# Apache flink rate limiting

Tom Rochette <tom.rochette@coreteks.org>

December 21, 2025 — 77e1b28a

In this article we'll cover adding rate limiting to an Apache Flink pipeline. While Apache Flink already contains some APIs that implement some form of rate limiting indirectly, such as `AsyncDataStream` through a capacity limit which limits the number of concurrent executions, these do not rate limit directly.

Rate limiting is critical in applications where calling 3rd party APIs with rate limits or quotas could result in multiple unnecessary retries or even failures to process an event successfully.

- Implementation of rate limiting
  - This consist mostly in including a call to a blocking implementation of a rate limiter just before you would execute the code you want rate limited. For example if you are rate limiting calls to an API, you would block just before sending your HTTP request.
- Rate limiting in relation to parallelism
  - When defining rate limiting we generally think of the overall rate limit per second. With Apache Flink parallelism, this number ends up being used by each instance, which makes the rate limit useless in this situation, unless it is divided by the number of parallel instances running the code that is rate limited.
  - One way to implement this local rate limit is given in the `open` function in Flink GuavaFlinkConnectorRateLimiter by dividing the global rate limit by `runtimeContext.getTaskInfo().getNumberOfParallelSubtas`
  - In a scenario where we could increase the parallelism indefinitely, what will determine if we autoscale will be how busy the vertex containing our rate limiting operation is. If we have a rate limit of 100/s, it will be 100% busy as soon as we're processing that many and are rate limited.
  - In most situations however we will want to cap our total rate so we will need to configure a maximum parallelism.
  - I've always thought that we should be making use of the full rate limit we're allowed, such that even with a parallelism of 1, if our rate limit is 1000/s, that is our rate limit. If the vertex containing the rate limited operation scales to 2, the rate limit of each operator would now be 500.
    - Earlier we mentioned that busyness was used to determine when to scale. In this scenario, it seems we would only scale once we reached 1000/s. But then scaling would not help us, as we would simply be 100% busy but on more instances.
    - This approach only makes sense if scaling happens for a different reason than rate limiting, such as getting capacity limited or because we cannot reach the rate limit on a single instance. Another reason could be that the rest of the operations in the vertex make it highly busy.
  - The downside of specifying a rate limit that is smaller than the total rate limit you have is that you will need to scale possibly unnecessarily because the instance could have handled all the rate limited operations of the global limit.
  - What options do we have?
    - Keep the rate limited in the same vertex and divide the global rate limit by the number of parallel subtasks
      - This allows us to benefit from lower overhead communication between operators
    - Isolate the rate limited operation in its own chain
      - This allows the operation to scale independently of what happens before and after.
      - The downside is increased overhead to transfer the data between the chains before and after.

* Partially isolate the operation in a new chain (either with the operators before or after)
  - It's a trade-off of the benefits of full isolation with no isolation, namely that overhead is partially reduced, but you may affect the parallelism of other operations within the same vertex.