FLIP-41: Unify Binary format for Keyed State

Authors: Tzu-Li (Gordon) Tai, Congxian Qiu, Stefan Richter, Kostas Kloudas

Status

Discussion thread	http://apache-flink-mailing-list-archive.1008284.n3.nabble.com/DISCUSS-FLIP-41-Unify-Keyed-State-Snapshot-Binary-Format-for-Savepoints-td29197.html
Vote thread	https://lists.apache.org/thread.html/f6fcb40d2bf9d2a889eb20cc7b38897b340ea897832770ad2b7b27d5%40%3Cdev.flink.apache.org%3E
JIRA	**FLINK-20976** - Getting issue details... STATUS
Release	1.13

Motivation

Currently, Flink managed user keyed state is serialized with different binary formats across different state backends.

The inconsistency exists as the following, for both checkpoints and savepoints:

- Different ways of writing metadata to facilitate iterating through serialized state entries across key groups on restore.
- Keys and values of the keyed state entries (e.g. `ValueState`, `ListState`, `MapState`, etc.) also have different binary formats in snapshots.

The differences mainly root from the fact that different state backends maintain working state differently, and consequently have specialized formats that allows them to introduce as less overhead as possible when taking a snapshot of the working state as well as restoring from the serialized form.

While for checkpoints it is completely reasonable to have state backend specific formats for more efficient snapshots and restores, savepoints should be designed with interoperability in mind and allow for operational flexibilities such as swapping state backends across restores.

Moreover, with the current status of state backend code, state backends are responsible of defining what format they write with. This adds overhead to developing new state backends in the future, as well as the possibility that yet another incompatible format is introduced, making the unification even harder to achieve.

So, the main goal of this proposal is the following:

- Unify across all state backends a savepoint format for keyed state that is more future-proof and applicable for potential new state backends. Checkpoint formats, by definition, are still allowed to be backend specific.
- Rework abstractions related to snapshots and restoring, to reduce the overhead and code duplication when attempting to implement a new state backend.

Current Status

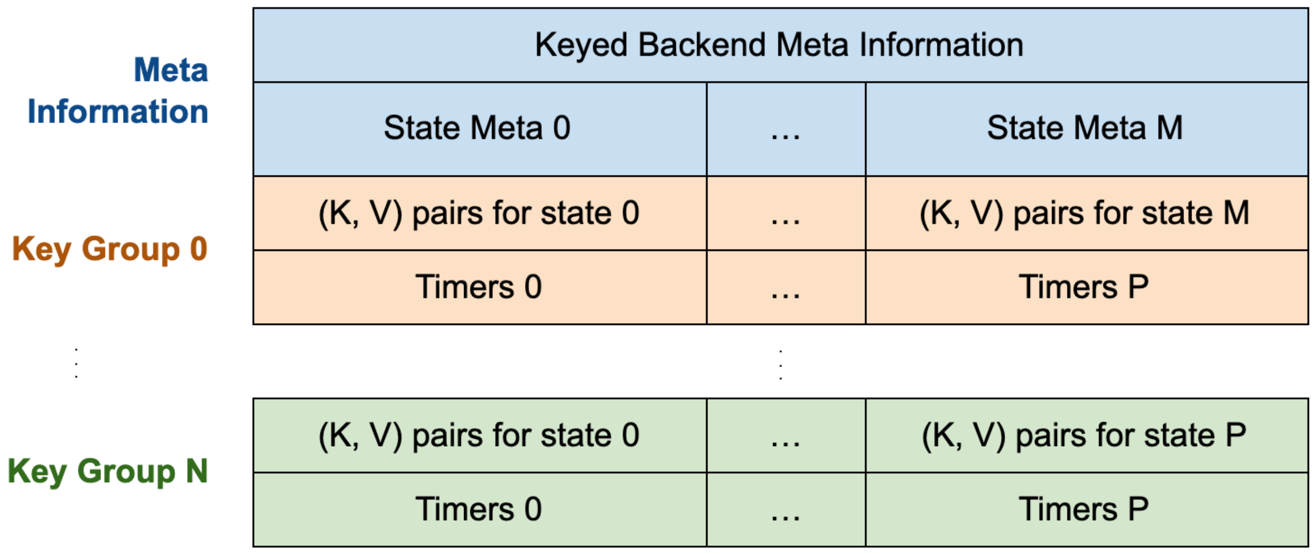
NOTE - the remainder of this document uses the following abbreviations.

- **KG:** key group
- **K:** partition key
- **NS:** namespace
- **SV:** state value (e.g. value of a `ValueState`, complete map of a `MapState`, etc.)
- **UK:** key in a user `MapState`
- **UV:** value in a user `MapState`
- **TS:** timestamp (of timers)

For us to reason about the proposed unified format later in this document, it is important to understand the current formats and the historical reasons of why they were defined as they are right now.

The written keyed state in snapshots of each operator contains mainly two parts -

1. meta information for the keyed backend, such as the snapshot of the key serializer being used, as well as meta information about the registered keyed states.
2. actual state entries, sequence ordered by `(KG, State ID)`.



Writing of the meta information is governed by the `KeyedBackendSerializationProxy` class. The class is commonly used by all subclasses of `SnapshotStrategy`, such as `RocksFullSnapshotStrategy` and `HeapSnapshotStrategy`, before writing the state data. Therefore, this part is already unified.

The difference lies in the format of the sequence of state entries. Specifically, they differ by -

1. the binary format of the key and values in each state entry, and
2. metadata or markers written along with the contiguous state entries to facilitate reading them from state handle streams on restore.

State entries key value binary layout

The difference in binary layout for key and values in state entries across state backends relates to how they are tailored for how each backend maintains working state.

For the `HeapKeyedStateBackend`, working state is maintained as in-memory state tables that are partitioned by key group. The state tables are nested maps of `Map<NS, Map<K, SV>>`, providing an outer scope by namespace and an inner scope by key.
Naturally, it is easy to iterate through the state objects by key group when writing to the checkpoint stream during a snapshot. Offsets for each key group is written in the savepoint metadata file, and each state value is written with its namespace and key so that it on restore, deserialized state objects can correctly re-populate the state tables.

For the `RocksDBKeyedStateBackend`, working state is already serialized and maintained as key-value pairs in RocksDB, with all pairs for a registered state stored in a single column family. In order to be able to iterate through states values in a column family by order of key group during a snapshot, the keys of working state in RocksDB are prepended with the key group ID (int), relying on the fact that RocksDB organizes the data in sorted ordering on the keys. Theoretically, since we already write the offsets of each key group in the savepoint's metadata, the final snapshotted binary layout does not need to have the key group prepended on each key. However, to exclude that, we would introduce extra overhead around removing and prepending again the key group bytes during snapshotting and restoring bytes back into RocksDB. This isn't reasonable considering the small size of the key group bytes; therefore, each snapshotted state value of a `RocksDBKeyedStateBackend` has the exact same layout as working state, containing also the key group bytes.

The table below provides an overview of the formats for the supported state primitives. Note that although the formats below are represented as tuples (2-tuple for RocksDB to represent it as a key-value entry in RocksDB, and a 3-tuple for heap to represent the levels of the nested map of a state table), the binary layout in the snapshot is really just a direct concatenation of all the serialized values.

	RocksDBKeyed StateBackend	**HeapKeyed StateBackend**	**Note**
ValueState	`[CompositeKey (KG, K, NS), SV]`	`[NS, K, SV]`	• `CompositeKey(...)` represents the composite key built via `RocksDBSerializedCompositeKeyBuilder#buildCompositeKeyNamespace`. • Moreover, the MSB bit of the composite key may also be flipped if it is the last entry of a state for a given key group (see `RocksFullSnapshotStrategy#setMetaDataFollowsFlagInKey`). Note: this only applies to snapshotted state; keys in working state should never have the MSB bit flipped.
ListState	`[CompositeKey (KG, K, NS), SV]`	`[NS, K, SV]`	• The format of SV for `ListState` is different between heap and RocksDB backend. • For the heap backend, the format is defined by the serializer obtained from the state descriptor, which is a `ListSerializer`. • RocksDB backend defines its own format in `RocksDBListState#serializeValueList`.

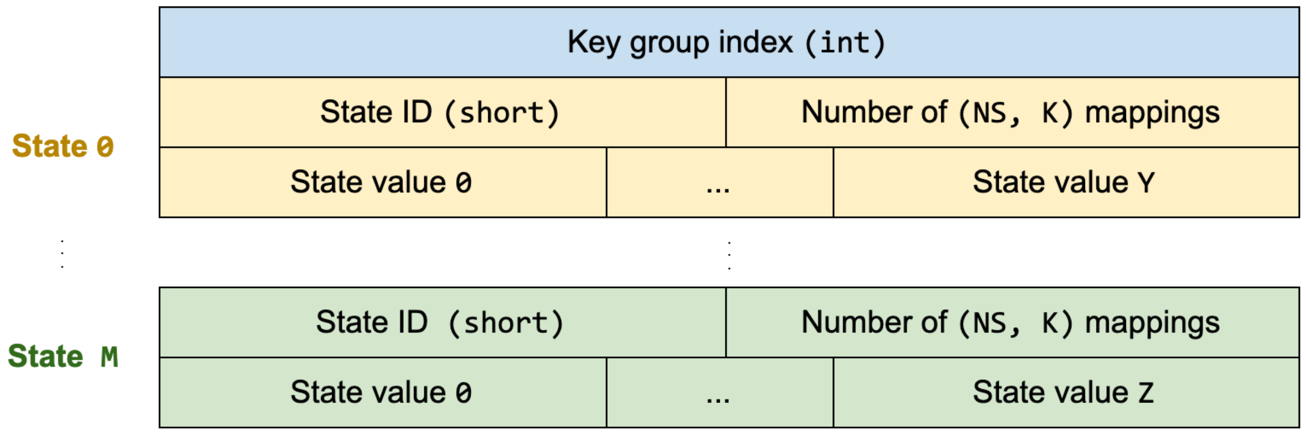
MapState	`[CompositeKey (KG, K, NS) :: UK, UV]`	`[NS, K, SV]`	• For the heap backend, each serialized state data is the complete user `MapState`, written using the serializer obtained from the state descriptor, which is a `MapSerializer`. • For RocksDB backend, each serialized state data is a single entry in the user Map`State`. User map key and value serializers are obtained from the `MapSerializer` provided by the state descriptor.
AggregatingState	`[CompositeKey (KG, K, NS), SV]`	`[NS, K, SV]`	
ReducingState	`[CompositeKey (KG, K, NS), SV]`	`[NS, K, SV]`	
FoldingState	`[CompositeKey (KG, K, NS), SV]`	`[NS, K, SV]`	
Timers	`[KG :: TS :: K :: NS, (empty)]`	`TS :: K :: NS`	• the timestamp is a long value with MSB sign bit flipped. • Value is empty in RocksDB

Metadata or markers for state iteration

The heap and RocksDB backend has different approaches in how they write metadata or markers in the checkpoint stream to facilitate iterating through contiguous state values of key groups at restore time.
The reason for the difference is mainly due to the fact that for the heap backend, the number of state values is known upfront when taking a snapshot. The following subsections describes this in detail.

`HeapKeyedStateBackend`

For the `HeapKeyedStateBackend`, the binary layout of the contiguous state values of a single key group is as follows:

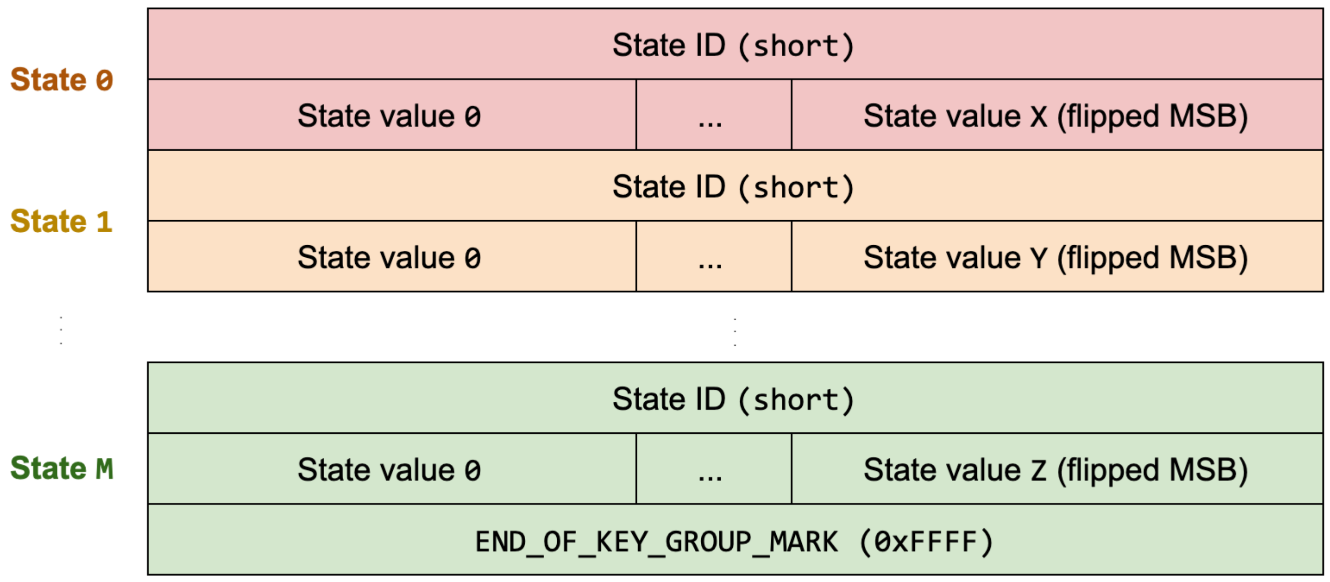


- The key group index at the beginning of the key group is not a requirement for being able to iterate the contiguous state values. It is written just for sanity checks on restore that the following state values do belong to the expected key group identified by its offset.
- The number of states to read from the stream was already determined after reading meta information about the keyed state state backend.
- The number of `(NS, K)` state entries to read for each state is written before the contiguous values. It is possible to get the total number of mappings upfront because state tables are in-memory.

Please see `HeapRestoreOperation#readKeyGroupStateData()` for the implementation.

`RocksDBKeyedStateBackend`

For the RocksDB`KeyedStateBackend`, the binary layout of the contiguous state values of a single key group is as follows:



- Unlike the heap backend, it is not possible to know the number of `(NS, K)` state value mappings upfront when taking a snapshot. Instead, the RocksDB backend writes metadata along with the stream of contiguous state values in a key group.
- The last state value mapping for a keyed state will have its MSB bit flipped to indicate that it is the last value for the currently iterated keyed state. Since the values are prepended with the key group which is always a positive integer, the result of flipping the MSB bit of the serialized value would always be setting the MSB bit to 1.
- When iterating through the contiguous values, if a value with a contiguous bit is reached, the next value would either be a state ID which indicates that the next values are of another new keyed state of the key group, or an `END_OF_KEY_GROUP_MARK` which indicates that there are no more values to be read for the key group.

Please see `RocksDBFullRestoreOperation#readKvStateData()` for the implementation.

Proposal

This proposal covers 2 goals:

- Define the unified binary format for keyed state backends.
- Extend / adjust internal abstractions around SnapshotStrategy and RestoreOperation so that new backends added in the future writes keyed state in savepoints in the unified format.

Unified binary format for keyed state

We propose to unify the binary layout for all currently supported keyed state backends to match the layout currently adopted by RocksDB, for all levels including the layout of contiguous state values of a key group as well as layout of individual state primitives.

The main consideration points for this decision is the following:

- The current way the heap backends iterates state values puts a constraint that the keyed state backend needs an efficient way to know the total number of `(NS, K)` state entries, so that this count can be written before the contiguous state values. This highly depends on the state backend; for example, for RocksDB obtaining this count is non-trivial. The approach that RocksDB adopts, by injecting markers and flags to indicate the end of states and key groups when reading from the file streams is definitely the more general and future-proof approach.
- The RocksDB backend snapshot format has excessive written data due to the key group bytes prepended for each `(NS, K)` state value mapping. As previously explained, the key group bytes are actually not required to be written with every state mapping in snapshots; they are only required for working state maintained by RocksDB to have an ordering of key value entries by key group. However, the size overhead of this redundancy would be far less prominent than the extra work that RocksDB otherwise requires to truncate and prepend again the key group bytes on each snapshot and restore operation.
- Keyed `MapState` are written as a single state value with the current heap backend's snapshot layout, while for RocksDB backend, a single state value maps to an entry in the `MapState`. The advantage in maintaining entries in the `MapState` as independent key-values in RocksDB is obvious - accessing and updating random keys in the `MapState` would not require serializing and deserializing the whole map. Trying to merge the separate entries when snapshotting so that the snapshotted version of `MapState` is a single state value would not work, since a user `MapSta te` can have arbitrarily large number of entries that would not fit into memory (again, this is a constraint because the serializer for writing maps, i. e. the `MapSerializer`, writes in a format that requires the total number of mappings to be known upfront). Moreover, the work of merging the entries on snapshot and flattening them again simply is non-trivial compared to dumping the key value bytes in RocksDB.

To conclude this, the format that RocksDB currently uses is the more general approach for arbitrarily large keyed state. It is feasible to allow heap backend to work with RocksDB backend's current format, but not the other way around.

Implementation

The goal for the proposed implementation is the following:

- Allow backends to have different snapshot formats for checkpoints and savepoints
- Allow seamless migration from previous Flink versions
- Use a common savepoint strategy used by all keyed backends to define the unified per key group binary layout of contiguous state entries
- Let binary layouts of state entries be defined by key and value serializers of each state primitive's descriptor. Internal state accessors that need to eagerly serialize / deserialize state (e.g. subclasses of `AbstractRocksDBState`) should be able to use those directly, instead of defining their own format.

The following sections goes over the main design choices.

Rework `SnapshotStrategy` class hierarchy to differentiate between savepoint and checkpoint strategies

Currently, the binary layout for keyed state in savepoints for the heap and RocksDB state backend is defined by `HeapSnapshotStrategy` and `RocksFullSnapshotStrategy`, respectively. Those classes are used for both savepoints as well as full checkpoints. To be able to define the layout for savepoints and checkpoints of keyed state independently, the hierarchy for `SnapshotStrategy` needs to be extended to differentiate a `KeyedBackendSavepointStrategyBase` and a `KeyedBackendCheckpointStrategyBase`. All existing strategies, including `HeapSnapshotStrategy`, `RocksFullSnapshotStrategy` and `RocksDBIncrementalSnapshotStrategy` should be rebased onto the checkpoint strategy base (and potentially renamed for better clarity).

Moreover, it should be made explicit that all keyed backends are composed of two different snapshotting strategies, one for savepoints and one for checkpoints. Therefore, the `AbstractKeyedBackend` should be instantiated with a `KeyedBackendSavepointStrategyBase` and a `KeyedBackendCheckpointStrategyBase`, with the correct one being respected on each snapshot attempt. This would eliminate the need for implementations of `AbstractKeyedBackend` to be responsible of any snapshotting concerns as it now comes via composition.

As part of this refactoring effort, we should also work towards a clearer base abstraction for all snapshot strategy implementations. For example, the strategies now commonly include a synchronous part that essentially prepares resources to be used by the asynchronous part of the snapshot. This can be better abstracted as the following:

```
public abstract AbstractSnapshotStrategy<T extends StateObject, SR extends SnapshotResources> implements
SnapshotStrategy<SnapshotResult<T>> {
    @Override
    public final RunnableFuture<SnapshotResult<T>> snapshot(...) {
        SnapshotResources snapshotResources = syncPrepareResources();
        return asyncSnapshot(snapshotResources, ...);
    }

    protected abstract SR syncPrepareResources();
    protected abstract RunnableFuture<SnapshotResult<T>> asyncSnapshot(
        SR syncPartResource,
        long checkpointId,
        long timestamp,
        CheckpointStreamFactory streamFactory,
        CheckpointOptions checkpointOptions);
}
```

`SnapshotResources` encapsulates resources for the asynchronous part of snapshots, such as -

- read-only snapshot of the state (for RocksDB, a snapshot of the database and for heap, a snapshot of the copy-on-write state tables). Any resources created need to be released when the asynchronous part completes.
- snapshot of all registered states' meta information

```
public interface SnapshotResources {
    void release();
    List<StateMetaInfoSnapshot> getStateMetaInfoSnapshots();
}
```

Unifying the format for keyed state via `KeyedBackendSavepointStrategyBase`

`KeyedBackendSavepointStrategyBase` is responsible for defining the unified binary layout for keyed state in savepoints. Subclasses would only be responsible for providing a state backend specific `KeyedStateSavepointSnapshotResource`. Apart from the state snapshot and meta info snapshots, the `KeyedStateSavepointSnapshotResource` additionally provides iterators for snapshotted keyed state -

```
public interface KeyedStateSavepointSnapshotResources extends SnapshotResources {
    List<KeyedStateSnapshotIterator> getKeyedStateSnapshotIterators();
}
```

The asynchronous part of the snapshot defines the per key group unified binary layout for keyed state in savepoints, and is made final. It returns a `SavepointKeyGroupsStateHandle`, which functionality wise is currently identical to `KeyGroupsStateHandle`. The reason to introduce a new type of state handle specifically for savepoints is so that on restore, we can use the state handle type to determine which restore strategy to use.

```
public abstract class KeyedBackendSavepointStrategyBase<K, KSR extends KeyedStateSavepointSnapshotResource>
implements AbstractSnapshotStrategy<SavepointKeyedStateHandle, KSR> {
    @Override
    public final RunnableFuture<SavepointKeyedStateHandle> asyncSnapshot(KSR resources, ...) {
        // return an AsyncPartCallable implementation that defines the unified per key group layout of state
entries
    }

    @Override
    protected abstract KSR syncPrepareResources();
}
```

Within the asynchronous part of the snapshot, keyed state snapshot iterators are obtained from the `KeyedStateSavepointSnapshotResource`. Please refer to the next section for more details regarding the iterators.

Iterating state entries for keyed state snapshots: `KeyedStateSnapshotIterator` and `KeyedStateSnapshotPerKeyGroupMergeIterator`

A `KeyedStateSnapshotIterator` is an iterator that iterates a single registered state entries in key group order. The asynchronous part of the snapshot operation then uses a `KeyedStateSnapshotPerKeyGroupMergeIterator` to combine multiple `KeyedStateSnapshotIterator` to iterate across all registered states key group by key group.

The interface for the `KeyedStateSnapshotIterator` is proposed to be the following:

```
public interface KeyedStateSnapshotIterator {
    void next();
    int getStateId();
    int getKeyGroup();
    byte[] getKey();
    byte[] getValue();
    boolean isNewKeyGroup();
    boolean isValid();
}
```

The iterator returns a `(byte[], byte[])` key value pair for each state entry. Subclass implementations are responsible for serializing state objects in the case of state backend that only lazily serializes state on snapshots (e.g. the heap backend). This should also provide enough flexibility for state backends that may have a mix of serialized and non-serialized working state, such as the disk-spilling heap backend that is currently being discussed [1].

Restore procedures and migrating from older versions

Likewise to the rework of the `SnapshotStrategy` hierarchy to differentiate between savepoints and checkpoints, the `RestoreOperation` hierarchy should also be changed correspondingly to include `KeyedBackendSavepointRestoreOperation` and `KeyedBackendCheckpointRestoreOperation`. All existing implementations for keyed backend restore procedures, namely `HeapRestoreOperation`, `RocksDBFullSnapshotRestoreOperation`, and `RocksDBIncrementalSnapshotRestoreOperation` should be rebased onto the checkpoint restore operation and possibly renamed for clarity purposes.

On restore, the keyed state backend builders first check the type of the assigned restored state handles. The following scenarios may be encountered:

- If it is a `SavepointKeyGroupsStateHandle`, then the restored state is assured to be a savepoint of the new unified format. It is safe to restore state from the state handles using `KeyedBackendSavepointRestoreOperation`.
- If it is a `KeyGroupsStateHandle`, then we have either restored from a checkpoint after the rework, or a savepoint before the rework. Either way, it will be safe to continue using the existing keyed state restore operations (`HeapRestoreOperation`, `RocksDBFullSnapshotRestoreOperation`, and `RocksDBIncrementalSnapshotRestoreOperation`) to read state, since these were previously used for savepoints as well.

Migrating from Previous Savepoints

With the proposed implementation, users will be able to seamlessly migrate from previous savepoints of older Flink versions. When reading from older versions, old read paths will be used. From then on, new savepoints of keyed state will be written in the new unified format.

References

[1] http://apache-flink-mailing-list-archive.1008284.n3.nabble.com/DISCUSS-Proposal-to-support-disk-spilling-in-HeapKeyedStateBackend-td29109.html