

Throttle a PCollection

<https://s.apache.org/beam-throttle-transform>

Damon Douglas (damondouglas@apache.org)

Overview

Web API Quota overusage remains a key challenge when reading from and writing to Web APIs in a parallelized data processing context such as Beam. This document proposes a Throttle PTransform that holds element throughput to minimize downstream API overusage.

To those on the Beam learning journey, consider yourself a welcome participant in this document. Please visit <https://beam.apache.org/documentation/glossary/> for definitions on any terms and feel free to ask any questions you may have.

Goals

Throttle PTransform aims to

- Control a PCollection throughput to a configured maximum rate
- throttle both bounded and unbounded PCollections using a single PTransform

Non Goals

The new transform will not aim to co-mingle Web API calls with a throttling strategy as discussed here <https://sre.google/sre-book/handling-overload/> nor does it aim to replace this strategy either. In fact, it's designed to work upstream to transforms that perform such adaptive throttling; this document describes a preventive throttle.

Background

Many firms' rich data sources exist behind Web APIs of subscribed services designed to support application contexts with usage typically lower than that of large-scale data processing workloads such as Beam. Overcoming this challenge mainly requires reducing the throughput of an upstream PCollection that drives the Web API request rate performed by a Beam connector.

This transform must overcome Beam's design for scale and parallelization that translates to an as-fast-as-possible data processing. Essentially, it counters Beam's fundamental design. Yet, Beam's directed acyclic graph approach to processing and analytics outweighs the inherent risk of this transform's requirement.

This document outlines transform behavior, followed by a specification of configuration details, concluding with implementation details and a risk management strategy.

Transform Behavior

Using [gherkin syntax](#), the following outlines the Throttle PTransform behavior mostly with respect to the developer experience.

As a Beam developer, I want to slow down element throughput of an API request PCollection, so that I can minimize API overusage and save costs.

Scenario: input elements hypothetically assigned same Window during processing

Strategies the Transform uses (see implementation section below) require Beam strategies to share state between invocations of a DoFn's ProcessElement method. In Beam this is restricted, in part, to a per Window basis. In the following scenario, elements are hypothetically assigned to the same Window during processing.

Given Input PCollection elements assigned to same Window during pipeline run
And Transform configured to throttle at $n / \text{duration}$
When Transform processes PCollection
Then Output PCollection preserves Window strategy
And Transform emits elements at processing time intervals $\geq \text{duration}$

Scenario: input elements hypothetically assigned different Windows during processing; throttle achieved on a per Window basis

In this scenario, elements are hypothetically assigned different Windows during processing. While not a behavior fully controlled by the Beam developer, it's still important to highlight. The Window naive developer may observe that the throttling behavior doesn't behave as they expect or even at all. The state sharing tools the transform has is limited to, in part, a per Window basis and therefore so is throttling.

Given Input PCollection elements assigned different Window during processing
And Transform configured to throttle at $n / \text{duration}$
When Transform processes PCollection
Then Output PCollection preserves Window strategy
And emission intervals of elements in different Windows overlap
And emission intervals of elements in same Window $\geq \text{duration}$

Scenario: transform must process a small amount of data

Elements need not be throttled that are processed already at a rate less than configured.

Given Transform configured to throttle at $n / \text{duration}$
And Transform must process an amount of data $< n / \text{duration}$
When Transform processes PCollection
Then Transform emits elements at typical non-throttled rates

Scenario: transform must process a large amount of data

Given Transform configured to throttle at $n / \text{duration}$
And Transform must process an amount of data $\geq n / \text{duration}$
When Transform processes PCollection
Then Transform emits elements $\leq n / \text{duration}$

Scenario: upstream PCollection is bounded or unbounded

The two scenarios illustrate that throttling behavior from a Beam developer perspective is similar to both bounded and unbounded PCollections. See implementation details below for how bounded and unbounded PCollections are handled differently; the user of Throttle does not see nor should care.

Given Input PCollection bounded
And Transform configured to throttle at $n / \text{duration}$
When Transform processes PCollection
Then Transform emits elements $\leq n / \text{duration}$ until drained

Given Input PCollection unbounded
And Transform configured to throttle at $n / \text{duration}$
When Transform processes PCollection
Then Transform emits elements $\leq n / \text{duration}$ until pipeline stops

Scenario: pipeline drain invoked; PCollection unbounded

Given Input PCollection unbounded
When pipeline drain invoked
Then Transform throttles remaining elements existing in the pipeline

Transform configuration details

Parameter name	Parameter type	Description	Optional/Required
numElements	integer	The maximum number of elements to emit within the configured interval.	Required
interval	Duration	The interval, measured in time, within which to maximally throttle numElements.	Required
collectMetrics	boolean	Configures the transform to count inputs and outputs	Optional

Implementation details

The core idea of the transform is to restrain throughput of input data at a goal rate. To achieve this, Input elements are assigned a random key in $[0, \text{numElements})$, and then a subsequent transform throttles their emission during the desired interval.

The Transform uses different approaches to throttle **unbounded** and bounded PCollections. For **unbounded** PCollections, the Transform applies a [Stateful DoFn](#); for bounded PCollections it applies a [Splittable DoFn](#). The same strategy can't be applied to both bounded and **unbounded** PCollections.

- For **unbounded** PCollections, using State and Timers is the natural solution. However, processing time Timers do not trigger for bounded PCollections (see Appendix below); a Stateful DoFn's [OnTimer](#) method isn't invoked. Therefore, using Timers to control emission rate is not feasible for bounded PCollections.
- In a bounded PCollection, the Transform requires a GroupByKey to guarantee the desired rate. However, an **unbounded** PCollection will not trigger a GroupByKey in the global window because the watermark will never pass the window's end. Changing the input's windowing strategy leads to throttling in separate windows, potentially defeating the Transform's chief feature. Therefore, only bounded PCollections in this throttling context are compatible with Splittable DoFns.

In both approaches, each key emits a single element per interval, as keys put a constraint on the pipeline's parallelism. This is how the Transform achieves a rate of numElements per interval emission.

Using [gherkin syntax](#), the following outlines the Transform's code path and decision behavior at a lower level.

Transform Background:

The following is true of all scenarios below.

Given Transform configured with numElements and interval (see above)
And Transform expands and returns a PCollection<T>
And Transform contains a private Splittable DoFn<KV<Integer, List<T>>, T>
And Transform contains a private Stateful DoFn<KV<Integer, T>, T>

Scenario: Transform prepares input PCollection for downstream processing

Given Input PCollection <bounded_or_unbounded>
When Transform processes PCollection
Then randomly assigns keys [0, numElements)
And applies resulting KV<Integer, T> PCollection to <downstream>

Cases:

bounded_or_unbounded	downstream
bounded	GroupByKey
unbounded	Stateful DoFn

Splittable DoFn or Bounded Background:

The following is true for all Splittable DoFn scenarios.

Given RestrictionT is a Restriction that stores a from and to number
And Restriction's from must < to
And DoFn BoundedPerElement

Scenario: Runner invokes Splittable DoFn GetInitialRestriction

Given element List size of KV<Integer, List<T>> is <size_or_null>
When Runner invokes GetInitialRestriction method
Then method returns a restriction [-1, <restriction_size>)

Cases:

size_or_null	restriction_size
null	0
0	0

n	n
---	---

Scenario: Runner invokes Splittable DoFn GetInitialWatermarkEstimatorState

When Runner invokes GetInitialWatermarkEstimatorState method
Then method returns the minimum value for any Beam timestamp

Scenario: Runner invokes Splittable DoFn NewWatermarkEstimator

When Runner invokes NewWatermarkEstimator method
Then method returns a watermark estimator that is controlled manually

Scenario: Runner invokes Splittable DoFn TruncateRestriction

When Runner invokes TruncateRestriction method
Then method returns a TruncateResult instantiated with the restriction

Scenario: Runner invokes Splittable DoFn ProcessElement method

Scenario does not specify obvious cases when List is null or empty and method returns ProcessContinuation.stop.

Additionally, the scenario does not show cases when from is -1 where the returned value is set to 0 before further processing.

Given element List of KV<Integer, List<T>> is **not** null or empty
And the Restriction's from vs to properties are <from_vs_to>
When Runner invokes ProcessElement method
Then the RestrictionTracker tryClaim returns <try_claim_outcome>
And the watermark estimator is <set_or_not>
And the OutputReceiver <emit_or_not>
And the method returns <process_continuation>

Cases:

from_vs_to	try_claim_outcome	set_or_not	emit_or_not	process_continuation
from = to - 1	false	not	not	stop
from < to - 1	true	Set to current processing	emit	Resume with delay of throttling interval (see transform

		timestamp		configuration above)
--	--	-----------	--	----------------------

Stateful DoFn or Unbounded Background:

The following is true for all Stateful DoFn scenarios.

Given DoFn contains a ProcessTime TimeDomain Timer
And DoFn contains a List<T> State Value
And OnTimer annotated with Timer's ID
And DoFn contains a last emitted timestamp State Value

Scenario: DoFn determines whether to emit element

Given List<T> is <list_state>
And last emitted timestamp is <last_read_state>
When DoFn determines whether to emit element
Then the OutputReceiver <emit_or_not>
And the List<T> state is written with <list_write_state>
And last emitted timestamp is written with <last_write_state>

Cases:

list_state	last_read_state	emit_or_not	list_write_state	last_write_state
empty		not	clear	
not empty	current - last <= interval	not	list_state	
not empty	current - last > interval	emit	list_state	current

Scenario: DoFn determines whether to set timer

Given List<T> is <list_state>
And timer is <timer_read_state>
And last emitted timestamp is <last_read_state>
When DoFn determines whether to set timer
Then the timer is <timer_write_state>

Cases:

list_state	time_read_state	last_read_state	timer_write_state
empty			clear

not empty	not null		
not empty	null	current - last > interval	interval
not empty	null	current - last <= interval	interval - current - last

Scenario: Runner invokes ProcessElement method

When Runner invokes ProcessElement method
Then the DoFn determines whether to emit element
And the DoFn determines whether to set the timer

Scenario: Runner invokes OnTimer method

When Runner invokes OnTimer method
Then the DoFn determines whether to emit element
And the DoFn determines whether to set the timer

Risk management strategy

The following iterates through this transform's risks, potentially influencing variables of such risk. What follows is a discussion on the design of unit and load tests.

Loss of data

Most standard DoFns emit elements for every ProcessElement invocation. Yet, several factors drive this transform's DoFns' element emission, held by some container until their designed timing. Such lack of 1 ProcessElement invocation to 1 emission leads to risk that code errors simply fail to emit expected elements.

Variables to consider are the throughput of input data, upstream Window Strategy, and bounded vs unbounded sources. Changes in the data sizes may expose errors in how the code handles the Splittable DoFn restriction and restriction tracker. Different Window Strategies may expose code errors, for example, that held elements are garbage collected after window expiration. Finally, the use of bounded and unbounded sources routes the two main code paths i.e. DoFns that pose their own unique risks.

Over/under throttling

As this transform inherently works against Beam's fundamental parallelization and as-fast-as-possible design goals, code errors translate to not achieving its chief aim. Correspondingly, in cases when the number of elements does not exceed the goal rate, the code errors fail by holding them unnecessarily.

As with loss of data, variables to consider are the throughput of input data, upstream Window Strategy, and bounded vs unbounded sources. Smaller data sizes may expose code errors in over throttling, while larger may expose under. Additionally, the Window Strategy validates documented limitations of the transform to only

throttle elements of the same window assignment where state associates. Finally, bounded and unbounded input PCollections have distinct execution paths.

Unwanted Windowing effects

This transform will not apply or alter the Window Strategy in any way. However, it's worth evaluating whether code errors lead to unwanted behavior in the element Window assignments and triggering. In addition to immediate unwanted effects to the transform's output PCollection, there may be unexpected outcomes to subsequently applied Window Strategies. Potential influencers are the upstream Watermarks, Window Strategy, and bounded vs unbounded sources. Additionally, the Splittable DoFn code influences the Watermark that may lead to unwanted effects to element Window assignments and triggering.

Measures and assertions

Test code can use existing tools to assert data integrity i.e. `PAssert.containsInAnyOrder` in Java. Evaluating over and under throttling requires measuring the timestamps of resulting elements and their in-between duration intervals. To reduce flakiness, a marginal expected error should be added or subtracted from the expected interval. Finally, assessing Windowing outcomes, tests should also use existing tools. For example, in Java, `PAssert.inWindow()` or `PAssert.inWindow().containsInAnyOrder`, etc.

TestStream considerations

It's important to call out that Java's `ProcessContinuation.resume.withDelay` does actual resume again with delay when using `TestStream`. Previous attempts to use this combination revealed that the Clock was not set to the current process time. Therefore, to generate unbounded sources, tests could simply use a real streaming source such as `PeriodicImpulse` with a end duration sensible for expected test execution times. `TestStream` may be helpful when evaluating the effects of upstream watermarks.

Unit tests

Unit tests should focus on code correctness using sizes of data and time ranges practical within expected test code execution durations. The categories of tests should follow the aforementioned risks to systematically evaluate varying data sizes, Window Strategies both upstream and downstream, and unbounded and bounded sources.

Load tests

Load tests evaluate the same as unit tests but under larger data loads using a non-local runner such as Dataflow. Under Dataflow, for example, tests will want to analyze the impact of increasing workers. Such tests may find that increasing data sizes and workers will expose failures to throttle. Additionally, memory use metrics may show unwanted increases along with additionally varying per element memory sizes.

Appendix

Preliminary data comparing PCollection boundedness and time domain DoFns

The reason for the following is to use an empirical approach to advocate that the state and timer approach is not compatible with bounded PCollection sources. Additionally, it preliminarily suggests that for **unbounded** sources, processing time based timers may serve as a tool to achieve the aim of the Throttle transform.

Four Java pipelines (Beam SDK version 2.53.0) were executed on Java's local runner and Dataflow to compare the emitted timestamps influenced by boundedness and the [TimeDomain](#) configured for the DoFn's Timer. The design of these pipelines is not to suggest any similarity with what was discussed above. They are meant to isolate the impact of PCollection boundedness and the TimeDomain employed by a DoFn on its OnTimer method invocation.

Pipeline applies a generated single-key KV PCollection to a DoFn configured with a Timer. These four pipelines varied on the TimeDomain, EVENT_TIME vs PROCESSING_TIME and whether the generated KV PCollection was bounded or **unbounded**.

In each pipeline, the timer configured DoFn emitted elements within its DoFn.OnTimer method, followed by a downstream DoFn that measured emitted timestamps within its ProcessElement method. Finally, the measured timestamps were logged with the duration elapsed since a static starting timestamp. It is easier to read the output this way and compare between pipelines. If OnTimer triggers on a configured interval (1s was used for each example), we would anticipate that, at least for processing time, the elapsed times would be roughly 1s from each other. Event time timer domains were compared for completeness of the experiment.

For bounded PCollection sources (i.e. batch), the PROCESSING_TIME Timer did not trigger its OnTimer method; the logs had no emitted timestamps. However, for the EVENT_TIME Timer, its OnTimer method triggered (we see output), however, the intervals between the emitted timestamps didn't exceed 0.02s. This suggests that despite `timer.offset(interval).setRelative()`, where interval was 1s, we see something much less. We would expect this as the trigger is based on the element timestamps.

For **unbounded** PCollection sources, both EVENT_TIME and PROCESSING_TIME Timer configured DoFns emitted data from their OnTimer method. However, as expected, the PROCESSING_TIME Timer configured DoFn's emitted intervals were more consistent and aligned with expectations. We mostly see an approximate 1s interval between these timestamps; sometimes there's an approximate 2s gap.

The table below summarizes the results as well as provides links to the four pipelines' gists.

TimeDomain	Boundedness	Abbreviated sample output	Gist
------------	-------------	---------------------------	------

EVENT_TIME	bounded	None PT1.120S: 234 PT1.143S: 235 PT1.144S: 236	https://gist.github.com/damondouglas/fcbbd39ecb9abdacce39eab18794cf63
PROCESSING_TIME	bounded	No output.	https://gist.github.com/damondouglas/bff0ef9c56fb7f1eaa5d24d1c2931d7d
EVENT_TIME	un bounded	None PT926.843S: 12465 PT929.045S: 12490	https://gist.github.com/damondouglas/e2ea7d0d50dc4453eb10108ed1929684
PROCESSING_TIME	un bounded	None PT466.068S: 11256 PT467.370S: 11281 PT469.121S: 11306 PT470.373S: 11331 PT471.991S: 11356	https://gist.github.com/damondouglas/8bc283b9fb6e6d316d83487a7cfed64d