SEP-32: Elasticity for Samza

Status

Current state: [ACCEPTED]

Discussion thread: https://lists.apache.org/thread/vjtl5fnf64kpkoxc591466y92dlt2bsb

Voting thread: https://lists.apache.org/thread/2v4gxtc6yoy77lkn2xg4n0dgj3t2mpgq

JIRA: SAMZA-2687

Released:

Problem

Some Samza jobs hit throughput issues which cannot be alleviated by sizing the job. One such issue often faced is the inability to scale up task count (and hence container count) beyond the number of input partitions. The need for this arises due to the process-time of the job's logic which is not under Samza' s control. Then the only way to address the job's low throughput is to manually increase the partition count of the input streams achieving a higher number of tasks. This design doc aims to solve the problem of throughput dependency on input partition count by introducing virtual tasks which are similar to tasks but consume a portion of the input partition.

Motivation

Throughput via parallelism is tied to the number of tasks which is equal to the partition count of input streams. If a job is facing lag and is already at the max container count = number of tasks = number of input partitions, then the only choice it is left with is to repartition the input. As most input systems (Kafka and Brooklin) need manual intervention to increase the number of stream partitions, a job can not be sized to alleviate lag as the scale of parallelism is limited to task count. Furthermore, input repartitioning does not even work for stateful jobs.

Impact of solving this problem

1. Job's scale is no longer limited by input partition count.
    a. Large process times arising from complex and long job DAG can be supported without causing lag.
    b. Improve stateful job throughput which SEP-5: Enable partition expansion of input streams does not solve. SEP-5 allows the repartitioning of the stateful job's input without losing all the state. It does not allow for more tasks within the job but rather allows the same number of tasks to consume the extra number of input partitions.
2. The streaming storage (aka input streams) is no longer forced to use the partitioning required to support the required processing parallelism.

Scope

This SEP focusses on designing elasticity for stateless job only. Elasticity for stateful jobs will be a future work and there will be a follow up SEP for it.

Requirements

The proposed solution should meet the following requirements.

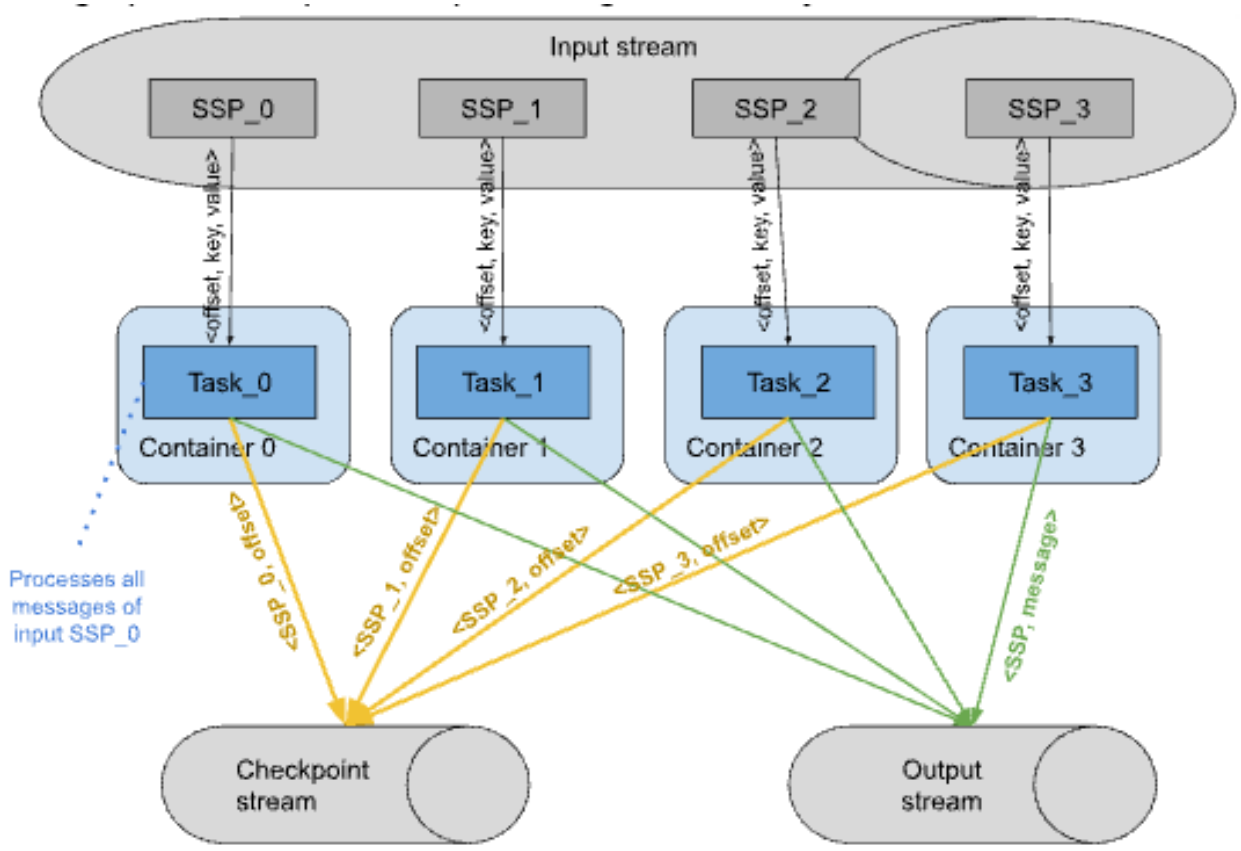
1. Support scale up and scale down of the job to increase throughput without limiting the scale up to the input streams' partition counts.
    a. Overcome the need for input stream repartitioning when the job faces lag.
2. Samza guarantees should be upheld:
    a. No data loss while scaling up and scaling down the job.
    b. In-order processing of messages with the same key.
3. The solution should be agnostic to input systems, output systems, intermediate/metadata system and the cluster-manager as well.
4. It should provide interfaces/api/configs for autosizing controller to size the job.
5. The solution should work for all stateless Samza jobs.
6. Design should be extensible for stateful jobs should be able to increase the number of (virtual) tasks beyond the number of input partitions while preserving state.

Assumptions:

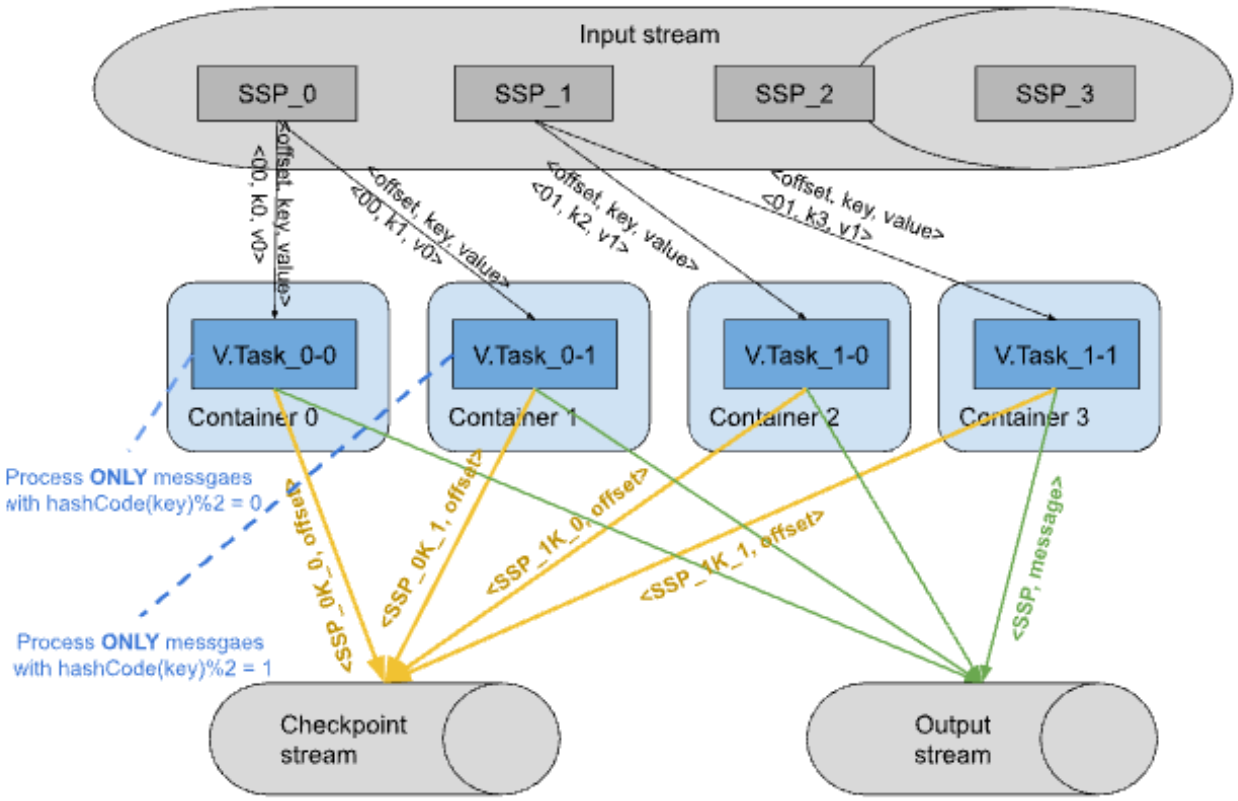
1. Future stateful job elasticity work will need the assumption that all state of a job is keyed by input key or by a state key obtained by applying a 1:1 function to input key (to figure out for a given state key which input key it belongs to).
2. Stateful job support in future work will be limited to jobs using beam api and blob store based state backup and restore.

Background

Each task processes a whole SSP. The maximum number of tasks and hence containers that can be used by the job is equal to the number of input partitions. For stateful jobs, it is further limited by the number of tasks the job originally started with. SEP-5 allows input repartitioning to multiple of original partition count but scaling up of task count can not happen without wiping out all of the existing state.



Existing consumption, processing and checkpointing in a Samza Job.
Max # of tasks = # of input partitions (SSPs)



Proposed consumption, processing and checkpointing in a Samza Job.
Max # of tasks = pow2 X # of input partitions (SSPs)

Proposed Changes

The main idea behind this solution is to allow consumption of a part of an SSP by a logical processing unit akin to a task. Instead of one task consuming all the messages within an SSP, this new unit will consume one portion of the messages within an SSP while the other portion(s) will be consumed by other units. A portion of the SSP will be identified as SSPKb = SSP-keyBucket and the new unit will be called a Virtual task. Since each virtual task consumes one portion of the SSP, if we want to scale up the task count by a factor of X, then we should have X portions per SSP. This X will be referred to as the *Elas ticity Factor* aka desired-number-of-SSP-portions aka virtual-task-count : task-count ratio.

Each message within an SSP is a key-value pair where value can consist of multiple fields. Message also has an offset which is its unique and deterministic identifier within the SSP. To define a portion within an SSP, we introduce the concept of KeyBucket which is the hashCode-of-key % desired-number-of-SSP-portions (% is modulo). If the key is null, then the offset will be used. Key here is the k in the k-v pair of the message and not a field within the value part of the message. Note that hashCode-of-key is a deterministic integer that is computed using the content of the key. Thus SSPKb aka *System StreamPartition-KeyBucket* is all the messages within an SSP whose *hashCode-of-key % desired-number-of-SSP-portions* is the same and there will be desired-number-of-SSP-portions number of SSPKb per SSP. For each input stream of the job, we use the same desired-number-of-SSP-portions. The hash function used for computing hashCode of key is not the same as the hash function in the producer of the input messages.

In order processing with a key is guaranteed because all messages with the same key end up in the same SSPKb and are processed by the same virtual task. Thus they are processed in the order they arrive.

During the lifetime of a job, there will be a need to split (scale up) or merge (scale down) virtual tasks to deal with changes in elasticity factor going up or down. The impact of split/merge will be discussed alongside each sub topic below. Currently splitting a task or v.task to twice the current count (for perf reasons - using power of 2 gives a fast mod computation) is supported.

JobModel

1. Current JobModel
    a. JobModel consists of <job-config, ContainerModels>
    b. ContainerModel consists of <container-id, Map<taskName, TaskModels>>
    c. TaskModel consists of <taskName, Set<SSP>, changelogPartition, TaskMode>

1. New JobModel
    a. TaskModel will get updated to *<taskName, Set<SSPKb>, changelogParition, TaskMode=virtual>*

2. Split/Merge virtual tasks
    a. Split: If a task reading an SSP needs to be split into two v.tasks then the new JobModel will contain two TaskModels in the place of one old TaskModel where the only difference between the two new TaskModels is the keyBucket= 0 or keyBucket=1. Splitting one v task into multiple v.tasks will be similar.
    b. Merge: When four v.tasks consuming from an SSP are merged into two, the number of SSPKb per SSP is reduced to two such that the number of keyBuckets will be 2 instead of 4.
3. SystemStreamPartitionGrouper to group input SSPs into tasks will handle elasticity as follows.
    a. GroupByPartition: Without elasticity, this grouper creates a task with name "Partition i" and this task's SSP set consists of partition i of each input stream. (SSP set of a task is the set of input partitions the task processes). Thus if there were 3 input streams with 2 partitions each then the number of tasks is 2 with this grouper. With elasticity enabled, this grouper gets the elasticityFactor X from config. It now creates X virtual tasks per original task "Partition i-j" with 0<= j < X and the SSP set of the virtual task consists of SSPKb-i-j which is the KeyBucket j of each input stream's partition i.
    b. GroupBySystemStreamPartition: Without elasticity, this grouper creates a task for each input SSP with task name being "SystemStreamPartition <system-name, stream-name, partition-number>" and the SSP set of the task is the single SSP. Thus if there were 3 input streams with 2 partitions each then the number of tasks is 6 with this grouper. With elasticity enabled, this grouper fetches the elasticityFactor X from the config. It now creates X virtual tasks per original task with name "SystemStreamPartition <system-name, stream-name, partition-number>-j" with the SSP set of virtual task being SSPKb-j which is keyBucket j of the input SSP.
    c. AllSspToSingleTaskGrouper: Without elasticity, this grouper creates num tasks = num containers and assigns every single input SSP to every task. This grouper is not supported with elasticity as it breaks the num tasks = num containers requirement.
    d. Note that for the SSP groupers, the split or merge is not that important as they take elasticityFactor and create virtual tasks.
4. TaskNameGroupers which group the tasks into containers will not need modification to handle elasticity for a simpler design. An optimization is proposed later to group virtual tasks consuming from same SSP into same container for lower consumption costs.

Consumption & processing

Consuming from the input streams will continue to be the same and operate with <key, value> pairs. Suppose a Task_0 that was consuming from SSP_0 is split into two virtual tasks. Then, the runloop for each virtual task will receive all the messages from the partition SSP_0 but the runloop will pass to VirtualTask_0-0 only the messages with hash(key) modulo 2 = 0 aka messages of SSP_0_0 (keyBucket = 0) while the VirtualTask_0-1 will process only the messages whose hash(key) modulo 2 = 1.

Splitting and Merging of virtual tasks is handled by the changes in JobModel changes when elasticityFactor changes. This means when the job model changes the task model then the SPPKb buckets per task changes. Based on this change, the grouper creates a virtual task with SSP set containing only the key buckets it should process. This gives the information to the runloop layer about which messages to pass onto the virtual task. No special changes are needed in this phase of the job.

Improved throughput without container increases

Splitting of a single task into virtual tasks will improve performance even if the container count is not increased as the container can now leverage multiple threads for the multiple virtual tasks thus increasing parallelism while maintaining in-order processing of messgaes with same key which is not possible with task.max.concurrency. This is not possible without splitting the original task. Additionally, since both virtual tasks are within the same container, there is no duplication consumption of the input (by definition of virtual tasks no duplicate processing) thus avoiding overwhelming the underlying input system. For example, virtual tasks within the same container with thread pool will avoid hitting kafka quota while increasing the throughput per partition.

Intermediate streams

Partition by operator creates an intermediate stream. The need to create this stream is to have the input messages partitioned by a key other than the input key. Hence after landing in the partition by stream, the input message and the message within this stream are not correlated. Suppose, input key was k_0 and partition by message's key was pk_101 then even prior to elasticity, these two keys could potentially end up in two different tasks. Hence if hash (k_0) modulo elasticityFactor = i and hash(pk_101) modulo elasticityFactor = j (i not equal to j) then they would end up in different virtual tasks without loss of correctness. In simpler words, intermediate streams are similar to input streams in that they also require the same guarantee of messages with same key arriving in order. So there is no special handling needed.

Input stream repartition

If the input stream is repartitioned, then the job will continue with previously configured elasticity. As in, suppose the input stream is repartitioned into twice the number of partitions and prior to the repartition, elasticity resulted in 2X number of virtual tasks. Then post repartition, there will be 4X virtual tasks (4X non-virtual task count) in the job. However, the throughput gain made by elasticity prior to the input repartition will now be automatically achieved by the repartition and hence elasticity can be scaled down - aka merge the virtual tasks appropriately.

Checkpointing

The entire flow for checkpointing will remain the same. Currently, there is one checkpoint message per task with the checkpoint itself containing *Map<SSP, offset>*. New checkpoint will contain *Map<SSPKb, offset>* and will be one per virtual task. Offset here corresponds to the original deterministic unique identifier of a message within an SSP and does not change with keyBucket concept introduction. These new checkpoints will be written to the checkpoint stream with information about SSPKb.

When two virtual tasks consuming two SSPKb of the same SSP are in a single container, their checkpoints will give different offsets for the SystemConsumers (aka consuming layer) to start consumption for a given SSP. In this case, the consuming layer takes the lower/older of the two offsets to start consuming from. The virtual task which had a checkpoint with a larger/newer offset in the SSP will filter out or ignore the messages w/ lower offset and start consuming when the offset matches its checkpoint.

Splitting and Merging of v.tasks

1. Splitting virtual tasks to twice the old virtual task count: since new task count is twice old count, each old (virtual) task will be split into two. Hence the old task's checkpoint will be used by the two new virtual tasks. This is handled by changes to the CheckpointManager within samza. Virtual task names will be decorated with elasticityFactor such as V.Task_0-1-2 indicating the original task name was 0, and this V.Task consumes keyBucket 1 when elasticityFactor is 2. Now, when the factor goes from 2 to 4, new virtual tasks are V.Task V.Task_0-0-4, V.Task_0-1-4, V.

Task_0-2-4 and V.Task_0-3-4. Of these V.Task_0-1-4 and V.Task_0-3-4 will inherit V.Task_0-1-2's checkpoint. This is because of the keyBucket is computed as hash(key) modulo elasticityFactor where hash(key) is the java hashcode of the key object. Since the key does not change for an IncomingMessageEnvelope, the key.hashCode() remains the same across computations - aka it is a deterministic integer irrespective of elasticityFactor. Now consider the integers that were falling into bucket 1 (out of bucket 0 and 1 with factor 2) when hashCode(key) modulo elasticityFactor is computed. When the elasticity factor goes from 2 to 4, these integers will fall into either bucket 1 or 3 (out of buckets 0,1,2 and 3 with factor 4) due to the nature of modulo and that 4 is a multiple of 2. Taking a simplistic example of hash codes being 0 through 8, we immediately see that all the odd codes that were going to bucket 1 with factor 2 will now go to either bucket 1 or 3 with factor 4.

2. Merging of two v.tasks into one: will result in reading checkpoints for the two v.tasks (of the same SSP) being merged and picking the min offset of the checkpointed offsets. This min of the offsets will become the starting offset for the new merged task. This is acceptable due to Samza's at least once guarantee. Note that the checkpoint of new merged VTask_0-1-2 will be computed from earlier tasks V.Task_0-1-4 and V.Task_0-3-4 which works similar to the splitting logic for key buckets.

Optimizations

1. Grouper changes to optimize the virtual-task assignment to containers such that virtual-tasks consuming from the same input partition are assigned to the same container so that the input consumption costs are lower and minimize consumption of same partition by several containers.
2. Minimize the deserialization costs by deserializing only those that are meant to be processed by the container. Minimizing at consumption layer aka not even consuming the messages of the partition that will not be processed by the container is not possible because we do not know before hand which messages are to be processed by a container and in most input systems, subscription is at the input partition level and not sub-partition level.

Limitations of current design

Consider the scenario where two virtual tasks consuming the same SSP are in a container. If one of the virtual tasks is much faster than the other virtual task then in this design, the overall throughput will be limited to the slower virtual task throughput. This is because they share the same SystemConsumer in the container which subscribes to the input SSP at a partition level and hence needs to consume at the rate of the slower virtual task. However, since both virtual tasks are in the same container, there is no increased consumption cost on the underlying input system. Throughput can be increased by moving the faster virtual task into a different container but this would incur a higher consumption cost on the input system. This is a known tradeoff in this design. This can be addressed in the future via better grouper mechanism or container assignment.

Rejected Solutions

All-to-all partitioning scheme

1. Idea: Task split with all virtual tasks consuming all partitions within an input stream. Note that this is different from the proposed solution in terms of how many SSPs a virtual task consumes. For eg If the input stream had partitions 0,1,2,4 originally with 2 containers. Tasks a,b in container 1 read 0,1 respectively and tasks c,d in container 2 read 2,4. So container 1 reads 0,1 and container 2 reads 2,4. With this rejected solution - there will be 8 v.tasks in say 2 containers but each v.task needs to read all 4 partitions - aka v.task a1 will read 0,1,2,4 and so on. This leads to both container 1 and 2 reading all 0,1,2,4 partitions.
2. Pros:
    a. Solves the hot partition problem.
3. Cons:
    a. Consumption will overwhelm the input system.
    b. Overhead of determining which input messages to process and which to ignore.

Per-key parallelism

1. Idea: Improve parallelism such that one key gets processed by one unit (split task or thread within a task) within the container.
2. Pros:
    a. Maintains ordering within the key. Proposed solution also guarantees this by virtue of using a key to determine the processing virtual task.
3. Cons:
    a. Number of keys within an input stream is not known apriori making it impossible to configure the number of threads or virtual tasks at the start of the job.
    b. Additionally, usually number of keys in an input stream is much much larger than typical number of tasks which could lead to a lot of metadata overhead.

KeyOrderedExecutor

1. Idea: Improve parallelism such that a container has a fixed thread pool and each thread consumes a set of keys leading to in order execution within keys.
2. Pros:
    a. Similar to elasticity design above and simpler design.
3. Cons:
    a. Elasticity design above is integrated into the engine framework and has the advantage of checkpointing per key bucket.
    b. Additionally, elasticity design allows for higher throughput than KeyOrderedExecutor if the virtual tasks consuming a single SSP are spread across multiple containers.

Public Interfaces

The following changes to public interfaces will arise out of this feature.

1. SystemStreamPartition in JobModel and Checkpoint will contain a fourth component KeyBucket. SSP class name will be retained. Thus the entries for SSP in JobModel and Checkpoint will look like "SystemStreamPartition: {System: A, Stream: B, Partition:0, KeyBucket:1}".
2. Similarly, task name in the JobModel and Checkpoint will also change to contain keyBucket info to identify the task. Thus a task created by GroupByPartition grouper which used to be named "Partition_0" will now be named "Partition_0-1-2" (0 is the partition number, 1 is the keyBucket and 2 is the elasticity factor).
3. These changes to JobModel and checkpoint will invariably also change the messages in the coordinator stream and checkpoint stream. While the message in the stream will retain its structure in entirety, only the SSP and task name components of the message will change as shown above.
4. New metrics will be introduced as part of the feature. Existing metrics will have virtual task name instead of the original task name.

Implementation and Test Plan

Implementation plan

1. SystemStreamPartition and Jobmodel to introduce KeyHash concept
2. Introduce virtual tasks in JobModel
3. Runloop and SystemConsumer updates to have virtual tasks process SSPKb
4. CheckpointManager changes - introduce virtual tasks into checkpoint
5. CheckpointManager changes - compute and update last processed offset per SSPKb
6. Above tasks for virtual task count decrease
7. Metrics - add new metrics for virtual tasks

Functional testing

1. Scale up and scale down without increasing kafka input partition count
2. Stateless jobs correctness -
    a. with no duplicate processing and respecting SSPKb
    b. Checkpoint migration with scale up and down with at least once guarantee upheld
3. Input stream repartition after virtual task scale up
4. Intermediate topic correctness

Performance testing

1. Process-ns due to computation of key hash per incoming message
2. Job startup time and commit-ns due to checkpoint migration
3. Memory footprint across all containers due to system consumer of multiple containers polling and consuming same partitions
4. Disk usage & latency - due to column families adoption for single store split into two sub-stores each of which has the same entries.

Monitoring and Operations

Metrics : Since a virtual task is the same as a task for all practical purposes - has a task model, similar task name, has task Instance, processes in the same manner, the container and task level metrics need not be redefined. All the existing container and task level metrics will work as is with an increase in the number of metrics as task count increased. Our current curation of these metrics to just consolidate all tasks' metrics will work as is.

1. **Job-model-generation-ns**: new metric to track how long it takes to generate the job model as compared to earlier. This duration involves computing key hashes for all SSPs, larger number of task models and container models.
2. **Keyhash-compute-ns**: new metric to track how long it takes to compute the key hash for each incoming message envelope (to be used to pass the envelope to the correct virtual task)
3. **checkpoint-compute-n**s: new metric to track how long it takes to figure out the checkpoint when task-scale-up or task-scale-down occurs.
4. **store-identify-ns**: new metric to track how much time is spent in identifying the sub-store the v.task should write. This will track the time during both phases - prior to v.task scale up and post v.task scale up.
5. **task-count:** number of tasks (virtual tasks) within the job. This will help us track and correlate over time how the lag and virtual task scale up and down are proceeding.
6. **total-input-consumption-ns:** new metric to track how long it takes for all containers to consume the input streams as multiple containers will be reading from the same SSP
7. Existing metrics to track: process-ns, commit-ns, disk-usage, get-ns/put-ns/delete-ns

Operations

1. Config to set elasticity scale to be 2X (or 4X..) resulting in 2X number of v.tasks compared to original non-v.task count.
2. Config to turn off elasticity and roll back task scale up.

Compatibility

Rollout of the feature is via config. Checkpoint and job model compatibility should be ensured during implementation.

Job model changes for the feature are basically adding a new field (keyBucket) to SSP in the task model. The SamzaObjectMapper which serializes and deserializes job model components including SSP serde will first be updated to prepare for SSP with KeyBucket as a preparatory step even before KeyBucket is introduced into the SSP object. This prep is mainly for the deserialization part of the object mapper making it able to read a job model which has keyBucket. This is to ensure forward compatibility in the sense that if a job that ran with (elasticity aka SSP with KeyBucket in its job model) rolls back to older samza version without keyBucket, it should be able to deserialize the modified job model (with keyBucket). Similarly, when KeyBucket is introduced into the SSP, the serialization part of the object mapper is kept flexible enough to be able to read a 3 part SSP containing only system, stream and partition as well as a 4 part SSP-Kb which includes the keyBucket info as well. This ensures backward compatibility so that jobs with existing job models (without keyBucket) can upgrade to the newer samza version which includes keyBucket. A similar exercise is needed for the checkpoint serde residing outside the object mapper.

Future work/optimizations

1. Optimization possibly after v1: Using SSPKb in virtual tasks, a container with just one of the virtual tasks for an SSP will still consume all the records for the SSP and deserialize all of them and then discard those that dont match the SSPKb. It would have huge perf gains (saving the CPU and memory overhead) if the discarding happens prior to the deserialization.
2. Explore new intermediate stream at each virtual task count increase - this will avoid needing grouper optimization for task to container assignment. Because, if no new stream, then old intermediate stream has as many partitions as original task count. When virtual tasks come in, since the partition count of this stream is less than virtual task count, this will need to be treated like another input stream where sub-partitions are read by virtual tasks. This means a container could end up reading multiple partitions of the original intermediate stream leading to duplicate consumption perf overhead (see optimization #1)
3. Support elasticity for stateful jobs.

Appendix

Why not supporting low level and changelog based state restore: low-level jobs can have ad hoc state creation where state kv pairs need not arise from input keys. Similarly, changelog brings in the complexity arising from the partitioning scheme of the changelog stream and aligning that with the partitioning scheme of the input stream. Consider the following partitioning schemes for an input stream.

1. **Round-robin:** If input is round-robin then there is no guarantee that a key will end up in the same partition. So if the input key ends in different partitions each time, the task dealing can write to only 1 changelog partition but writing to the task's changelog means when the key ends up in another input partition then the state becomes meaningless. So, for a stateful job, with round-robin input, that wants elasticity, the effect is the same as just throwing away the state, since the job was anyways somehow able to deal with that.
2. **Hash mod/hashing:** A simple solution for this scheme would be to create a large number of changelog partitions to start with and map several of them to a single task giving the illusion of a single changelog partition for the task. For example, 4096 changelog partitions and map 128 of them to each of the 32 tasks. Issues are :
    a. Very expensive to maintain so many partitions.
    b. Needs a 1:1 function between state and input key (to figure out for a given state key which input key it belongs to)
    c. Migration for existing stateful jobs becomes painful with needing repartitioning of changelog topic.
3. **Consistent-hashing**: In this partitioning scheme, the keys of a topic are put into segments around a hash ring. Segment is the partition. For stateful jobs to work with this scheme, there needs to be a 1:1 mapping between the input segment and changelog segment. If a virtual task reads only a part of the input segment (essentially like splitting the segment) then it needs to know which part of the changelog segment to use. This can then work only if there is a 1:1 bijective function.