SEP-28: Samza State Backend Interface and Checkpointing Improvements

Status

Current state: ACCEPTED

Discussion thread: http://mail-archives.apache.org/mod_mbox/samza-dev/202106.mbox/%3cCA+6YmWVVxz=Xr244rPG2a-a6QAoR0mjrW9CK41-U7tSuv8oY4Q@mail.gmail.com%3e

JIRA: **SAMZA-2591** - Getting issue details... STATUS

Released:

Motivation

State restore is one of the most significant bottlenecks for Samza jobs to get back online since it requires incremental updates from a Kafka changelog topic (if the host the job started on does not already have the state via host affinity or standby containers). In order to decrease the state restore time, Samza must be able to use a blob store or a distributed file system for state backup, such as Azure blob store or HDFS, which will enable bulk restore of the local state store rather than incremental restore from the Kafka changelog. However, using such a remote store will increase the expected time for uploading the local RockDB changes during commit because of larger upload latency. Since we are uploading the delta to the remote store, we can expect an especially increased delta if a RocksDB compaction takes place, since it will cause the previous sstable files to be reshuffled and remerged into new files. Because Samza commit is exclusive with processing by default, making it asynchronous by default is required to introduce faster state restores without degrading processing performance for existing Samza jobs.

Although there is a current implementation for “Async commit”, the feature is only for *enabling* commit under special cases when asynchronous processing is enabled, rather than allow commit to occur asynchronously from processing and is therefore not what we need for this problem

Problem

The purpose of this enhancement proposal is to accommodate local state restore in Samza from sources other than a changelog based restore, more specifically to accommodate remote blob stores. Currently, the framework only supports incremental writes to a Kafka changelog as a remote state restore option which results in fast commit time but slow restore times. There are 2 problems arises when we migrate the framework to a remote store based backup:

1. Slow backups: To add support for remote stores, the commit phase of the Runloop must be changed to support both remote store commits as well. To counteract the larger amount of data needed to be written to the remote store, we enable the commit phase to occur concurrently to processing.
2. Backwards compatibility: the new commit scheme will have the ability to commit to both changelog and remote store for the same commit.
3. Transactional State Checkpointing: The framework does not support transactional state checkpointing for remote stores as the current format is only intended for changelog checkpoints.

Proposed Changes

Checkpoint Format changes

With the upcoming changes to the commit lifecycle to support storage systems other than changelog, we must introduce a new checkpointing format to reflect the flexibility of the stored checkpoint data. We must:

1. Distinguish the input checkpoint from the state checkpoints
2. State checkpoint must support both Kafka changelog state checkpoints and Blob store state checkpoints
3. Feature must be backwards compatible with previous checkpoints, and must be able to rollback

Instead of writing the new checkpoint message format to a new checkpoint topic (v2), we are writing to the same topic under a different namespace of the checkpoint topic (v1) is also possible for the migration. The checkpoint manager will selectively pull the checkpoints from the checkpoint topic which will match the key of the written checkpoint (Current behaviour). The advantage of this approach is that the migration will no longer require creating a new Kafka topic. However, since we are planning to enable this for all users, the versions that are currently being used, could not have forward compatibility available for being able to read the new checkpoint format from the old checkpoint stream.

The checkpoint v2 will not include another field for state checkpoints will be named to **StateCheckpointMarkers:**

```
CheckpointV2 implements Checkpoint {

private final CheckpointId checkpointId;
private final Map<SystemStreamPartition, String> inputOffsets;
private final Map<String, Map<String, String>> stateCheckpointMarkers;

}
```

The StateCheckpointMarker is a map of StateBackendFactory (see State Backup and Commit Interface Changes) to a map of store name to specific state backend offsets. For example, this would be changelog offsets for Kafka state backend, while the blob id will be used for the Blob store backend.

State Backup and Commit Interface Changes

In general, in order to accommodate for async commits, we will be splitting the commit phase into 2 sections in order to separate the async and sync operations:

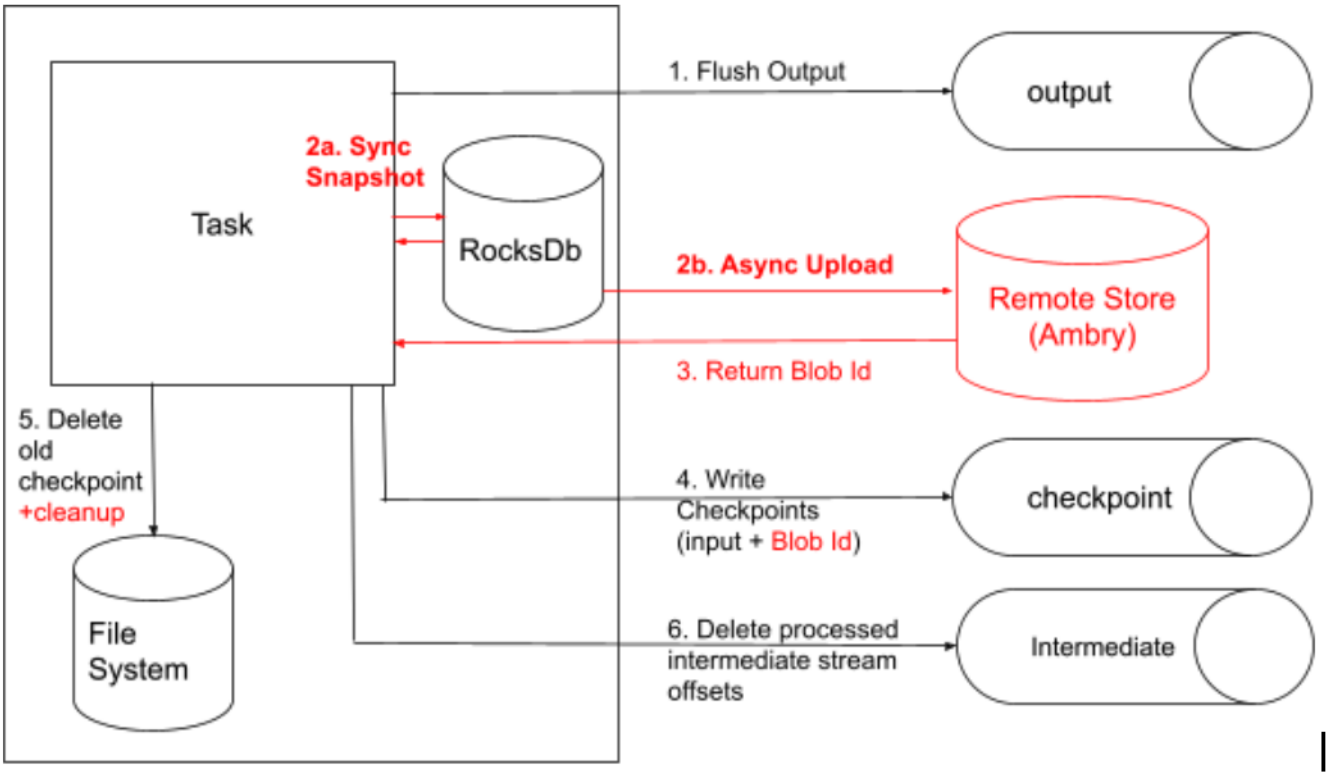
1. **Sync Snapshot**: Section that is exclusive with processing but occurs quickly. Responsible for *PREPARING FOR UPLOAD* current local state (Example: taking the delta-snapshot of the local store and store the offsets)
2. **Async Upload:** Section that occurs concurrently with processing which performs the *WRITING* to remote state (Example: upload of the delta-snapshot to remote DB and write to the checkpoint topic)

Furthermore to address exclusive, non-overlapping commits, as well as the possible case where state upload takes too long, the new **“max.commit.delay”** configuration will be introduced with a reasonable default to determine the behavior of system when the “Upload” section exceeds “task.commit.ms” during a commit, ie a second commit is triggered before all 2 sections of the last commit completed:

1. If Upload section exceeds and total time spent in the Upload section <= max.commit.delay, then we ***skip*** the current commit and allow the ongoing commit in the Upload section to finish
2. If Upload section exceeds and total time spent in the Upload section > max.commit.delay, then we ***block*** the new commit as well as any processing until the previous commit in the Upload section is finished

Finally to achieve the same transactional guarantees as the current transactional state checkpoint model, we must save the remote backup store “offsets” to the local disk and the checkpoint stream during the task commit as well, so that the restore process may map a specific on disk version to a remote version of the state store. The “offsets” in the remote store must be a unique identifier to the version of the remote store, in the case of a blob store, this will be a Blob id which is obtained from the store after a successful upload.

The new commit phase for using remote stores will be the following:

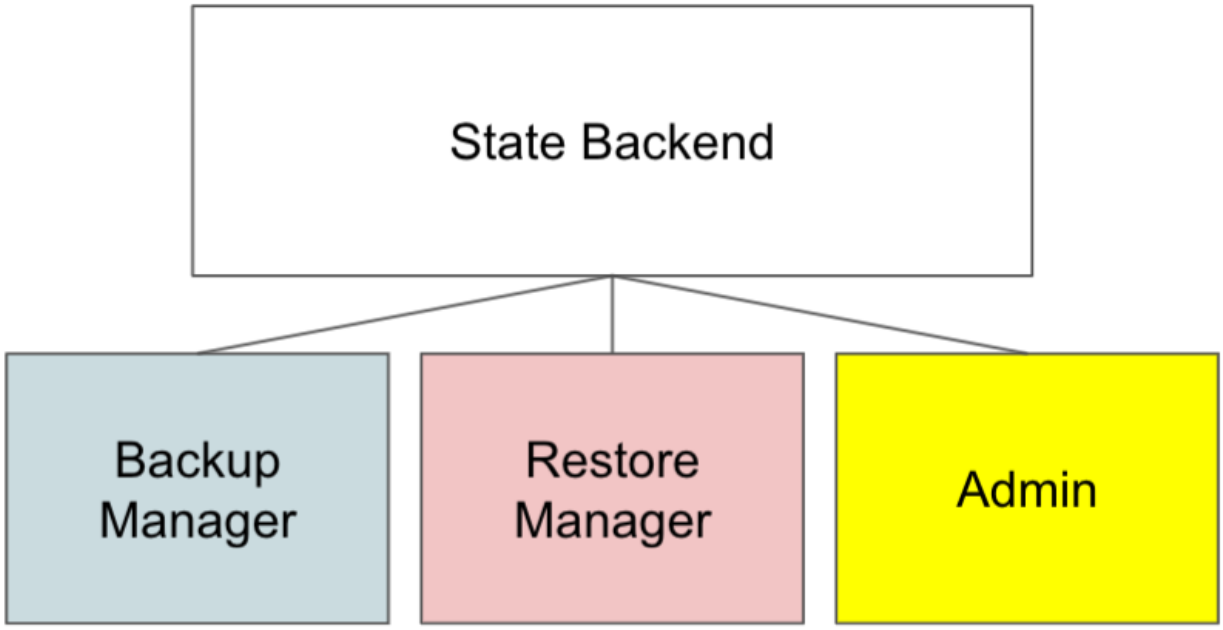


Note that since all the steps that occur after step 2b (Async Upload) are dependent on the blob id returned from the Blob store, all the following steps (3-7) must be completed concurrently to processing. The commit() call from the runloop will complete after step 2b, allowing the runloop to continue to process messages. Also note that with this model, to preserve existing functionality, the *blob id* and the *remote store* could be swapped out for *changelog offset* and *kafka changelog* respectively. For Kafka Changelog, however, writing to the topic must occur synchronously to processing, since any messages processed during the flush may result in a higher offset returned on step 3. This is why flushing the changelog must occur before the 2b step. Currently the changelog flush logic occurs in step 1 where all producers are flushed.

Another important step to note is “deleting the old checkpoint from local FS” (step 6). Although this will not be changed for now, this will be important for the remote store implementation to take note of since it will be responsible for any cleanup of the remote store state. For example, if we would like to keep X backups in the remote store, we should be able to delete blobs if the number exceeds X.

In order to add the async upload here, we would need to the remote store flush to be in the asynchronous upload phase. Also, the blob id must be added to be the information stored in the checkpoint topic and local file system.

The state interface changes introduces a new concept of the StateBackendFactory which is used to fetch the associated BackupManager, RestoreManager and Admin (Used for any resource initialization and validation) for the associated state backend. The planned supported state backends are KafkaChangeLogStateBackend and BlobStoreStateBackend.

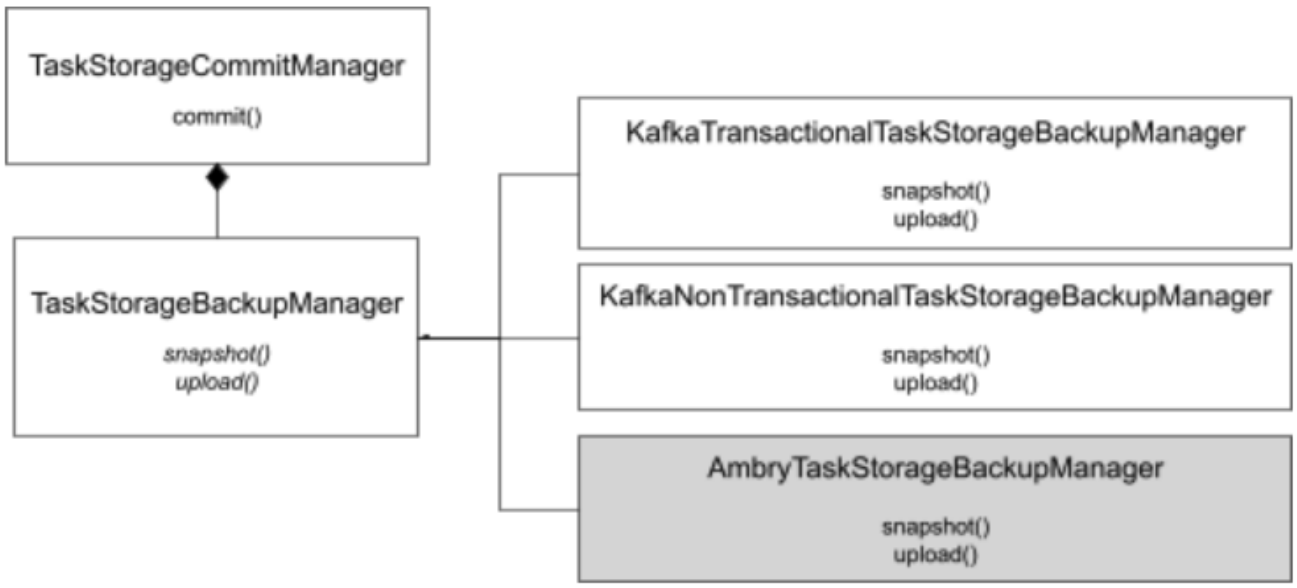


BackupManager API

In order to fully abstract out the mechanism of the upload to be async for both remote store write and the steps after a Kafka flush, we need to change the TaskStorageManager to use different flush operations depending on the store backup implementation: for kafka, we read the latest flushed changelog messages synchronously then perform the following steps asynchronously, whereas for the remote blob store, we will calculate the diff from the previous local state and initiate the upload in an async manner. Not that in both these cases the StorageEngines must be flushed beforehand. We will be replacing the TaskStorageManager class and abstracting the common operations that the store backups need to call into separate classes called "**TaskStorageCommitManager**" and "**TaskStorageBackupManager** ". These classes will handle backup of the local state storage into the remote state, depending on the specific upload implementation for the remote storage. The TaskStorageBackupManager consist of the existing StorageManage API plus the following:

1. Abstract methods **Snapshot** and **Upload** such that any specific state backup implementation must override these steps.
    a. The current behaviour for changelog will be implemented here as KafkaTractionactionalTaskStoreBackupManager and KafkaNonTransactionalTaskStoreBackupManager respectively, snapshot() for these will consist of a flush() on the Kafka changelog producer. The snapshots will be a noop since we are continuously writing to the changelog.
    b. The new behaviour for remote Blob stores will have a put call during upload() via an BlobStoreTaskStorageBackupManager. Snapshot() here will consist of creating the delta to upload().
2. Using the thread executor passed in from the container for background thread working calling the Upload() method specifically with a timeout of "max.commit.delay" (see Backup Upload Threading)
3. After the upload(), returning the ***State Checkpoint***(See Checkpoint Format Changes) so that it can be checkpointed at the same time as the input offsets to the checkpoint topic. The remote state identifier will be the offset for Changelog and Blob id for remote store.

The class structure will be the following, note that existing methods were omitted:



The lifecycle of the **TaskStorageCommitManager** will follow the current TaskStorageManager class, that is, created during container.apply() for each task via TaskStoreManagerFactory which determines the implementation of TaskStorageBackupManager to use based on configuration. Furthermore, it will contain a number of TaskStorageBackup managers based on the number of different systems that the job is configured to backup with (See Dual Commit). The TaskStorageCommitManager's responsibility is to perform the common operations of all restore managers, including flushing the state stores the correct path on disk, evoking snapshot() and upload() as well as writing the checkpoint A new config, "**stores.<store-name>.backup.factories"**, will be used which allows the user to decide the specific backup managers above used for the store (supporting BlobStore and one of KafkaTransactional or NonTransactional backup manager), with the default of *KafkaTransactionalTaskStorageBackupManager*. We additionally check for the now deprecated config "task.transactional.state.checkpoint.enabled" which will set the state backup manager to *KafkaNonTransactionalTaskStorageBackupManager* for backwards compatibility.

Backup Upload Threading Model

The Upload threads will be in a thread pool created by the container on startup. During task initialization, the container will create a thread pool that is min (number of tasks in container, 64) and will pass on the executor to each task. This is done to place a cap on the number of threads spawned for commit for a particular container process. During commit this executor will be passed onto the TaskStorageCommitManager which uses it to schedule the async upload call on the commit thread pool.

The first call to commit will set the volatile variable lastCommitStartTimeMs of an upload operation in taskInstance and submit the TaskStorageCommitManger.upload() work to the executor in the from the container. When the upload completes, it will reset the lastCommitStartTimeMs. All subsequent calls to TaskCommitManager.commit() will perform the above task if lastCommitStartTimeMs is not set, otherwise they will check the currentTimestamp against the lastCommitStartTimeMs to determine if the previous upload has exceeded "max.commit.delay".

1. If so, we block on the upload that is currently executing until currentTS - lastCommitStartTimeMs of the upload that is being executed is less than commit timeout
2. Otherwise, we proceed with processing and skip the current commit

Dual Commit

To ensure that state data is never lost during a migration to another remote store backup system, we must allow the users to backup to multiple remote storages/changelogs during the same commit phase of the job. This way, any problems onboarding with a new backup system could be easily rolled back by restoring from any supported state restore path. Furthermore, this feature could be used for cross-colo state replication in the future.

To accomplish this the following modifications must be done:

1. TaskCommitManager must be able to call the snapshot and upload phases of multiple TaskStoreBackupManagers during commit()
2. All TaskStateBackupManager must write to the checkpoint topic at the same time
3. TaskRestoreManager must have checkpoint information on all the remote backup implementations in order to be able to restore from any system.

For (1), the "**stores.<store-name>.backup.factories**" (default *KafkaTransactionalTaskStorageBackupManager*) config will be used to be a comma separated list of backup managers to enabled (planning on supporting BlobStore and Kafka)

For (2) and (3), in the case where "**stores.<store-name>.backup.factories**" contain more than one factory, the upload() phase of the backup manager is executed both on the same commit thread pool and on the callback, will merge the committed offsets and the blob id into the CheckpointV2's StateCheckpointMarkers (see Checkpoint Format Changes) field to be written to the checkpoint stream.

During restore, the implementation of the restore of the TaskRestoreManager will be decided based on a "**stores.<store-name>.restore.factory**" config. If the necessary information from the Marker is not present for the given backup manager (during migration), the job will log an ERROR and restore from the other Marker if it exists, otherwise bootstrap from the start of the Kafka changelog.

RestoreManage API

Similarly to TaskStorageManager, we must also change the behaviour of TaskRestoreManager to ensure that the restore will correctly handle the new structure of the *CheckpointV2* (See checkpoint changes) provided from the TaskStorageBackupManager. Since the existing behaviour is to trim the changelog in order to get to a state parity between the loaded local state and the offset of the changelog committed to the checkpoint topic, the same result must be guaranteed with the remote store restore. Since we are saving the blob store blob id, instead of the offset, instead of trimming, we may load the blob that matches the blob id store locally.

The checkpoint resolution would take the following steps:

1. If the local checkpointId matches the remote checkpointId (pulled from checkpointV2 of the checkpoint stream), use that version
2. Else load the state (either changelog offset or blob store id) from the checkpointV2

Note that for (2a), we are picking the latest checkpointId from the checkpoint stream to preserve the existing transactional state functionality. In order to get the exact copy of the state at commit, we trim the changelog/ fetch matching blob id. The local checkpointId could only be newer than the remote checkpointId , since write checkpoint Id to FS (step 4) completed before the write to checkpoint stream (step 5).

For cases where users have performed dual commit backups (ie commit to both to remote store and to changelog), the checkpoint topic's checkpointId information will contain both the changelog offset as well as the RemoteStoreMetadata in CheckpointedRestoreMarker. The source of the backup system will be decided on the **task.backup.manager** configuration.

Public Interfaces

Checkpoint.java

```
public interface Checkpoint {
  /**
   * Gets the version number of the Checkpoint
   * @return Short indicating the version number
   */
  short getVersion();

  /**
   * Gets a unmodifiable view of the last processed offsets for {@link SystemStreamPartition}s.
   * The returned value differs based on the Checkpoint version:
   * <ol>
   *    <li>For {@link CheckpointV1}, returns the input {@link SystemStreamPartition} offsets, as well
   *      as the latest KafkaStateChangelogOffset for any store changelog {@link SystemStreamPartition} </li>
   *    <li>For {@link CheckpointV2} returns the input offsets only.</li>
   * </ol>
   *
   * @return a unmodifiable view of last processed offsets for {@link SystemStreamPartition}s.
   */
  Map<SystemStreamPartition, String> getOffsets();
}
```

StateBackendFactory.java

```
/**
 * Factory to build the Samza {@link TaskBackupManager}, {@link TaskRestoreManager} and {@link
StateBackendAdmin}
 * for a particular state storage backend, which are used to durably backup the Samza task state.
 */
public interface StateBackendFactory {
  TaskBackupManager getBackupManager(JobContext jobContext,
      ContainerModel containerModel,
      TaskModel taskModel,
      ExecutorService backupExecutor,
      MetricsRegistry taskInstanceMetricsRegistry,
      Config config,
      Clock clock,
      File loggedStoreBaseDir,
      File nonLoggedStoreBaseDir);

  TaskRestoreManager getRestoreManager(JobContext jobContext,
      ContainerContext containerContext,
      TaskModel taskModel,
      ExecutorService restoreExecutor,
      MetricsRegistry metricsRegistry,
      Config config,
      Clock clock,
      File loggedStoreBaseDir,
      File nonLoggedStoreBaseDir,
      KafkaChangelogRestoreParams kafkaChangelogRestoreParams);

  StateBackendAdmin getAdmin(JobModel jobModel, Config config);
}
```

TaskBackupManager.java

```
/**
 * <p>
 * TaskBackupManager is the interface that must be implemented for any remote system that Samza persists its
state to
 * during the task commit operation.
 * {@link #snapshot(CheckpointId)} will be evoked synchronous to task processing and get a snapshot of the
stores
 * state to be persisted for the commit. {@link #upload(CheckpointId, Map)} will then use the snapshotted state
 * to persist to the underlying backup system and will be asynchronous to task processing.
 * </p>
 * The interface will be evoked in the following way:
 * <ul>
 *   <li>Snapshot will be called before Upload.</li>
 *   <li>persistToFilesystem will be called after Upload is completed</li>
 *   <li>Cleanup is only called after Upload and persistToFilesystem has successfully completed</li>
 * </ul>
 */
public interface TaskBackupManager {

  /**
   * Initializes the TaskBackupManager instance.
   *
   * @param checkpoint last recorded checkpoint from the CheckpointManager or null if no last checkpoint was
found
   */
  void init(@Nullable Checkpoint checkpoint);

  /**
   *  Snapshot is used to capture the current state of the stores in order to persist it to the backup manager
in the
   *  {@link #upload(CheckpointId, Map)} (CheckpointId, Map)} phase. Performs the commit operation that is
   *  synchronous to processing. Returns the per store name state checkpoint markers to be used in upload.
   *
   * @param checkpointId {@link CheckpointId} of the current commit
   * @return a map of store name to state checkpoint markers for stores managed by this state backend
   */
  Map<String, String> snapshot(CheckpointId checkpointId);

  /**
   * Upload is used to persist the state provided by the {@link #snapshot(CheckpointId)} to the
   * underlying backup system. Commit operation that is asynchronous to message processing and returns a
   * {@link CompletableFuture} containing the successfully uploaded state checkpoint markers .
   *
   * @param checkpointId {@link CheckpointId} of the current commit
   * @param stateCheckpointMarkers the map of storename to state checkpoint markers returned by
   *                               {@link #snapshot(CheckpointId)}
   * @return a {@link CompletableFuture} containing a map of store name to state checkpoint markers
   *         after the upload is complete
   */
  CompletableFuture<Map<String, String>> upload(CheckpointId checkpointId, Map<String, String>
stateCheckpointMarkers);

  /**
   * Cleanup any local or remote state for checkpoint information that is older than the provided checkpointId
   * This operation is required to be idempotent.
   *
   * @param checkpointId the {@link CheckpointId} of the last successfully committed checkpoint
   * @param stateCheckpointMarkers a map of store name to state checkpoint markers returned by
   *                               {@link #upload(CheckpointId, Map)} (CheckpointId, Map)} upload}
   */
  CompletableFuture<Void> cleanUp(CheckpointId checkpointId, Map<String, String> stateCheckpointMarkers);

  /**
   * Shutdown hook the backup manager to cleanup any allocated resources
   */
  void close();

}
```

TaskRestoreManager.java

```
/**
 * The helper interface restores task state.
 */
public interface TaskRestoreManager {

  /**
   * Initialize state resources such as store directories.
   */
  void init(Checkpoint checkpoint);

  /**
   * Restore state from checkpoints, state snapshots and changelogs.
   * Currently, store restoration happens on a separate thread pool within {@code ContainerStorageManager}. In
case of
   * interrupt/shutdown signals from {@code SamzaContainer}, {@code ContainerStorageManager} may interrupt the
restore
   * thread.
   *
   * Note: Typically, interrupt signals don't bubble up as {@link InterruptedException} unless the restore
thread is
   * waiting on IO/network. In case of busy looping, implementors are expected to check the interrupt status of
the
   * thread periodically and shutdown gracefully before throwing {@link InterruptedException} upstream.
   * {@code SamzaContainer} will not wait for clean up and the interrupt signal is the best effort by the
container
   * to notify that its shutting down.
   */
  void restore() throws InterruptedException;

  /**
   * Closes all initiated resources include storage engines
   */
  void close();
}
```

TaskStorageAdmin.java

```
/**
 * Admin responsible for loading any resources related to state backend
 */
public interface StateBackendAdmin {
  /**
   * Create all the resources required per job per store state backend
   */
  void createResources();

  /**
   * Validate all resources required per job per state for state backend
   */
  void validateResources();
}
```

Implementation and Test Plan

List of items to validate:

1. Serde of checkpoints v2: successfully reads and writes v2 checkpoints
2. Backwards compatibility for checkpoints v2: can read and deserialize v1 checkpoints when “readV2Checkpoints” are disabled
3. Forwards compatibility for checkpoints v2: successfully skips new read checkpoints v2 with different keys on previous versions samza
4. Dual checkpoints write: v2 writes both the v1 checkpoint message and v2 checkpoint message must be present in to checkpoint topic
5. Test switching readCheckpointV2 enabled config: could switch between v2 and v1 checkpoint read on the latest version via config in new version

We need to cross check this feature heavily with the blob store backup implementation after it is completed:

- Avg upload speed (also upload speed after RocksDB compaction) to set max.commit.delay

For functionality testing, this feature needs to be tested with:

- Existing Kafka changelog backup
  - Async processing enabled and disabled
  - Transactional state enabled and disabled
- Future blob store backup
  - Async processing enabled and disabled

For scalability testing, we will need to create a job for each of the implementations, running on EI and PROD with a large number of containers > 64 preferably. We have to test the following edge cases:

- containers with tasks > 64
- Jobs calling process after every message

Compatibility, Deprecation, and Migration Plan

Checkpoint V2 Migration Plan

The rollout occur in the following steps:

1. Create new checkpoint messages with a new version (v2)
2. Write checkpoints to ***both*** v1 and v2 checkpoints, but read only from the v1 checkpoint
3. Enable job to restore from the new checkpoint message (via config)
4. Ensure that the job is stable with the new checkpoint scheme on new topic
5. Stop write to old checkpoint topic and delete old checkpoint topic (*Cannot roll back at this point*)

For step 5), this may be running for up to a quarter before the checkpoint could be deleted to ensure stability, since step 4 cannot be rolled back.

Backup and Restore Interfaces Migration Plan

Based on the checkpointId retrieved from checkpoint topic, the restore task must be able to determine the specific TaskRestoreManager to be used. This way, any task backup manager migration would be simple:

1. Change **stores.<store-name>.backup.factories** to be desired target managers of BlobStore and Kafka (Default to kafka)
2. Wait for the checkpoints v2 to be written
3. Task restart will load from the new restore manager (based on checkpoint in based on **stores.<store-name>.restore.factory** config)

Rejected Alternatives

Single Threaded per Task Executor

Initially, the idea of a single threaded executor in the TaskBackupManager was considered, to keep the threading model simple and strictly per task. Upload will be done with a single threaded executor managed by the TaskStorageBackupManager, resulting in 1 upload thread per task. These threads will be easily identified during a thread dump and the mapping to task should be obvious.

However, for cases where there is a large number of tasks per container which greatly exceeds the number of cores on our machines, creating another thread per task is not feasible and we may run into file descriptor limits and overall suboptimal threading performance. Since as a framework we cannot control the number of tasks per container currently, we will avoid this situation by capping the number of commit threads to a specific number per container as described in the Backup Upload Threading section.