监控Apache Flink应用程序(入门)

caolei

Exported on 01/10/2020

Table of Contents

原文地址：https://www.ververica.com/blog/monitoring-apache-flink-applications-101

这篇博文介绍了Apache Flink内置的监控和度量系统，通过该系统，开发人员可以有效地监控他们的Flink作业。通常，对于一个刚刚开始使用Apache Flink进行流处理的DevOps团队来说，选择对应的指标来监控Flink应用程序是非常艰巨的。在与许多大规模部署过Apache Flink的组织合作之后，我想与社区的朋友们分享下我的经验及一些最佳实践。

随着越来越多的核心业务应用程序运行在Apache Flink上，性能监控在成功的生产环境部署中变得非常重要。它确保何故障或停机时间都可以被立即识别并尽快得到解决。

监控与观察相结合是故障诊断和性能调优的先决条件。如今，随着现代企业应用程序的复杂性和交付速度的加快，工程团队必须理解并在任何给定的时间点上对其应用程序的状态有一个完整的认识和概述。

1 Flink指标体系

Flink作业监控的基础是它的度量系统，该系统由两个部分组成: Metrics和MetricsReporters。

1.1 Metrics

Flink提供了一套全面的内置Metrics:

- JVM堆/非堆/直接内存的使用情况(任务粒度)
- 作业重启次数(作业粒度)
- 每秒处理的数据量(操作符粒度)
- ……

作为用户，您可以并且应该向函数中添加应用程序相关的metrics。这些metrics包括无效记录的counter或托管状态下临时缓冲记录的counter等。除了counters之外，Flink还提供了其他类型的metrics，比如gauges和histograms。关于如何向Flink的metrics系统注册您自己的metrics，请参阅Flink文档[1]。在这篇博客中，我们将重点讨论如何最大限度地利用Flink的内置metrics。

1.2 MetricsReporters

所有metrics都可以通过Flink提供的REST API来查询。但是，用户可以配置MetricsReporters将metrics发送到外部系统。Apache Flink为最常用的开箱即用的监控工具(JMX、Prometheus、Datadog、Graphite和InfluxDB)提供报告程序。有关如何配置报告程序的信息，请参阅Flink的MetricsReporter文档[2]。

在这篇博客的其余部分中，我们将介绍一些监控Apache Flink应用程序的最重要的指标。

1 https://ci.apache.org/projects/flink/flink-docs-release-1.7/monitoring/metrics.html#registering-metrics
2 https://ci.apache.org/projects/flink/flink-docs-release-1.7/monitoring/metrics.html#reporter

2 健康状况

3 监控

您要监控的第一件事就是您的作业是否实际处于运行状态。此外，还可以监控重启的次数以及自上次重启之后的时间。

通常来说，成功的检查点是应用程序总体健康状况的一个强大指示器。对于每个检查点，检查点屏障需要流经Flink作业的整个拓扑结构，并且事件和屏障不能相互超越。因此，一个成功的检查点显示没有通道是完全拥挤的。

3.1 关键指标

指标	范围	描述
uptime	job	The time that the job has been running without interruption.
fullRestarts	job	The total number of full restarts since this job was submitted.
numberOfCompletedCheckpoints	job	The number of successfully completed checkpoints.
numberOfFailedCheckpoints	job	The number of failed checkpoints.

3.2 仪表盘示例

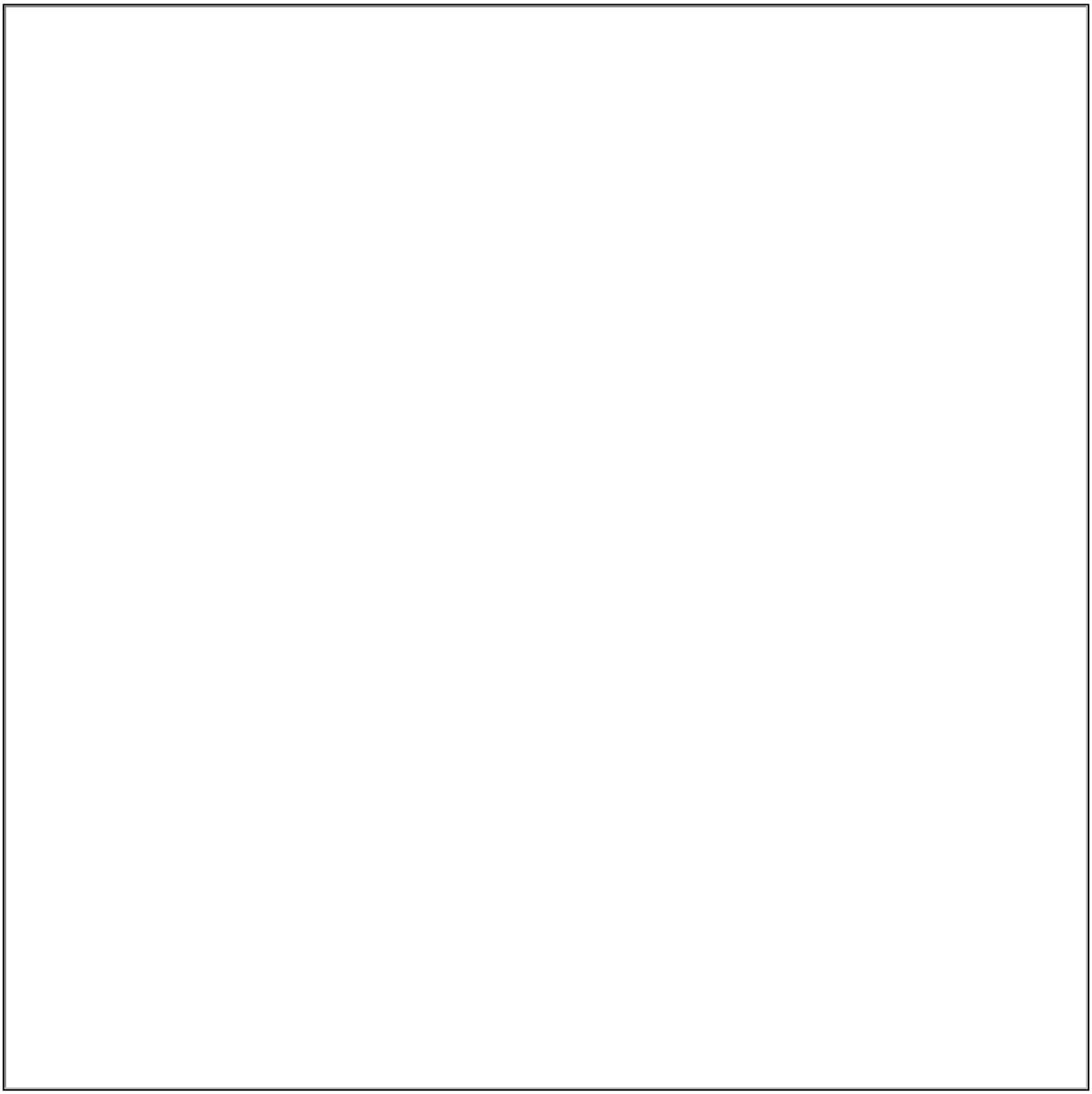

Figure 1: Uptime (35 minutes), Restarting Time (3 milliseconds) and Number of Full Restarts (7)

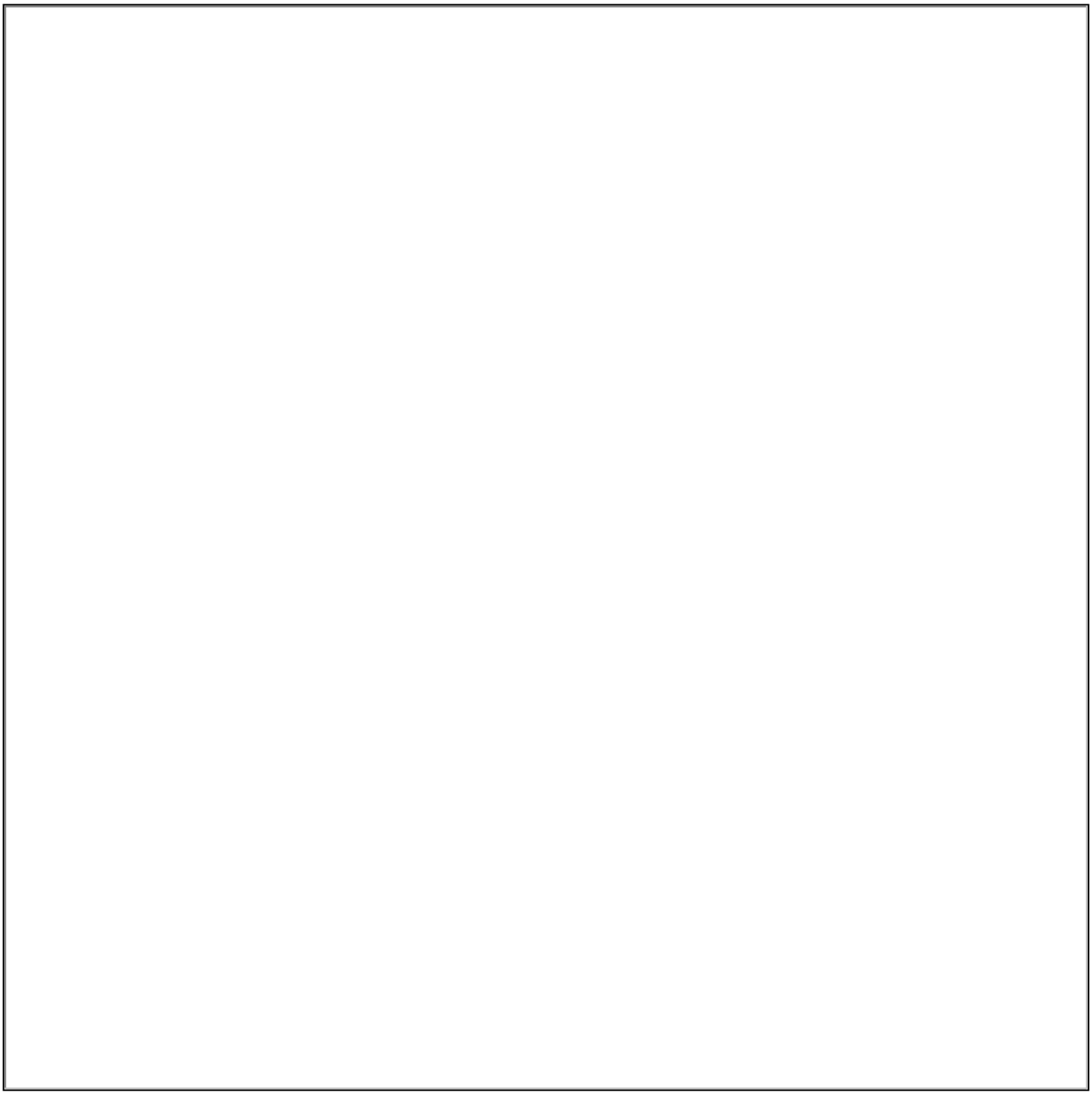

Figure 2: Completed Checkpoints (18336), Failed (14)

3.3 可能的报警条件

- fullRestarts > threshold
- numberOfFailedCheckpoints > threshold

4 进度和吞吐量监控

知道您的应用程序正在运行并且检查点正常工作是件好事，但是它并不能告诉您应用程序是否正在实际取得进展并与上游系统保持同步。

4.1 吞吐量

Flink提供了多个metrics来衡量应用程序的吞吐量。对于每个operator或task(请记住:一个task可以包含多个chained-task[3]），Flink会对进出系统的记录和字节进行计数。在这些metrics中，每个operator输出记录的速率通常是最直观和最容易理解的。

4.2 关键指标

Metric	Scope	Description
numRecordsOutPerSecond	task	The number of records this operator/task sends per second.
numRecordsOutPerSecond	operator	The number of records this operator sends per second.

3 https://ci.apache.org/projects/flink/flink-docs-release-1.7/dev/stream/operators/#task-chaining-and-resource-groups

4.3 仪表盘示例

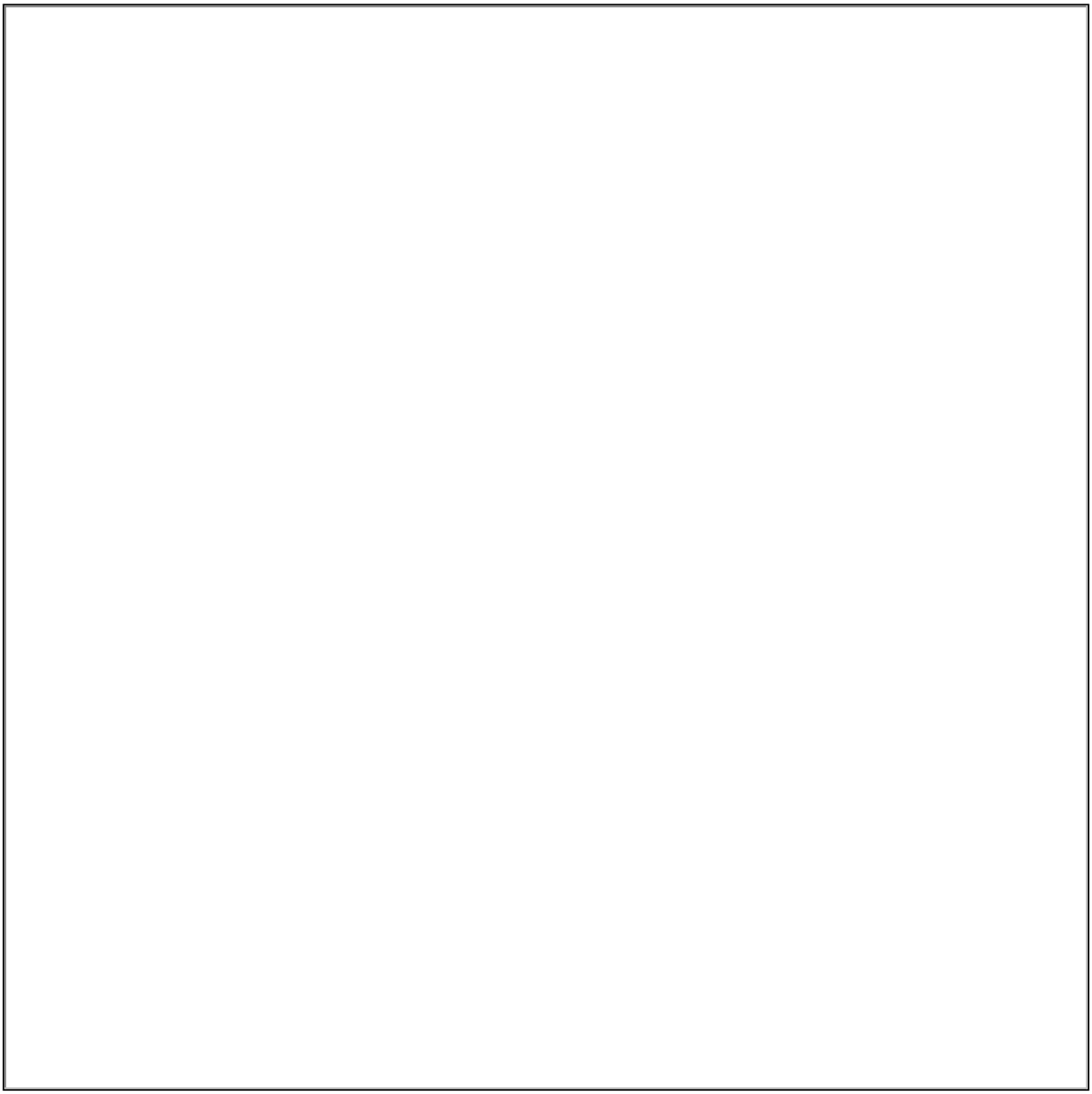

Figure 3: Mean Records Out per Second per Operator

4.4 可能的报警条件

- recordsOutPerSecond = 0 (for a non-Sink operator)

请注意：目前由于metrics体系只考虑Flink的内部通信，所以source operators的输入记录数是0，而sink operators的输出记录数也是0.

4.5 进度

对于使用事件时间语义的应用程序来说，watermarks随着时间的推移而变化是非常重要的。watermarks的时间t表名框架再也不应该期望接收到时间戳比t早的事件了,相反,那些时间戳小于t的operations将会被触发的触发。例如,当watermarks超过30时，结束于t = 30的事件时间窗口将被关闭并计算。

因此，您应该在应用程序中对事件时间敏感的operators(如流程函数和窗口)上监控watermarks。如果当前处理时间与被称为 even-time skew的watermarks之间的差异非常高，那么它通常意味着可能会出现两种情况。首先，它可能意味着您只是在处理旧的事件，例如在停机后的追赶期间，或者当您的工作无法继续，而事件正在排队时。其次，它可能意味着单个上游子任务很长时间没有发送watermarks(例如因为它没有收到任何基于watermarks的事件)，这也阻止了下游操作符中的watermarks的进展。

4.6 关键指标

Metric	Scope	Description
currentOutputWatermark	operator	The last watermark this operator has emitted

4.7 仪表盘示例

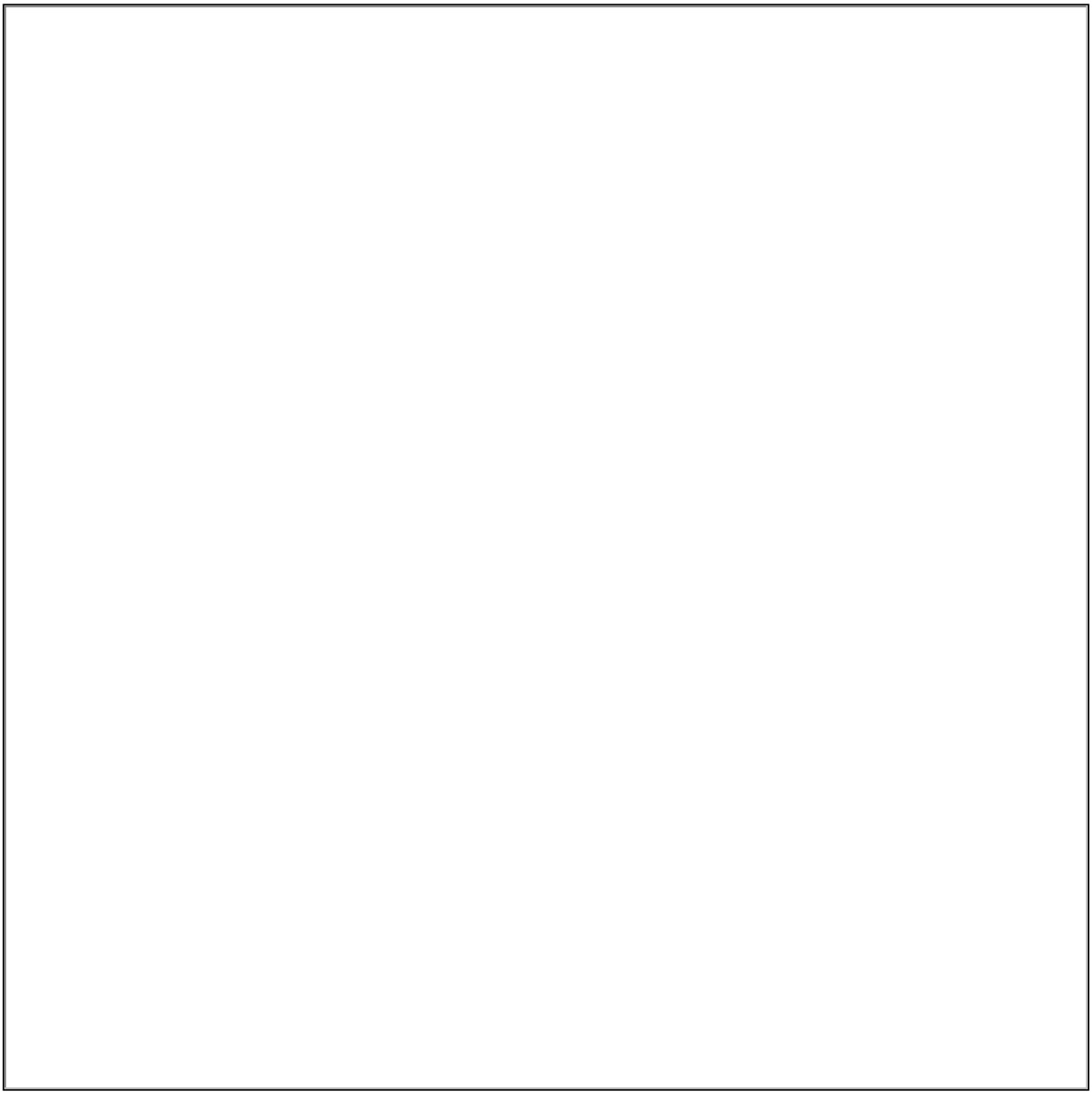

Figure 4: Event Time Lag per Subtask of a single operator in the topology. In this case, the watermark is lagging a few seconds behind for each subtask.

4.8 可能的报警条件

- 4.8.1 currentProcessingTime - currentOutputWatermark > threshold

4.9 "Keeping Up"

当从消息队列中消费消息时，通常有一种直接的方法来监控应用程序是否正常运行。通过使用特定于连接器的metrics，您可以监视当前消费者组的消息队列头部的落后程度。Flink可以从大多数source获得基本metrics。

4.10 关键指标

Metric	Scope	Description
records-lag-max	user	applies to **FlinkKafkaConsumer** The maximum lag in terms of the number of records for any partition in this window. An increasing value over time is your best indication that the consumer group is not keeping up with the producers.
millisBehindLatest	user	applies to **FlinkKinesisConsumer** The number of milliseconds a consumer is behind the head of the stream. For any consumer and Kinesis shard, this indicates how far it is behind the current time.

4.11 可能的报警条件

- records-lag-max > threshold
- millisBehindLatest > threshold

4.12 Monitoring Latency

Generally speaking, latency is the delay between the creation of an event and the time at which results based on this event become visible. Once the event is created it is usually stored in a persistent message queue, before it is processed by Apache Flink, which then writes the results to a database or calls a downstream system. In such a pipeline, latency can be introduced at each stage and for various reasons including the following:

1. It might take a varying amount of time until events are persisted in the message queue.

2. During periods of high load or during recovery, events might spend some time in the message queue until they are processed by Flink (see previous section).
3. Some operators in a streaming topology need to buffer events for some time (e.g. in a time window) for functional reasons.
4. Each computation in your Flink topology (framework or user code), as well as each network shuffle, takes time and adds to latency.
5. If the application emits through a **transactional** sink, the sink will only commit and publish transactions upon successful checkpoints of Flink, adding latency usually up to the checkpointing interval for each record.

In practice, it has proven invaluable to add timestamps to your events at multiple stages (at least at creation, persistence, ingestion by Flink, publication by Flink; possibly sampling those to save bandwidth). The differences between these timestamps can be exposed as a user-defined metric in your Flink topology to derive the latency distribution of each stage.

In the rest of this section, we will only consider latency, which is introduced inside the Flink topology and cannot be attributed to transactional sinks or events being buffered for functional reasons (4.).

To this end, Flink comes with a feature called Latency Tracking[4]. When enabled, Flink will insert so-called latency markers periodically at all sources. For each sub-task, a latency distribution from each source to this operator will be reported. The granularity of these histograms can be further controlled by setting *metrics.latency.granularity* as desired.

Due to the potentially high number of histograms (in particular for *metrics.latency.granularity: subtask*), enabling latency tracking can significantly impact the performance of the cluster. It is recommended to only enable it to locate sources of latency during debugging.

4.12.1 Key Metrics

Metric	Scope	Description
latency	operator	The latency from the source operator to this operator.
restartingTime	job	The time it took to restart the job, or how long the current restart has been in progress.

4 https://ci.apache.org/projects/flink/flink-docs-release-1.7/monitoring/metrics.html#latency-tracking

4.12.2 Example Dashboard Panel

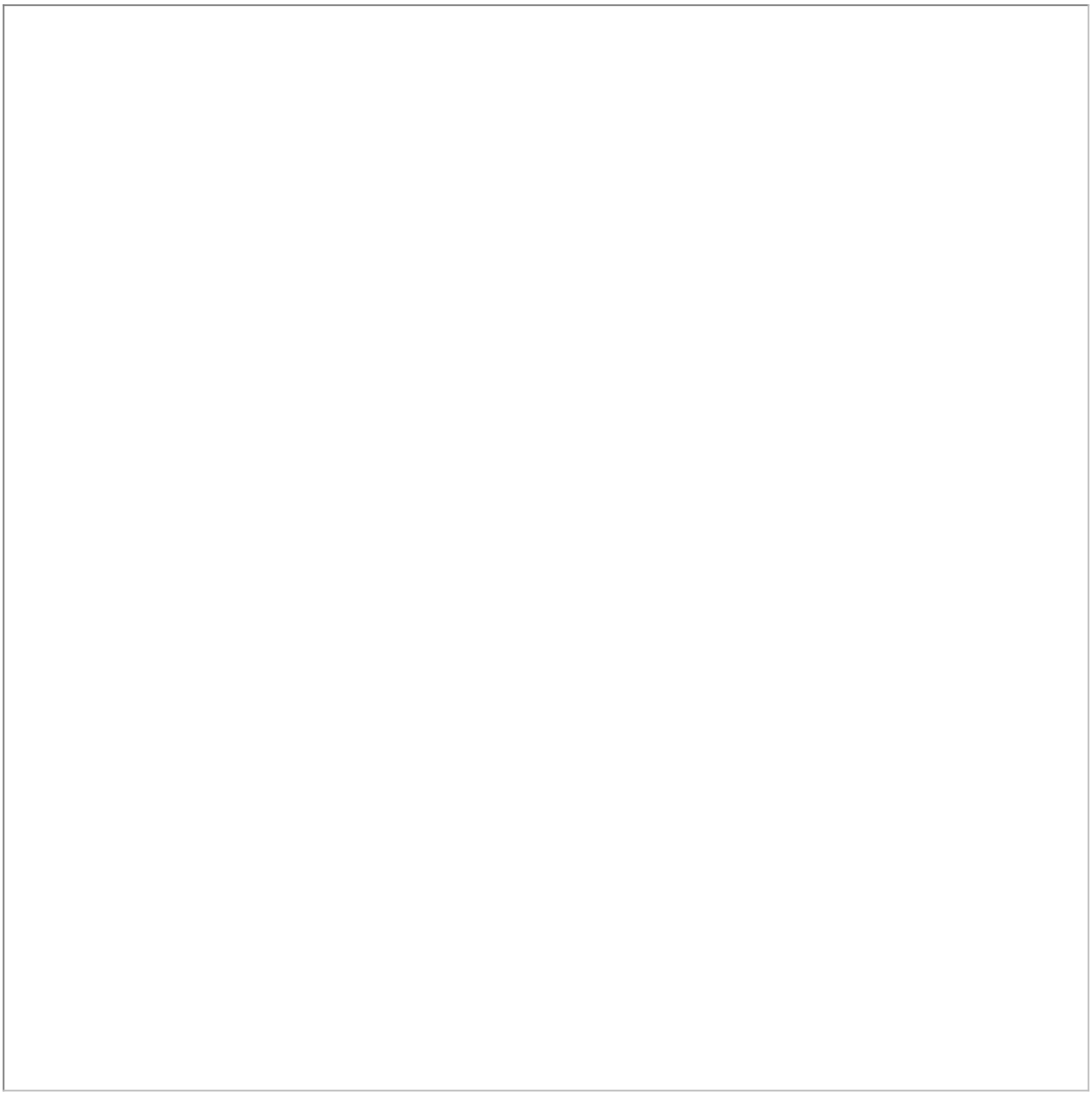

Figure 5: Latency distribution between a source and a single sink subtask.

4.13 JVM Metrics

So far we have only looked at Flink-specific metrics. As long as latency & throughput of your application are in line with your expectations and it is checkpointing consistently, this is probably everything you need. On the other

hand, if you job's performance is starting to degrade among the first metrics you want to look at are memory consumption and CPU load of your Task- & JobManager JVMs.

4.13.1 Memory

Flink reports the usage of Heap, NonHeap, Direct & Mapped memory for JobManagers and TaskManagers.

- Heap memory - as with most JVM applications - is the most volatile and important metric to watch. This is especially true when using Flink's filesystem statebackend as it keeps all state objects on the JVM Heap. If the size of long-living objects on the Heap increases significantly, this can usually be attributed to the size of your application state (check the checkpointing metrics[5] for an estimated size of the on-heap state). The possible reasons for growing state are very application-specific. Typically, an increasing number of keys, a large event-time skew between different input streams or simply missing state cleanup may cause growing state.
- NonHeap memory is dominated by the metaspace, the size of which is unlimited by default and holds class metadata as well as static content. There is a JIRA Ticket[6] to limit the size to 250 megabyte by default
- The biggest driver of Direct memory is by far the number of Flink's network buffers, which can be configured[7].
- Mapped memory is usually close to zero as Flink does not use memory-mapped files.

In a containerized environment you should additionally monitor the overall memory consumption of the Job- and TaskManager containers to ensure they don't exceed their resource limits. This is particularly important, when using the RocksDB statebackend, since RocksDB allocates a considerable amount of memory off heap. To understand how much memory RocksDB might use, you can checkout this blog post[8] by Stefan Richter.

4.13.1.1 Key Metrics

Metric	Scope	Description
Status.JVM.Memory.NonHeap.Committed	job-/ taskmanager	The amount of non-heap memory guaranteed to be available to the JVM (in bytes).
Status.JVM.Memory.Heap.Used	job-/ taskmanager	The amount of heap memory currently used (in bytes).
Status.JVM.Memory.Heap.Committed	job-/ taskmanager	The amount of heap memory guaranteed to be available to the JVM (in bytes).

5 https://ci.apache.org/projects/flink/flink-docs-release-1.7/monitoring/metrics.html#checkpointing
6 https://issues.apache.org/jira/browse/FLINK-10317
7 https://ci.apache.org/projects/flink/flink-docs-release-1.7/ops/config.html#configuring-the-network-buffers
8 https://www.da-platform.com/blog/manage-rocksdb-memory-size-apache-flink?__hstc=216506377.c9dc814ddd168ffc714fc8d2bf20623f.1550652804788.1550652804788.1550652804788.1&__hssc=216506377.3.1551426921706&__hsfp=3017175250

Status.JVM.Memory.Direct.MemoryUsed	job-/ taskmanager	The amount of memory used by the JVM for the direct buffer pool (in bytes).
Status.JVM.Memory.Mapped.MemoryUsed	job-/ taskmanager	The amount of memory used by the JVM for the mapped buffer pool (in bytes).
Status.JVM.GarbageCollector.G1 Young Generation.Time	job-/ taskmanager	The total time spent performing G1 Young Generation garbage collection.
Status.JVM.GarbageCollector.G1 Old Generation.Time	job-/ taskmanager	The total time spent performing G1 Old Generation garbage collection.

4.13.1.2 Example Dashboard Panel

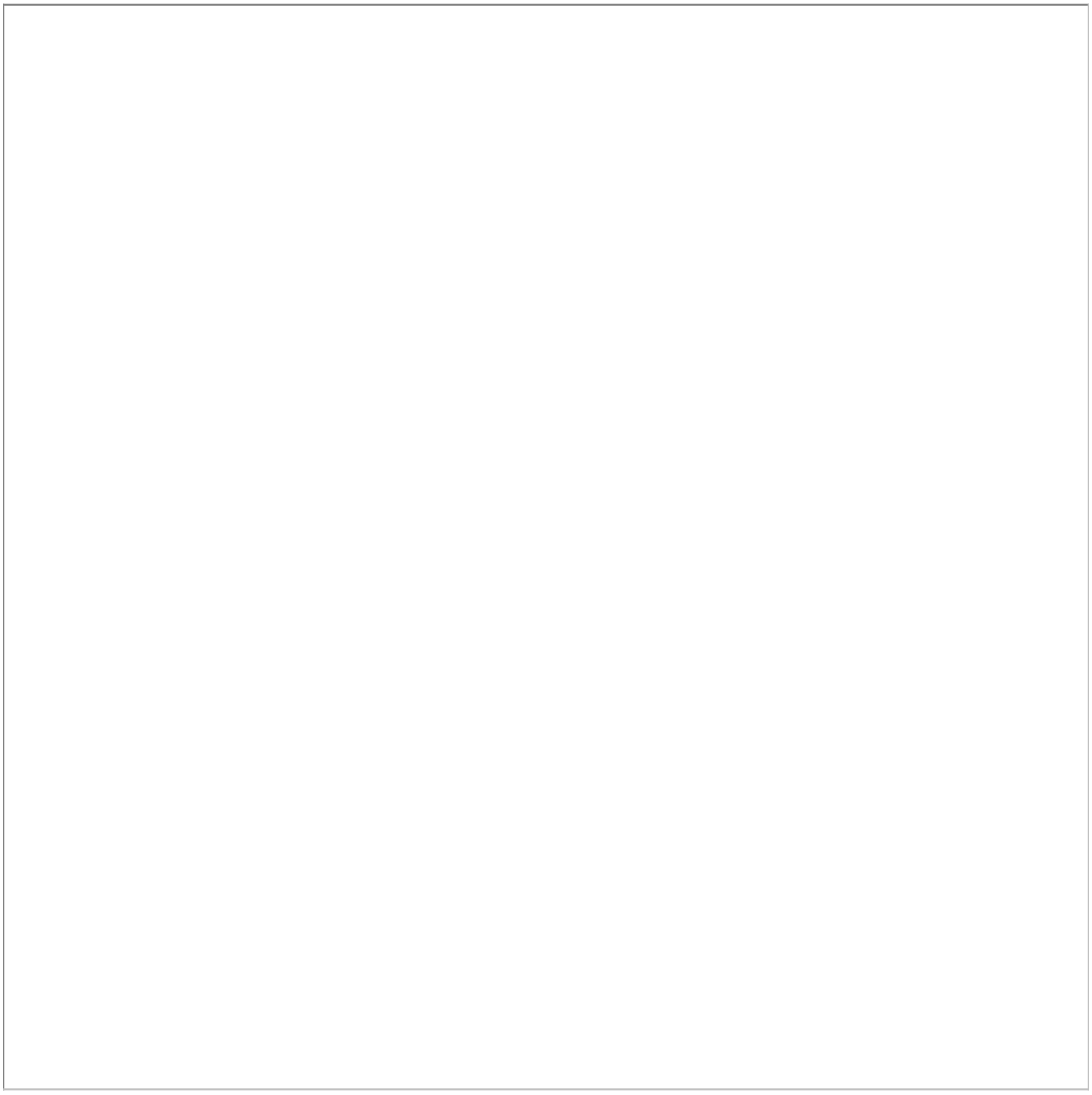

Figure 6: TaskManager memory consumption and garbage collection times.

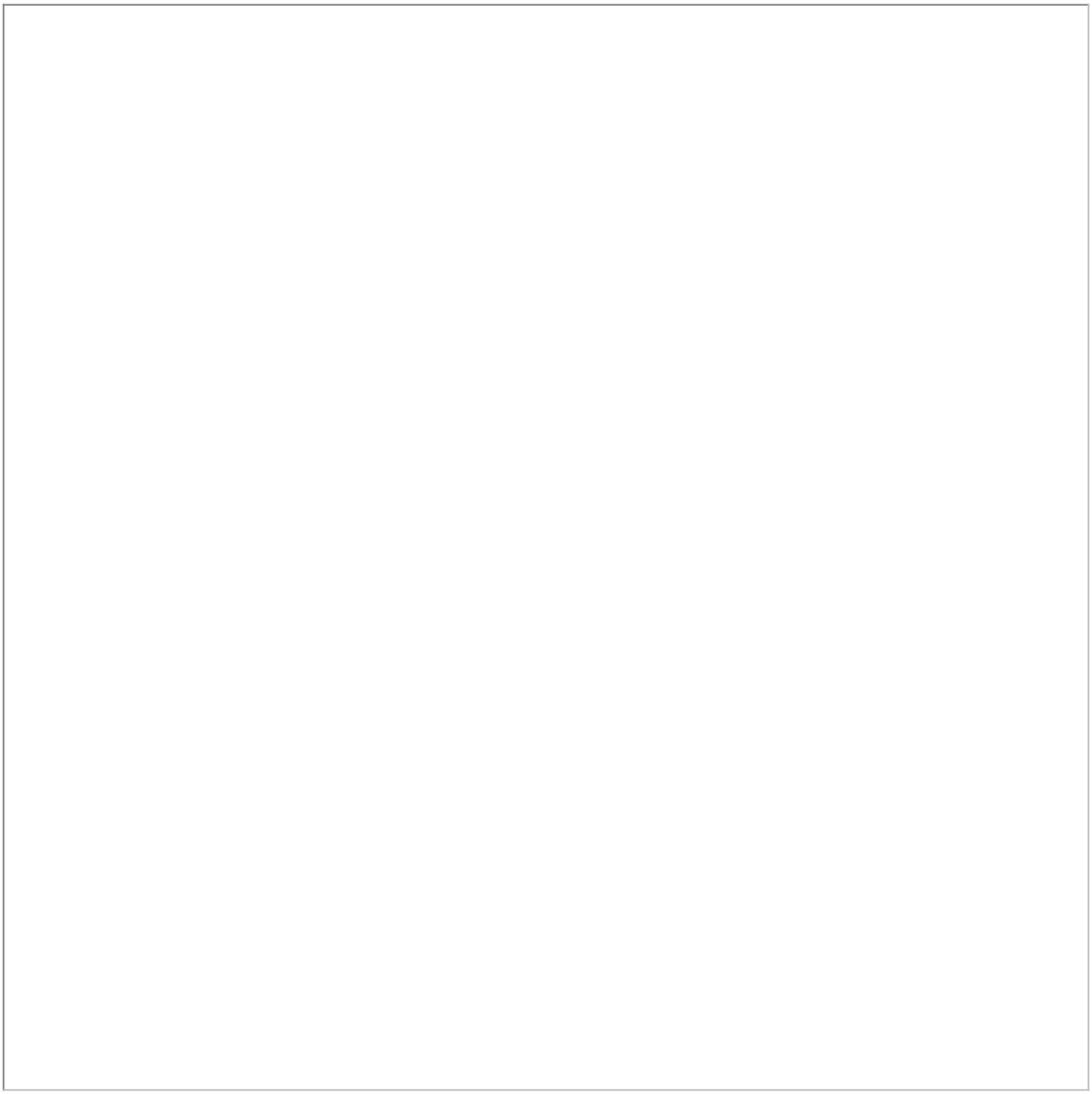

Figure 7: JobManager memory consumption and garbage collection times.

4.13.1.3 Possible Alerts

- container memory limit < container memory + safety margin

4.13.2 CPU

Besides memory, you should also monitor the CPU load of the TaskManagers. If your TaskManagers are constantly under very high load, you might be able to improve the overall performance by decreasing the number of task slots per TaskManager (in case of a Standalone setup), by providing more resources to the TaskManager (in case of a containerized setup), or by providing more TaskManagers. In general, a system already running under very high load during normal operations, will need much more time to catch-up after recovering from a downtime. During this time you will see a much higher latency (event-time skew) than usual.

A sudden increase in the CPU load might also be attributed to high garbage collection pressure, which should be visible in the JVM memory metrics as well.

If one or a few TaskManagers are constantly under very high load, this can slow down the whole topology due to long checkpoint alignment times and increasing event-time skew. A common reason is skew in the partition key of the data, which can be mitigated by pre-aggregating before the shuffle or keying on a more evenly distributed key.

4.13.2.1 Key Metrics

Metrics	Scope	Description
Status.JVM.CPU.Load	job-/ taskmanager	The recent CPU usage of the JVM.

4.13.2.2 Example Dashboard Panel

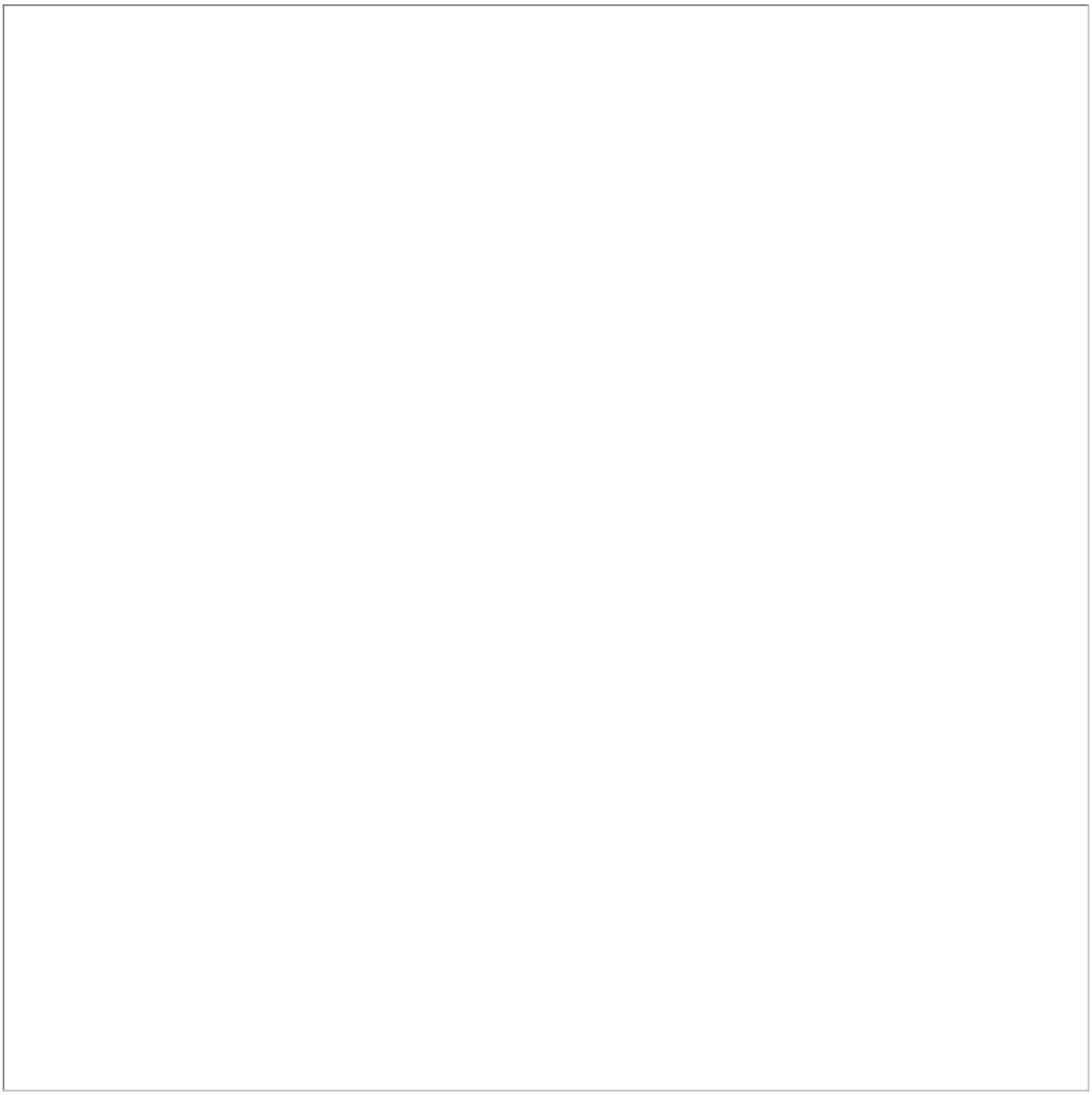

Figure 8: TaskManager & JobManager CPU load

4.14 System Resources

In addition to the JVM metrics above, it is also possible to use Flink's metrics system to gather insights about system resources, i.e. memory, CPU & network-related metrics for the whole machine as opposed to the Flink processes alone. System resource monitoring is disabled by default and requires additional dependencies on the classpath. Please check out the Flink system resource metrics documentation[9] for additional guidance and details. System resource monitoring in Flink can be very helpful in setups without existing host monitoring capabilities.

4.15 Conclusion

This post tries to shed some light on Flink's metrics and monitoring system. You can utilise it as a starting point when you first think about how to successfully monitor your Flink application. I highly recommend to start monitoring your Flink application early on in the development phase. This way you will be able to improve your dashboards and alerts over time and, more importantly, observe the performance impact of the changes to your application throughout the development phase. By doing so, you can ask the right questions about the runtime behaviour of your application, and learn much more about Flink's internals early on.

Last but not least, this post only scratches the surface of the overall metrics and monitoring capabilities of Apache Flink. I highly recommend going over Flink's metrics documentation[10] for a full reference of Flink's metrics system.

9 https://ci.apache.org/projects/flink/flink-docs-release-1.7/monitoring/metrics.html#system-resources
10 https://ci.apache.org/projects/flink/flink-docs-release-1.7/monitoring/metrics.html