming: Parallelism Annotation

- To hint autoscaling to target "N"
- Add parallelism annotation to a DoFn: `@parallelism(N)`

Right Fitting

ML Aware Streaming: Right Fitting

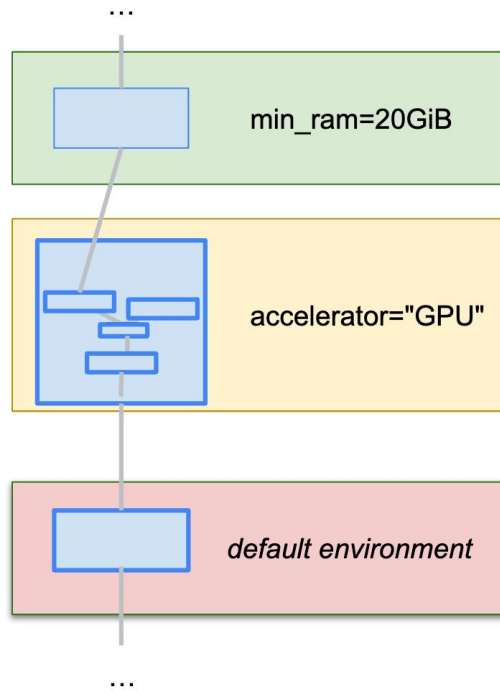
```
p | "Read elements" >> ...
```

```
| beam.ParDo(MyDoFn()).with_resource_hints(min_ram="20GiB") |
```

```
| MyCompositePTransform().with_resource_hints(accelerator="GPU")
```

```
| beam.Map(...)
```

```
| "Write elements" >> ...
```



06

Observability

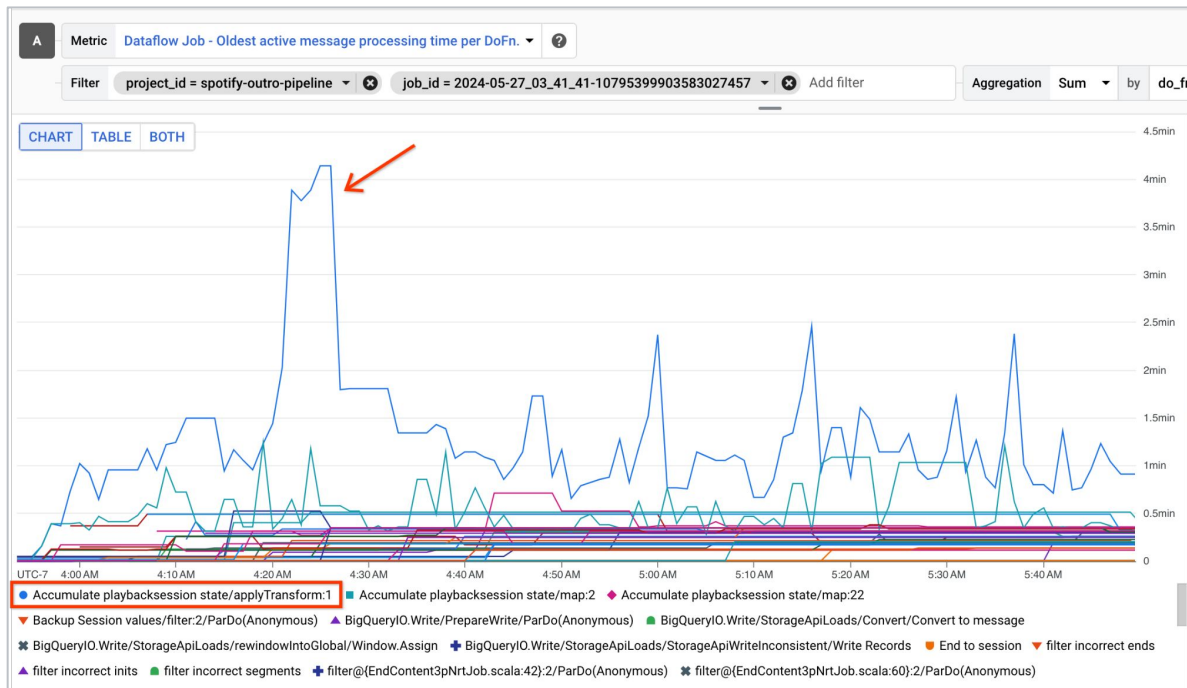
Metrics

Observability: Metrics

- `job/dofn_latency_average`: average processing time for a single DoFn
- `job/dofn_latency_max`: maximum processing time for a single DoFn
- `job/dofn_latency_min`: minimum processing time for a single DoFn
- `job/dofn_latency_num_messages`: number of messages processed by a single DoFn
- `job/dofn_latency_total`: total processing time in a single DoFn
- `job/oldest_active_message_age`: long the oldest active message processing within a DoFn

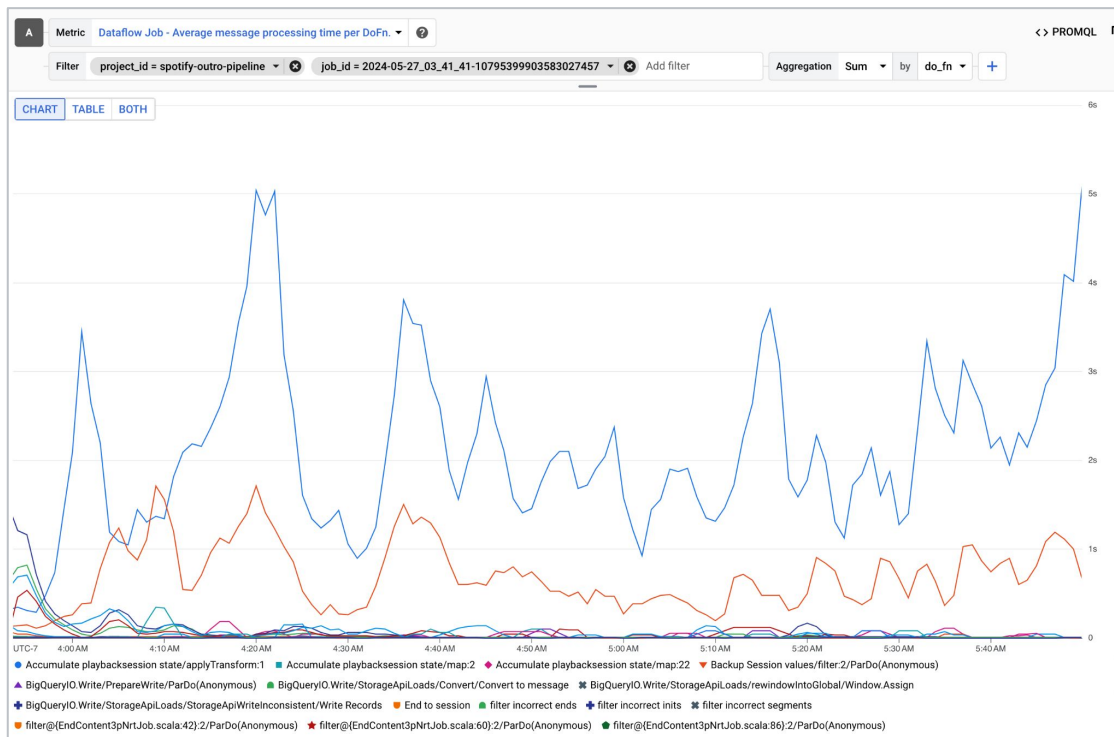
Observability: Metrics

Which DoFn is slowest?



Observability: Metrics

Which DoFn adds the most latency on average?



Bottleneck Detection

Observability: Bottleneck Detection

- A running Streaming Dataflow pipeline consists of a series of components:
 - streaming shuffles,
 - user-defined function (DoFn) processing threads,
 - persistent state checkpointing,
 - etc.
- Interconnected by queues, pushed from upstream to downstream.

Observability: Bottleneck Detection

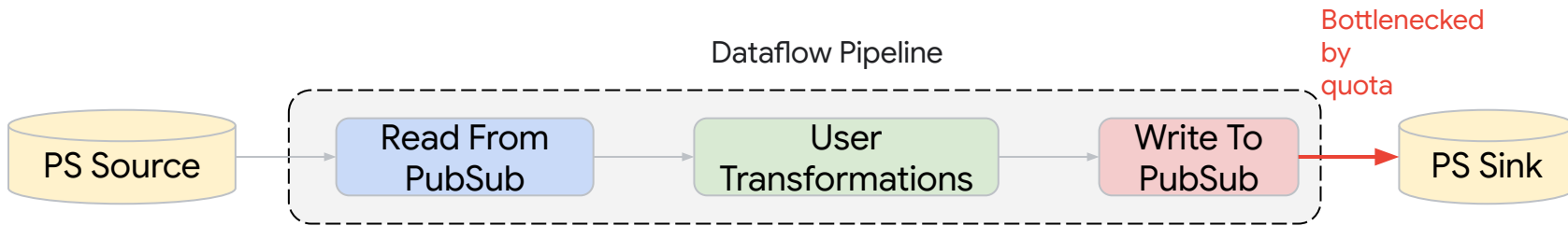
- The overall throughput capacity of a pipeline is frequently constrained by a single component, or "bottleneck."
- The pipeline's ability to accept and process input data cannot exceed the rate at which this bottleneck can operate.

Observability: Bottleneck Detection

- The primary objective of bottleneck detection is to precisely identify and comprehend the nature of the bottleneck.
- This eliminates guesswork in pinpointing the fundamental cause of pipeline delays and provides actionable insights for improvement.

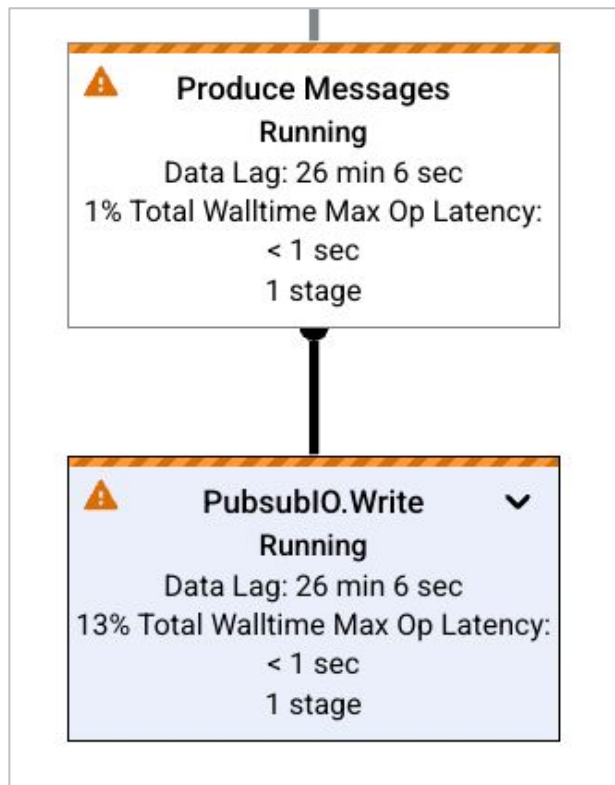
Observability: Bottleneck Detection

- Example

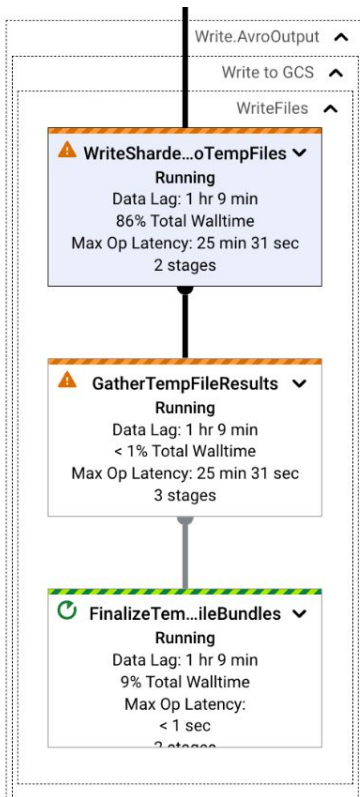


Observability: Bottleneck Detection

- Shown in UI

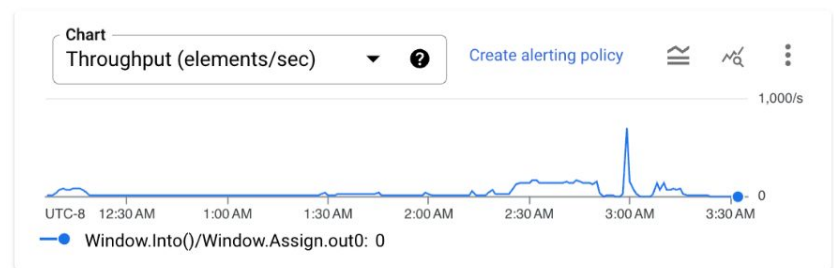


Observability: Bottleneck Detection



Wall time (% of Total)	1 hr 36 min (86%)
Max operation latency	25 min 31 sec See Logs for potential stuck operation traces
Bottleneck keys	13 sharding keys are bottlenecked with significant backlog. This is likely due to hot keys, insufficient key parallelism, long operations, underprovisioning, or other issues. Hot keys troubleshooting guide Insufficient parallelism troubleshooting guide

Input collections



Window.Into()/Window.Assign.out0

Elements added 476,477

(Approximate)

Estimated size 240.16 GB

Observability: Bottleneck Detection

Step info

Step name	WriteToSplunk/Write Splunk events
System watermark (Lag) ?	February 25, 2025 at 8:54:01 AM UTC-8 (2 sec)
Data watermark (Lag) ?	February 17, 2025 at 3:46:05 PM UTC-8 (7 days 17 hr)
Wall time (% of Total) ?	16 hr 27 min (67%)
Max operation latency ?	2 min 28 sec
Bottleneck keys ?	<p>⚠️ 1 sharding keys are bottlenecked with significant backlog. This is likely due to hot keys, insufficient key parallelism, long operations, underprovisioning, or other issues.</p> <p>Hot keys troubleshooting guide 🔗</p> <p>Insufficient parallelism troubleshooting guide 🔗</p>

Observability: Bottleneck Detection

← bottleneck-det...

■ STOP

📄 CREATE SNAPSHOT

🗑️ ARCHIVE

🔗 SHARE

⋮

📧 SEND FEEDBACK

JOB GRAPH

EXECUTION DETAILS

JOB METRICS

COST

RECOMMENDATIONS

AUTOSCALING

Job steps view
Graph view

CLEAR SELECTION

1 stage

Window Disp...Every 5sec

Running

Data Lag: 16 min 1 sec

< 1% Total Walltime

Max Op Latency: < 1 sec

1 stage

GroupByKey

Running

Data Lag: 25 min 19 sec

< 1% Total Walltime

Max Op Latency: < 1 sec

2 stages

GenerateSli...Operations

26 elements/s

Data Lag: 25 min 19 sec

100% Total Walltime

Max Op Latency: 23 sec

1 stage

Step info

Step name

GenerateSlightlySlowOperations

System watermark (Lag)

June 11, 2025 at 11:49:22 AM UTC-7 (15 min 56 sec)

Data watermark (Lag)

June 11, 2025 at 11:39:59 AM UTC-7 (25 min 19 sec)

Wall time (% of Total)

6 days 15 hr (100%)

Bottleneck Status

⚠️ Processing is ongoing but backlog is steady.
This step is likely to be a bottleneck due to:
High work queue time - too many keys or not enough workers or worker harness threads
[Bottlenecks troubleshooting guide](#)

Max operation latency

23 sec

Key Parallelism

59996

Transform Function

com.google.cloud.dataflow.integration.BottleneckDetectorTest Function

Input collections

Chart

Throughput (elements/sec)

⚠️ No data is available for the selected time frame.

UTC-7 11:40 AM 11:50 AM 12:00 PM

Advanced Debugging


Observability: Advanced Debugging


- Context: Processing was slow by a worker and I want more details.
- Problem: Currently need to dig through logs, only >5m processing is logged.
- Solution: Provide user worker thread stacks in the Job UI


Observability: Advanced Debugging


The screenshot displays the Google Cloud Dataflow console interface. At the top, the breadcrumb navigation shows 'Dataflow / Jobs / Dataflow job details'. Below this, the job name 'beamapp-gith...' is followed by several action buttons: 'STOP', 'CREATE SNAPSHOT', 'IMPORT AS PIPELINE', 'SHARE', 'ARCHIVE', and 'SEND FEEDBACK'. A horizontal menu contains tabs for 'JOB GRAPH', 'EXECUTION DETAILS', 'JOB METRICS', 'COST', 'RECOMMENDATIONS', 'AUTOSCALING', and 'Debug Capture'. The 'Debug Capture' tab is currently selected and highlighted. On the left side of the main content area, there is a 'Job steps view' dropdown menu with 'Graph view' selected. In the center, a job step card for 'ReadPubSub' is shown, indicating it is 'Running' with a 'Data Lag: 0 sec' and 'Max Op Latency: < 1 sec'. On the right side, there is a 'CLEAR SELECTION' button. A prominent red arrow points from the bottom right towards the 'Debug Capture' tab.


Observability: Advanced Debugging


 Dataflow / Jobs / Dataflow job details


 beamapp-gith... ■ STOP 📄 CREATE SNAPSHOT + IMPORT AS PIPELINE 🔗 SHARE 🗑️ ARCHIVE ⋮ SEND FEEDBACK

 JOB GRAPH EXECUTION DETAILS JOB METRICS COST RECOMMENDATIONS AUTOSCALING **Debug Capture**



 Worker Selector :




[sjhhd-sasdds-224asada-2342](#)
[sjhhd-sasdds-224asada-2343](#)
[sjhhd-sasdds-224asada-2345](#)

Observability: Advanced Debugging

```

java.base@11.0.20/jdk.internal.misc.Unsafe.park(Native Method)
java.base@11.0.20/java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:234)
java.base@11.0.20/java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.awaitNanos(AbstractQueuedSynchronizer.java:2123)
java.base@11.0.20/java.util.concurrent.ScheduledThreadPoolExecutor$DelayedWorkQueue.take(ScheduledThreadPoolExecutor.java:1182)
java.base@11.0.20/java.util.concurrent.ScheduledThreadPoolExecutor$DelayedWorkQueue.take(ScheduledThreadPoolExecutor.java:899)
java.base@11.0.20/java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1054)
java.base@11.0.20/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1114)
java.base@11.0.20/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:628)
java.base@11.0.20/java.lang.Thread.run(Thread.java:829)

--- Threads (6): [Thread[DataflowWorkUnits-1061,5,main], Thread[DataflowWorkUnits-1052,5,main], Thread[DataflowWorkUnits-1049,5,main], Thread[DataflowWorkUnits-1062,5,main], Thread[DataflowWorkUnits-1063,5,main], Thread[DataflowWorkUnits-1064,5,main]] State: TIMED_WAITING stack: ---
java.base@11.0.20/jdk.internal.misc.Unsafe.park(Native Method)
java.base@11.0.20/java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:234)
java.base@11.0.20/java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.awaitNanos(AbstractQueuedSynchronizer.java:2123)
java.base@11.0.20/java.util.concurrent.LinkedBlockingQueue.poll(LinkedBlockingQueue.java:458)
java.base@11.0.20/java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1053)
java.base@11.0.20/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1114)
java.base@11.0.20/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:628)
java.base@11.0.20/java.lang.Thread.run(Thread.java:829)

--- Threads (4): [Thread[otp445010547-24,5,main], Thread[otp445010547-29,5,main], Thread[otp445010547-30,5,main], Thread[otp445010547-27,5,main]] State: TIMED_WAITING stack: ---
java.base@11.0.20/jdk.internal.misc.Unsafe.park(Native Method)
java.base@11.0.20/java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:234)
java.base@11.0.20/java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.awaitNanos(AbstractQueuedSynchronizer.java:2123)
app//org.apache.beam.runners.dataflow.worker.repackaged.org.eclipse.jetty.util.thread.QueuedThreadPool$Runner.idleJobPoll(QueuedThreadPool.java:974)
app//org.apache.beam.runners.dataflow.worker.repackaged.org.eclipse.jetty.util.thread.QueuedThreadPool$Runner.run(QueuedThreadPool.java:1018)
java.base@11.0.20/java.lang.Thread.run(Thread.java:829)

--- Threads (3): [Thread[RMI TCP Connection(idle),5,RMI Runtime], Thread[grpc-default-executor-11,5,main], Thread[grpc-default-executor-12,5,main]] State: TIMED_WAITING stack: ---
java.base@11.0.20/jdk.internal.misc.Unsafe.park(Native Method)
java.base@11.0.20/java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:234)
java.base@11.0.20/java.util.concurrent.SynchronousQueue$TransferStack.awaitFulfill(SynchronousQueue.java:462)
java.base@11.0.20/java.util.concurrent.SynchronousQueue$TransferStack.transfer(SynchronousQueue.java:361)
java.base@11.0.20/java.util.concurrent.SynchronousQueue.poll(SynchronousQueue.java:937)
java.base@11.0.20/java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1053)
java.base@11.0.20/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1114)
java.base@11.0.20/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:628)
java.base@11.0.20/java.lang.Thread.run(Thread.java:829)

--- Threads (2): [Thread[grpc-default-worker-ELG-1-1,5,main], Thread[grpc-default-worker-ELG-1-2,5,main]] State: RUNNABLE stack: ---
app//org.apache.beam.vendor.grpc.v1p69p0.io.netty.channel.epoll.Native.epollWait0(Native Method)
app//org.apache.beam.vendor.grpc.v1p69p0.io.netty.channel.epoll.Native.epollWait(Native.java:193)
app//org.apache.beam.vendor.grpc.v1p69p0.io.netty.channel.epoll.EpollEventLoop.epollWait(EpollEventLoop.java:304)
app//org.apache.beam.vendor.grpc.v1p69p0.io.netty.channel.epoll.EpollEventLoop.run(EpollEventLoop.java:368)
app//org.apache.beam.vendor.grpc.v1p69p0.io.netty.util.concurrent.SingleThreadEventExecutor$4.run(SingleThreadEventExecutor.java:994)
app//org.apache.beam.vendor.grpc.v1p69p0.io.netty.util.internal.ThreadExecutorMap$2.run(ThreadExecutorMap.java:74)
app//org.apache.beam.vendor.grpc.v1p69p0.io.netty.util.concurrent.FastThreadLocalRunnable.run(FastThreadLocalRunnable.java:30)

```

Thank you!

Questions? Feel free to reach out!

[linkedin.com/in/tomstepp](https://www.linkedin.com/in/tomstepp)

[linkedin.com/in/ryan-wigg](https://www.linkedin.com/in/ryan-wigg)

