Acta Univ. Sapientiae Informatica **13,** 1 (2021) 39–59

DOI: 10.2478/ausi-2021-0003

Towards autoscaling of Apache Flink jobs

Balázs VARGA
ELTE Eötvös Loránd University
Budapest, Hungary
email: balazsvarga@student.elte.hu

Márton BALASSI
Cloudera
Budapest, Hungary
email: mbalassi@cloudera.com

Attila KISS
J. Selye University
Komárno, Slovakia
email: kissae@ujs.sk

Abstract. Data stream processing has been gaining attention in the past decade. Apache Flink is an open-source distributed stream processing engine that is able to process a large amount of data in real time with low latency. Computations are distributed among a cluster of nodes. Currently, provisioning the appropriate amount of cloud resources must be done manually ahead of time. A dynamically varying workload may exceed the capacity of the cluster, or leave resources underutilized. In our paper, we describe an architecture that enables the automatic scaling of Flink jobs on Kubernetes based on custom metrics, and describe a simple scaling policy. We also measure the effects of state size and target parallelism on the duration of the scaling operation, which must be considered when designing an autoscaling policy, so that the Flink job respects a Service Level Agreement.

1 Introduction

Apache Flink [5, 18, 10] is an open-source distributed data stream processing engine and framework. It can perform computations on both bounded and

Computing Classification System 1998: C.2.4
Mathematics Subject Classification 2010: 68M14
Key words and phrases: Apache Flink, autoscaling, data stream processing, big data, kubernetes, distributed computing

unbounded data streams using various APIs offering different levels of abstraction. A Flink application consists of a streaming pipeline, which is a directed graph of operators performing computations as nodes, and the streaming of data between them as edges.

Flink applications can handle large state in a consistent manner. Most production jobs make use of stateful operators that can store internal state via various state backends, such as in-memory or on disk. Flink has an advanced checkpointing and savepointing mechanism to create consistent snapshots of the application state, which can be used to recover from failure or to restart the application with an existing state [4, 3].

These streaming jobs are typically long-running, their usage may span weeks or months. In these cases, the workload may change over time. The application must handle the changed demands while meeting the originally set service level agreement (SLA). This changing demand may be predictable ahead of time, in case some periodicity is known, or there are events that are known to influence the workload, but in other cases, it is bursty and unpredictable. Statically provisioning resources and setting the job's parallelism at launch-time is unsuited for these long-running jobs. If too few resources are allocated (under-provisioning), the application will not keep up with the increasing workload, and start missing SLAs. If the resources are provisioned to match the predicted maximum load, the system will run over-provisioned most of the time, not utilizing the resources efficiently, and incurring unnecessary cloud costs.

Flink jobs' parallelism can not be changed during runtime. It is possible however to take a savepoint, then restart the job with a different parallelism from the snapshot. If the job is running in the cloud, it is also possible at this point to provision (or unprovision) additional resources, new instances that can perform computations. This is called horizontal scaling.

Restarting a job is an expensive operation. The state must be written to a persistent storage beforehand, which can be done asynchronously, but restoring from this savepoint after the restart can take a considerable amount of time. Meanwhile, the incoming workload is not being processed, so the restarted application has to catch up with this additional delay. Scaling decisions should therefore be made wisely. We must monitor various metrics of the running job, take into account the delays allowed by the SLA, and decide whether the trade-off of the scaling is worth it. Algorithms that make this decision automatically, reacting to the changing load dynamically, and performing the actual scaling operation are of great value, and make the operations of long-running streaming applications feasible and efficient.

Container orchestrators such as Kubernetes [9] allow us to both automate the mechanics of the scaling process, and to implement the custom algorithms that make the decisions. We have set up a system that can perform these scaling operations using Kubernetes' Horizontal Pod Autoscaler resource [20] and Google's open-source Flink operator [21].

In this paper, we discuss the architecture of this system. We describe a simple scaling policy that we have implemented, that is based on operator idleness and changes of the input records' lag. Additionally, we analyze the downtime caused by the scaling operation and how it is influenced by the size of the application state. We make observations regarding these results, that should be considered when designing an autoscaling policy to best meet a given SLA while minimizing overprovisioning.

2 Related work

Cloud computing is a relatively new field, but in the recent years it has gained a large interest among researchers.

The automatic scaling of distributed streaming applications consists of the following phases [13]: a monitoring system provides measurements about the current state of the system, these metrics are analyzed and processed, which is then applied to a policy to make a scaling decision (plan). Finally, the decision is executed, a mechanism performs the scaling operation. Most research is focused on the analytic and planning phase.

The authors of [13] have reviewed a large body of research regarding autoscaling techniques. They categorize the techniques into five categories: (1) threshold-based rules, (2) reinforcement learning, (3) queuing theory, (4) control theory, and (5) time series analysis based approaches.

The DS2 controller [11] uses a lightweight instrumentation to monitor streaming applications at the operator level, specifically the proportion of time each operator instance spends doing useful computations. It works online and in a reactive manner. It computes the optimal parallelism of each operator by a single traversal of the job graph. The authors have implemented instrumentation for Flink among other streaming systems. They have performed experiments on various queries of the Nexmark benchmarking suite to show that DS2 satisfies the SASO properties [1]: stability, accuracy, short settling time, and no overshoot. The job converges to the optimal parallelism in at most 3 steps (scalings). The resulting configuration exhibits no backpressure, and provisions the minimum necessary resources.

PASCAL [12] is a proactive autoscaling architecture for distributed streaming applications and datastores. It consists of a profiling and an autoscaling phase. In the profiling phase, a workload model and a performance model are built using machine learning techniques. These models are used at runtime by the autoscaler to predict the input rate and estimate future performance metrics, calculate a minimum configuration, and to trigger scaling if the current configuration is different from the calculated target. For streaming applications, these models are used to estimate the CPU usage of each operator instance for a predicted future workload. The authors show that their proactive scaling model can outperform reactive approaches and is able to successfully reduce overprovisioning for variable workloads. In our work, we use a different metric from the CPU load, based on how much the job lags behind the input. Our policy is a reactive approach, but it might be interesting to explore whether a proactive model could be built on these metrics.

Ghanbari et. al. [8] investigate cost-optimal autoscaling of applications that run in the cloud, on an IaaS (infrastructure as a service) platform. They propose an approach that uses a stochastic model predictive control (MPC) technique. They create a model of cloud and application dynamics. The authors define a cost function that incorporates both cloud usage costs, as well as the expected value of the cost or penalty associated with the deviation from certain service level objectives (SLOs). These SLOs are based on metrics that describe the overall performance of the application.

In our work, we aim to describe the characteristics of scaling Flink jobs, to serve as a base for an optimal scaling policy in the future. We also provide an architecture for making and executing the scaling decisions.

3 System architecture

We have implemented an autoscaling architecture on Kubernetes. This section gives an overview of the components involved in running, monitoring and scaling the applications.

3.1 Kubernetes operator

Flink applications can be executed in different ways. Flink offers per-job, session and application execution. Flink supports various deployment targets, such as standalone, Yarn, Mesos, Docker and Kubernetes based solutions. There are various managed or fully hosted solutions available by different vendors.

There are also multiple approaches of running Flink on Kubernetes. One method is to deploy a standalone cluster on top of Kubernetes. In this case, Kubernetes only provides the underlying resources, which the Flink application has no knowledge about. Flink also supports native Kubernetes deployments, where the Flink client knows about and interacts with the Kubernetes API server.

We have decided to use the standalone mode combined with Kubernetes' operator pattern [6] to manage Flink resources. The open-source operator [21] by Google defines Flink clusters as custom resources, allowing native management through the Kubernetes API and seamless integration with other resources and the metrics pipelines. The operator encodes Flink-specific knowledge and logic in its controller.

The desired state of the cluster is specified in a declarative manner, conforming to the format defined in the custom resource definition (CRD). The user submits this specification to the Kubernetes API server, which creates the *FlinkCluster* resource. The operator, installed as a deployment, starts to track the resource. The reconciliation loop performs four steps.

1. It observes the resource, and its sub-resources, such as *JobManager* or *TaskManager* deployments, ingresses, etc.

2. The controller uses the observed information to compute and update the *Status* fields of the resource through the API.

3. The desired state of the individual cluster components is calculated, based on the (potentially changed) observed specification, and the observed status.

4. Finally, the desired component specifications are applied through the API.

This loop runs every few seconds for every *FlinkCluster* resource in the Kubernetes cluster.

3.2 Scale subresource

We have modified the operator to expose the *scale* subresource on the *FlinkCluster* custom resource. This exposes an endpoint that returns the current status of the scaling, which corresponds to the number of *TaskManager* replicas and the job parallelism, as well as a selector, which can be used to identify the

Pods that belong to the given cluster. Additionally, this endpoint can be used to set the desired number of replicas in the *FlinkCluster* Spec.

The scaling process starts with this step, the desired replicas are set through the scale subresource. The scaling is done in multiple steps, with intermediate desired deployments in the process. The operator can keep track of the cluster's and the job's state, and perform the steps of a scaling operation. As the scale subresource's *replicas* specification changes, the operator first requests a savepoint and the deletion of the existing cluster. Once this is complete, it computes the desired deployment (step 3 of the reconciliation loop) with the newly desired replicas. When the cluster components are ready, it resubmits the job with the appropriate parallelism, starting from the latest savepoint.

3.3 Custom metrics pipeline

The scaling decisions are based on metrics from the Flink job. We have used Prometheus [2] to scrape the job's metrics. Flink has an established metrics system, including access to connector metrics (such as Kafka). We have used Flink's Prometheus reporter to expose the metrics to Prometheus.

To access Prometheus metrics through the Kubernetes metrics API, we have used an adapter [16]. It finds the desired time series metrics in Prometheus, connects them to the appropriate Kubernetes resources, and performs aggregations, exposing the results as queryable endpoints in the custom metrics API. The metrics we have decided to calculate will be described in detail in Section 4. Figure 1 shows the overview of the metrics pipeline.

3.4 Horizontal Pod Autoscaler

The Horizontal Pod Autoscaler (HPA) [20] is a built-in Kubernetes resource. It can control the scaling of Pods in a replication controller, such as ReplicaSet or Deployment. It can also scale custom resources whose Scale subresource is exposed and the scaling logic is implemented. Since we have done this for *FlinkCluster* resources, we can set them as scaling targets for the HPA.

The Horizontal Pod Autoscaler uses the currently observed replica count (either calculated by counting pods with certain labels, or read from the scale endpoint), as well as various types of metrics to calculate the desired number of replicas, by the following formula:

$$\mathrm{desiredReplicas} = \max_{i \in \mathrm{usedMetrics}} \left\lceil \mathrm{currentReplicas} \times \frac{\mathrm{currentMetricValue}_i}{\mathrm{desiredMetricValue}_i} \right\rceil \quad (1)$$

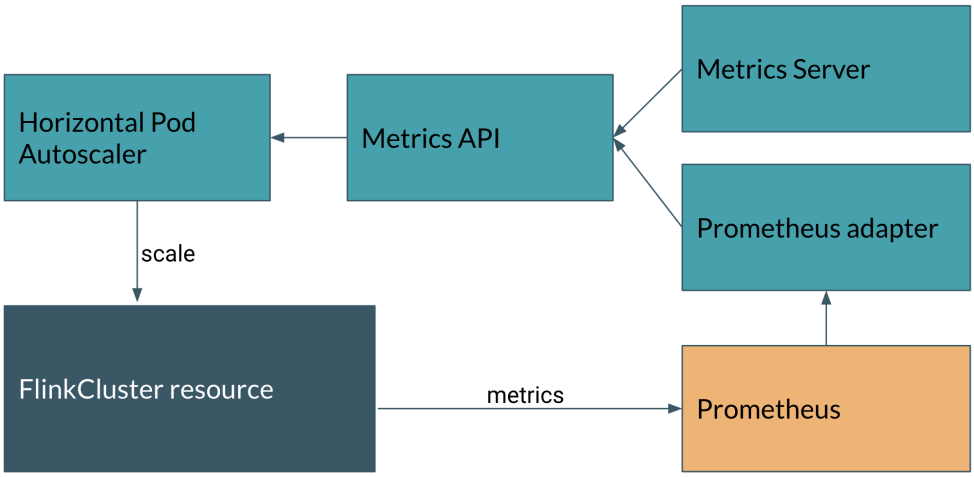

Figure 1: The custom metrics pipeline with Prometheus, used in our scaling architecture.

We have used the latest, *autoscaling/v2beta2* version of HPA, which has support for both resource, custom, and external metrics. We can specify multiple metrics, including those we have exposed through Prometheus and the custom metrics API. The HPA calculates the desired replica count based on each metric, and uses the largest value to scale the resource. In our setup, we have used two metrics through the custom metrics pipeline described above. The metrics will be described in Section 4.

The above equation assumes that the application can handle workload proportionally to the number of replicas, and that the metrics linearly change with the utilization, and therefore inversely with the number of replicas. This means that if the current metric is n times the desired metric, the desired replicas are calculated to be n times the current replicas. After the scaling, we expect the current metric value to decrease to near the desired metric value.

4 Scaling policy

We have implemented a scaling policy inspired by [15, 7]. We assume that the stateful Flink job reads a stream of data from a Kafka topic, performs some calculations on the records, and emits the results to another Kafka topic. Apache Kafka [14] is an open-source event streaming, message broker platform. Producers can write and consumers can read topics that are stored in a

distributed manner among Kafka brokers. The data in the topics is broken up to partitions.

The data is assumed to be evenly distributed, i.e. there is no skew among the number of messages in the partitions, so the parallel operator instances are under a nearly equal load.

The scaling policy uses two different metrics. One is based on the relative changes in the lag of the input records, and the other on the proportion of time that the tasks spend performing useful computations.

4.1 Relative lag changes

The goal of autoscaling is for the Flink job to be able to handle the incoming workload. In distributed systems, failures are inevitable and a part of normal operation, so the processing capacity of the job should slightly exceed the rate of incoming records. This way, in the case of a failure, the job can catch up after a recovery. However, the job should not be overprovisioned by a large factor, to avoid wasting resources.

Kafka keeps track of the committed offsets in each partition (number of records that have been read by Flink), as well as the latest offset (the number of records). The difference between these is called the consumer *lag*. Flink uses a separate mechanism from Kafka's offsets to keep track of how much it has consumed, due to the way it handles checkpoints and consistency, but it is still possible to extract information about how far the consumption lags behind.

Flink's operators are deployed to tasks as multiple parallel instances. Each task of the Kafka source operator is assigned a number of partitions of the input topic. A total lag or an average lag metric would be more useful, but it is not available with this setup. Therefore, we give an upper bound for the total lag using the *records_lag_max* metric, which returns for instance the maximum of the lags in the partitions that they read. For example, consider a topic with 1 million messages evenly balanced among 20 partitions, when read from the beginning by a Flink job with 4 consumer instances. Each instance would get assigned 5 partitions, and when reading from the beginning, the lag for each partition would be 50000, hence the *records_lag_max* would also be 50000 for each instance. As the job consumes the messages (faster than they are produced), the lag in each partition would decrease, and this metric would give the largest among them for each task instance. We give an overestimation for the total lag by multiplying this metric for each instance with the corresponding *assigned_partitions* metric, and summing this value for all tasks with the source operator's instances. Equation 2 summarizes this calculation.

$$\mathrm{totalLag} = \sum_{i \in \mathrm{SourceTasks}} \mathrm{records\_lag\_max}_i \times \mathrm{assigned\_partitions}_i \quad (2)$$

If this lag is nearly constant, the job is keeping up with the workload. If the lag is increasing, the job is underprovisioned and needs to scale up. If the lag is decreasing, the job may be overprovisioned. However, this may be a desired scenario if there was a large lag, as the latency will decrease if the job can process the lagging records.

To decide whether the current parallelism is enough to handle the workload, we take the rate of change of the lag metric (in records / second), using PromQL's (Prometheus' query language) [2] *deriv* function over a period of 1 minute.

The specific value of lag by itself is not too meaningful, and neither is its rate of change. To be useful, we compare it to the rate at which the job processes records (in records/second), which can be calculated by summing the *records_consumed_rate* metric for each operator instance of the Kafka source, which we call *totalRate*. To smooth the effect of temporary spikes, we use a 1-minute rolling average of this metric.

The ratio of these two metrics gives a dimensionless value, which represents how much the workload is increasing (positive) or decreasing (negative) relative to the current processing capabilities. After adding 1, the resulting number is the multiplier necessary for the number of replicas to keep up with the load.

$$\mathrm{relativeLagChangeRate} = 1 + \frac{\mathrm{deriv}(\mathrm{totalLag})}{\mathrm{totalRate}} \quad (3)$$

This number is only meaningful for scale up decisions, since a decreasing lag might still warrant the current number of replicas, until the application catches up to the latest records. By using *relativeLagChangeRate* as the *currentMetricValue*, and 1 as the *desiredMetricValue* for the Horizontal Pod Autoscaler, as described in Equation 1, the desired replicas are properly calculated when scaling up. For downscaling, multiplying the parallelism with this value would still correctly make the job match the incoming message rate, but a downscaling is likely not warranted until the lag is below a certain threshold. The utilization portion of the policy, described in the next subsection, is responsible for making downscaling decisions.

To make the job catch up to the lag after the scaling, and to account for future failures, it might be worthwhile to slightly overshoot when scaling up.

This could be achieved by setting the desired metric value to a slightly smaller number or including a multiplier in the fraction in Equation 3.

When the job can process messages at the rate they are generated, or if the job is overprovisioned with no lag, then the change of the lag is 0, giving the *relativeLagChangeRate* metric a value of 1. These two cases need to be distinguished, as in the latter case scaling down might be possible. Since the HPA sets the parallelism to the maximum dictated by various metrics, a value of 1 for *relativeLagChangeRate* in the case of 0 lag would prevent a downscaling based on other metrics (whose values would be between 0 and 1).

To avoid this, we need some logic not to use this metric when the total lag is below a desired threshold. To simulate this, we can use the following value instead in the query for the HPA:

$$\frac{\mathtt{totalLag} - \mathtt{threshold}}{\mathtt{abs}(\mathtt{totalLag} - \mathtt{threshold})} \times \mathtt{relativeLagChangeRate} \tag{4}$$

If the total lag is less than the threshold, then the fraction in the equation is -1, which makes the above expression negative. Then, this metric will no longer prevent the downscaling based on the other metric, since the HPA takes the maximum of the values calculated based on various metrics. If the total lag is larger than the threshold, then the fraction is 1, so this metric will be taken into account when making the scaling decision.

As noted before, this metric always overestimates, since the total lag is calculated based on the maximum lag of each subtask. If the data has a large skew, the effect of this overestimation may be very large. Therefore, it may be necessary to adjust the desired value of this metric based on knowledge about the actual data being processed. If the number of subtasks matches the number of Kafka partitions, and each subtask processes one partition, then the metric is based on the true lag value. Obviously, due to scaling, this will not be the general case, because the parallelism changes over the lifetime of a job. In the future, it would be desirable to expose a metric about the lag in each of the partitions.

4.2 Utilization

As noted in the previous subsection, another rule is necessary for the policy to distinguish between cases when the lag remains unchanged because the job's processing capabilities match the incoming rate, and when the job could process more than the incoming rate, but there are no records to be processed.

This rule is based on the *idleTimeMsPerSecond* metric. It is a task-level built-in metric in Flink, that represents the time in milliseconds that a task is idle. A task can be idle either because it has no data to process, or because it is backpressured (it is waiting for downstream operators to process their data).

If a task is idle due to a lack of incoming records, then the job is overprovisioned for the current load. We define the utilization metric for a job as the average portion of time its tasks spend in a non-idle state.

In our observations, the tasks executing the source operators (Kafka consumers) have shown an *idleTimeMsPerSecond* metric of 0 ms when there was no lag and the job was able to keep up with the messages, and 1000ms when the job was processing events at its maximum capacity, due to the source tasks being backpressured by the downstream processor operators. This is not an accurate representation of the overall utilization of the job, therefore we have excluded the tasks containing the first operator from the calculation. For this metric to be useful, the source operators should not be chained to others.

The metric can be expressed with the following formula:

$$\mathtt{utilization} = 1 - \frac{\mathtt{avg}_{\mathtt{nonSourceTasks}}(\mathtt{idleTimeMsPerSecond})}{1000} \tag{5}$$

This is a dimensionless number between 0 and 1, representing the average utilization of tasks in the job. We use this as the *currentMetricValue* in the Horizontal Pod Autoscaler, and set the desired value to a number less than 1.

If the utilization is above the desired value, it may trigger a scale up before the lag metric. Its main purpose however, is to scale down when the utilization is low. If this is the case, the lag metric is 0, so the job would be able to process more records than the incoming rate. For example, if the target utilization is 0.8, and its current value is 0.4, then a scale down should be triggered, the new parallelism should be half of the original.

This simple metric assumes that the job's tasks are under even load. A finer-grained approach could be used with more knowledge about a job's specifics, such as which operators perform heavy calculations, and consider tasks with different weights in the average.

5 The effects of scaling on performance

The scaling operation requires the snapshotting of the whole application state onto persistent storage. Additionally, in our Kubernetes operator implementation, the original job's pods are destroyed, and new pods are provisioned

instead with the appropriate replica count. The Flink job then has to be restarted from the recently snapshotted state.

In this section, we analyze how the size of the state affects the downtime during the scaling operation. We perform measurements regarding the savepoint duration, the overall downtime, and the loading time of the savepoint after the scaling.

5.1 Experimental setup

We have created a simple Flink job that reads records from a topic in Apache Kafka [14], keys them by one of the fields, performs a simple calculation in a *KeyedProcessFunction*, and writes the results to another topic. The job stores a random string of configurable size in a keyed state. The number of keys is also configurable.

A separate data generator job produces the records of the input topic. The creation timestamp is stored as a field of each record. The main job's last operator before the Kafka producer calculates the difference between its own processing time and the stored timestamp, and writes this elapsed time to the emitted record. This value can be used to determine the total downtime in Section 5.3. We disregard the differences among the nodes' clocks.

We have used the operator's savepoint mechanism, as well as the scale endpoint to trigger savepoints and restarts. We have observed the duration of savepointing on the JobManager dashboard. To calculate the downtime and latency distribution, we have observed the output Kafka topic's records with the elapsed time field.

5.2 Effects of state size on savepoint duration

Flink has an aligned checkpoint mechanism [3]. When triggering a snapshot, a snapshot barrier is inserted at each input operator, which flows along the streaming pipeline along with the records. An operator takes a snapshot of its state when it has received the snapshot barriers from all of its inputs. When operators are under different loads, this can significantly delay the snapshotting of certain operators' states.

The latest versions of Flink also support unaligned checkpoints [19] (but not unaligned savepoints), where checkpoint barriers can overtake buffered records, and thus avoid this delay caused by the alignment. This does not affect the time required for the I/O operations involving the checkpoint. In future

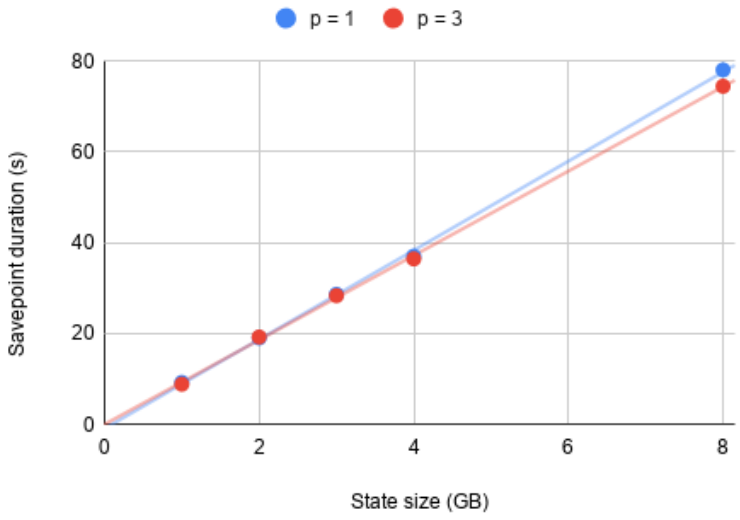

Figure 2: Duration of taking a savepoint as measured by Flink, with relation to state size, when the job runs with a parallelism of 1 or 3.

research, it might be worthwhile to investigate how using aligned or unaligned checkpoints instead of savepoints might affect performance and scaling time.

Each operator must serialize its state and write it to a persistent storage. We have measured how the size of application state affects this portion of the snapshotting process. To avoid the fluctuation caused by the barrier mechanism, we have stopped the data generator, and thus the stream of incoming records, before taking snapshots. At this point, the keyed application state had the sizes of {1.0, 2.0, 3.0, 4.0, 8.0} GB. The application has run with a parallelism of 1 and 3. We have used Amazon's Elastic File System (EFS) to store the snapshots. This is one of the storage options that can be used for Kubernetes, but there are other implementations and providers that may offer different performance characteristics. In the future, it might be worthwhile to compare the offerings of various providers.

We have manually triggered savepoints 5 times for each combination of parallelism and state size. The measurements, read from the Flink dashboard, were rounded to seconds, and an average value was taken. As seen on Figure 2, state size has a strong linear correlation to the serialization and writing time of the snapshotting phase.

The parallelism of the job did not affect the savepoint duration. It is limited by the write rate of the file system. EFS offers 100 MiB/s (105 MB/s) bursting performance. In our measurements, we have confirmed that savepoints were written at this rate.

This has been measured when the application is in a healthy state and not backpressured. The job did not need so spend time synchronizing the checkpoint barriers, as the pipeline was simple, with no multi-input operators. Backpressure does significantly increase the time to the completion of a savepoint, since the snapshot barriers must flow through all operators of the job. The beginning of writing an operator's state may thus be delayed until the barrier reaches it. The magnitude of this effect depends highly on the specific job.

The snapshotting of an operator's state happens asynchronously, with a small impact on the overall performance of the pipeline. While this effect itself may cause delays in processing and slow the pipeline, it is unlikely to cause problems in real-world scenarios due to its small magnitude, so we have decided not to focus on this effect.

5.3 Effects of state size on scaling downtime

The majority of the downtime during a scaling operation is due to the loading of state from the persistent storage. The properly partitioned state must be distributed to the operator instances from the persistent storage.

We have measured how the total downtime of the processing job is affected by the size of the state. We have manually triggered trigger scale-up and scale-down operations with different state sizes. To measure the downtime, we have used the method described in Subsection 5.1.

With state sizes of approximately 1, 2, 4 and 8 GB, we have found that the creation of the infrastructure done by Kubernetes, the deletion and initialization of pods is a factor with a large variance, that is responsible for the larger portion of the scaling time. Additionally, if the Flink image is not available on the node, or the image pull policy is set to *Always*, its pulling is another factor that we have little control over, and might dominate the scaling time.

In an extreme case, the cluster has no available resources to provision the requested number of pods, and the job is in a stopped state until resources become available, or a new node is initialized by some autoscaling mechanism. To make measurements feasible, we limit the experiments to cases when enough resources are available on the cluster, and new pods can be provisioned without delay.

However, even in this limited case, our observations have shown a large variance in the scaling times due to the above described factor of waiting for Kubernetes. Figure 3 shows the downtimes we have measured for target parallelisms of 2 to 8, with state sizes between 2 and 8 GB. Based on the observed data, we were not able to establish the effect of either the state size or the target parallelism on the scaling time.

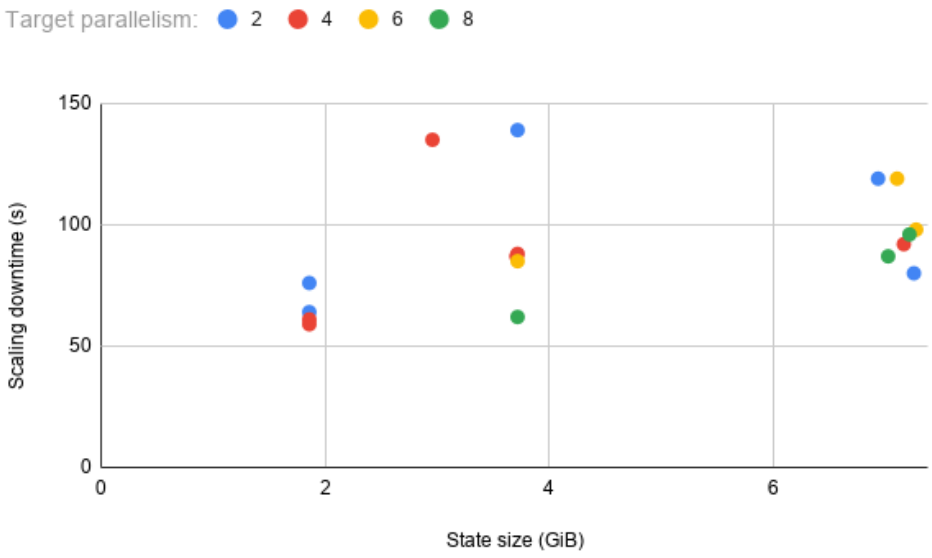

Figure 3: Total time for the scaling operation, during which the job processes no messages.

5.4 Effects of state size on savepoint restoration duration

To directly observe the effects of state size, while eliminating the large variance described in the previous section, we have taken a different approach. We have added a measurement directly to the *OperatorChain* class' *initializeStateAndOpenOperators* method. This exposes the duration of initializing each operator's state. This can either be a newly created state, or one restored from a savepoint. In our setup, only one operator had stored state, whose size we have had control over. We have exposed the load time as a gauge type metric through Prometheus, and filtered it by the operator name in a query.

We have performed measurements with state sizes up to 8 GB, and target parallelisms of 2, 4, 6, and 8. In each measurement, we have recorded the state initialization time of each stateful parallel operator instance. The operator instances work in a parallel manner, so the job is able to start processing at the full rate after the last subtask has loaded its state. We have measured the averages and the maximums of the subtask load times.

We have hypothesized the correlation between state size and load times to be linear. This was confirmed by performing a linear regression on the data points of the maximum and average values, while disregarding the target parallelism component. The R^2 values of 0.905 and 0.833 respectively indicate that state size explains a large portion of the load times' variability. Figure 4 shows the data points and the trendlines for linear regression.

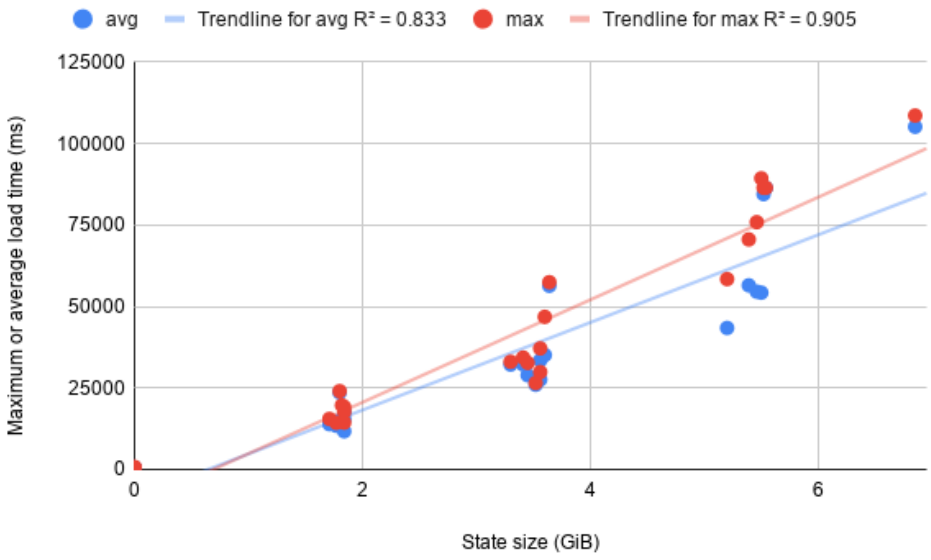

Figure 4: Maximum and average state initialization times of all instances of the stateful operator in job restarts with various state sizes.

5.5 Effects of target parallelism

Each parallel operator instance loads only a portion of the state, partitioned by some key. With a higher parallelism, the state that each operator must load is smaller, which may result in a quicker load time. However, there may be

other bottlenecks (e.g. disk or network performance), that may limit the rate of loading the state.

Flink uses key groups as the smallest unit of distributing keyed state. Each key group gets assigned to a parallel instance of an operator. The number of key groups is equal to the maximum parallelism, which can only be set when the job is first started. If key groups are not distributed evenly, the state size is uneven between operator instances. It is recommended [17] to use parallelisms that are divisors of the maximum parallelism setting to avoid this issue. In our measurements, we have used a maximum parallelism setting of 720, which is divisible by most of the parallelism settings we have measured with (except for 32).

We have performed measurements to determine what the effect of the target parallelism is on the load time. We have used a cluster with 6 AWS *m5.2xlarge* instances, and ran the benchmark job with approximately 10 GiB state. We have initiated the scaling operation with target parallelisms of 4, 6, 8, 12, 16, 24, and 32, scaling from various parallelisms. The job was initially started with a parallelism of 8.

We have measured the state load times of each subtask using the same tooling as before. We have scaled the job to each parallelism 6 times, and taken the averages of the maximum and the average load times of each measurement.

Based on the results of the measurements, seen in Figure 5, we were not able to observe a clear correlation between the target parallelism and the loading time of the state.

6 Discussion and future work

We have worked on the problem of autoscaling Flink applications to adapt to the current workload. We have built and described a scaling architecture based on Kubernetes and its operator pattern.

We have focused on Flink jobs with inputs from Kafka, and detailed a simple scaling policy based on relative changes to the Kafka consumer lag, as well as the idle rate of the tasks in the job. The policy is implemented on a Horizontal Pod Autoscaler with custom metrics. We have described this scaling architecture in detail. It can be used to implement different policies with minimal modifications. For more complex setups, the Horizontal Pod Autoscaler may be replaced with a custom controller that implements an advanced scaling algorithm.

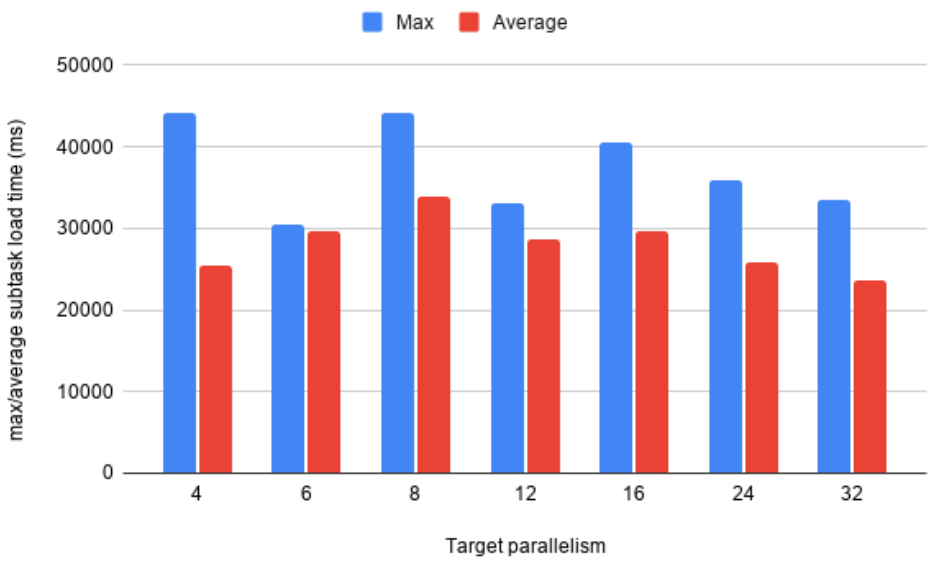

Figure 5: The average of the 6 measurements for maximum and average load times.

We have investigated the factors that affect the downtime caused by the scaling operation. Due to the steps of the scaling, the processing latencies of records may greatly increase during and after the scaling, as there is some downtime when there are no records being processed.

We have found that there is a linear correlation between the state size and the duration of savepointing to a persistent storage, excluding the time spent aligning the checkpoint barriers. While this is done asynchronously, it causes a delay between when the scaling is triggered and the beginning of the downtime.

Our measurements showed that there is a large variance in the total downtime caused by the scaling operation. We were not able to establish the effect of state size or target parallelism at this point. The time it takes for Kubernetes to reconcile the number of necessary pods, pull the images and start the Flink processes takes up a large portion of the total duration.

This means that with this setup there is a lower bound to the maximum latencies and the downtime. Therefore it is not possible to give guarantees that every record will be processed with a low latency, the SLA has to allow for occasional downtimes on the scale of a few minutes, if the application uses autoscaling.

In our measurements, we have tried to break down the factors that influence the job's restarting time. When restarting from a savepoint, portions of the state are loaded by the subtasks from the persistent storage. The duration of this loading is one of the factors that affect the total downtime. We have found that the state size is linearly correlated with this duration. We have not found a correlation between the parallelism of the restarted job and the maximum or the average load time of its subtasks.

Our described policy may serve as a basis for designing a more advanced autoscaling setup. When the policy must respect an SLA, the scaling downtime is a factor to take into consideration. A longer restart time means that the job will accrue a larger lag during the scaling operation. Thus, it might violate the SLA for a longer period. It might make sense to overprovision by a larger factor to account for this and make the job catch up quicker.

In future work, it may be worthwhile to investigate how to incorporate our results about the effects of state size to the policy. If we can calculate the scaling downtime based on state size, it can be approximated how much additional lag the restart will cause, and how much the total lag will be. Based on this, we may calculate an appropriate scaling factor that allows the job to catch up (to decrease the lag below a threshold) within the time allowed by the SLA.

With a good autoscaling architecture and policy, long-running Flink jobs will be able to keep up with changing workloads while utilizing resources effectively. Our contributions serve as a basis towards building such an architecture and designing an optimal policy.

Acknowledgements

The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

References

[1] T. Abdelzaher, Y. Diao, J.L. Hellerstein, C. Lu, X, Zhu, Introduction to control theory and its application to computing systems, in: *Performance Modeling and Engineering*, Springer US, Boston, MA, 2008, 185–215. ⇒41

[2] B. Brazil, Prometheus: Up & Running : Infrastructure and Application Performance Monitoring, *O'Reilly Media, Inc.*, 2018 ⇒44, 47

[3] P. Carbone, G. Fóra, S. Ewen, S. Haridi, K. Tzoumas, Lightweight asynchronous snapshots for distributed dataflows, *arXiv preprint 1506.08603*, 2015 ⇒40, 50

[4] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, K. Tzoumas, State management in Apache Flink: Consistent stateful distributed stream processing, in *Proc. VLDB Endow.* **10,** 12 (2017) 1718–1729 ⇒40

[5] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink: Stream and batch processing in a single engine, *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* **38,** 4 (2015) 28–38 ⇒39

[6] J. Dobies, J. Wood, Kubernetes Operators: Automating the Container Orchestration Platform, *O'Reilly Media, Inc.*, 2020 ⇒43

[7] Fabian Paul, Autoscaling Apache Flink with Ververica Platform Autopilot, *Ververica Blog*, 2021, https://www.ververica.com/blog/autoscaling-apache-flink-with-ververica-platform-autopilot ⇒45

[8] H. Ghanbari, B. Simmons, M. Litoiu, C. Barna, G. Iszlai, Optimal autoscaling in a IaaS cloud, in: *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, Association for Computing Machinery, New York, NY, USA, 2012, 173–178 ⇒42

[9] K. Hightower, B. Burns, J, Beda, Kubernetes: Up and Running Dive into the Future of Infrastructure, *O'Reilly Media, Inc.*, 1st edn., 2017 ⇒41

[10] F. Hueske, V. Kalavri, Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications. *O'Reilly Media, Inc.*, 2019 ⇒39

[11] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, T. Roscoe, Three steps is all you need, Fast, accurate, automatic scaling decisions for distributed streaming dataflows, in: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, USENIX Association, USA, 2018, 783–798 ⇒41

[12] F. Lombardi, A. Muti, L. Aniello, R. Baldoni, S. Bonomi, L. Querzoni, Pascal: An architecture for proactive auto-scaling of distributed services. *Future Generation Computer Systems* **98** (2019), 342–361 ⇒42

[13] T. Lorido-Botrán, J. Miguel-Alonso, J. Lozano, A review of auto-scaling techniques for elastic applications in cloud environments, *Journal of Grid Computing* **12** (2014) 559–592 ⇒41

[14] N. Narkhede, G. Shapira, T. Palino, Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale. *O'Reilly Media, Inc.*, 2017 ⇒45, 50

[15] F. Paul, Towards a Flink Autopilot in Ververica platform, *Flink Forward Global (2020)*, https://www.flink-forward.org/global-2020/conference-program#towards-a-flink-autopilot-in-ververica-platform ⇒45

[16] S. Ross, Prometheus Adapter for Kubernetes Metrics APIs (10.04.2021), https://github.com/DirectXMan12/k8s-prometheus-adapter ⇒44

[17] ∗∗∗ Setting max parallelism, *Cloudera Docs / Streaming Analytics 1.2.0* (10.04.2021), https://docs.cloudera.com/csa/1.2.0/configuration/topics/csa-max-parallelism.html ⇒55

[18] ∗∗∗ Apache Flink — Stateful Computations over Data Streams (10.04.2021), https://flink.apache.org/ ⇒39

[19] ∗∗∗ Stateful Stream Processing, *Apache Flink Documentation* (10.04.2021), https://ci.apache.org/projects/flink/flink-docs-release-1.12/concepts/stateful-stream-processing.html ⇒50

[20] ∗∗∗ Horizontal Pod Autoscaler, *Kubernetes Documentation* (10.04.2021), https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ ⇒41, 44

[21] ∗∗∗ Kubernetes Operator for Apache Flink (10.04.2021), https://github.com/GoogleCloudPlatform/flink-on-k8s-operator ⇒41, 43

Received: March 22, 2021 • Revised: April 11, 2021