Upgrading Apache Flink Applications: State of the Union

- Tzu-Li (Gordon) Tai, Software Engineer at data Artisans & Apache Flink PMC

dataArtisans

ABOUT DATA ARTISANS

Original Creators of
Apache Flink®

dA

PLATFORM

Real Time Stream Processing
Enterprise Ready

Upgrading Stateful Flink Streaming Applications

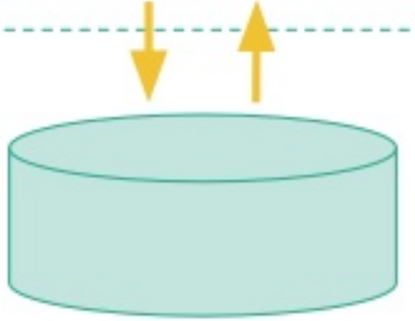
overview of the general concerns ...

 |

Anatomy of a Flink stream job upgrade

Flink job user code

Local state backends



local read / writes that manipulate state

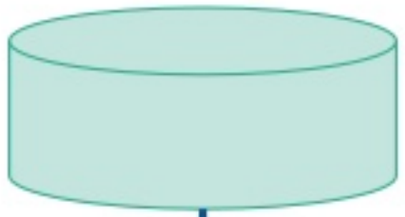
Persisted state bytes

Anatomy of a Flink stream job upgrade

Flink job user code

Local state backends

Persisted state bytes



local read / writes that manipulate state

persist to DFS on savepoint

Anatomy of a Flink stream job upgrade

Flink job user code

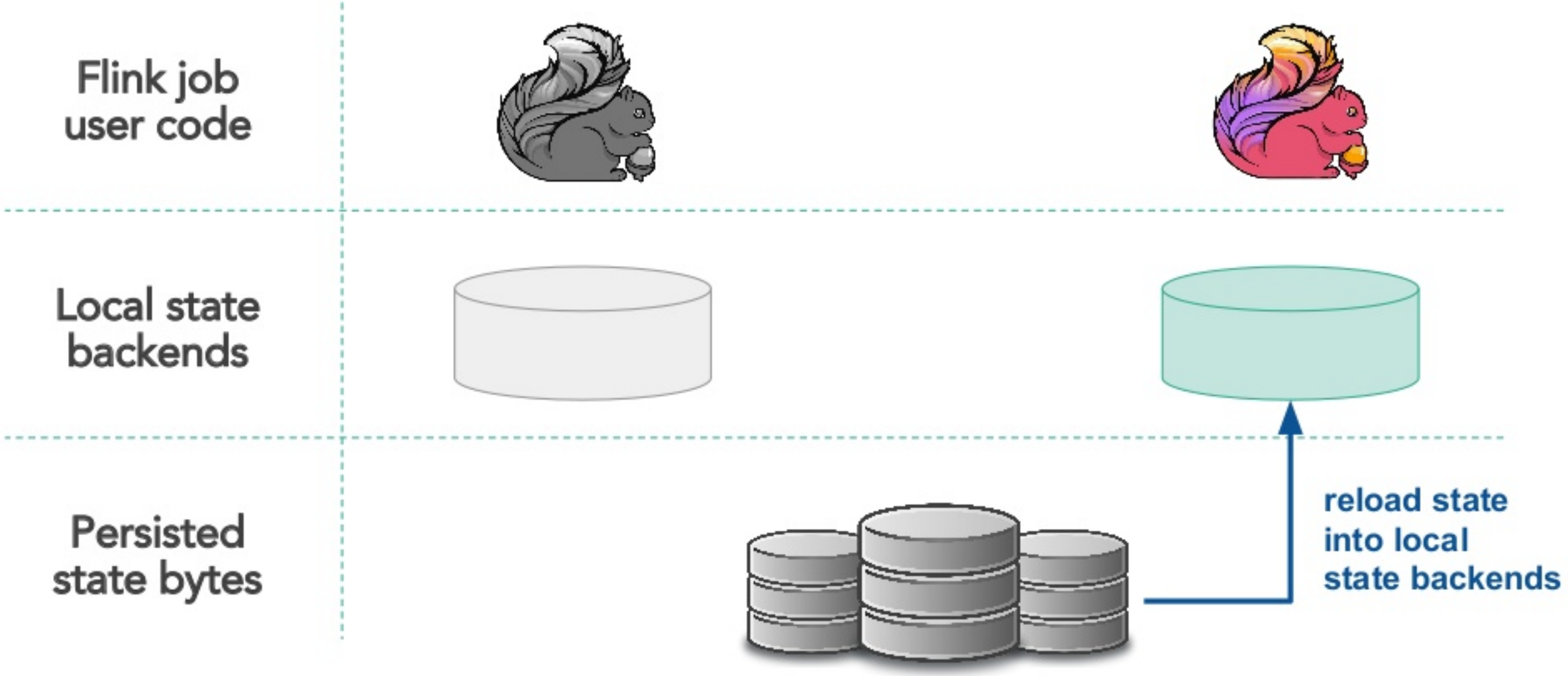
Upgrade Application

1. *Upgrade Flink cluster*
2. *Fix bugs*
3. *Pipeline topology changes*
4. *Job reconfigurations*
5. *Adapt state schema*
6. *Upgradability dry-run*
7. *...*

Local state backends

Persisted state bytes

Anatomy of a Flink stream job upgrade

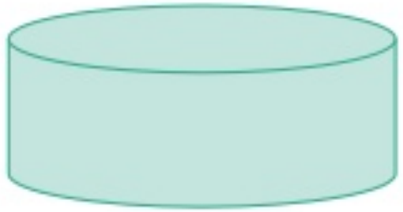


Anatomy of a Flink stream job upgrade

Flink job user code

Local state backends

continue to manipulate state



Persisted state bytes

What's already / almost available?

- ✓ Pipeline topology changes
  - Removing / adding operators
- ✓ Job reconfigurations
  - Rescale job / operator parallelism
  - Swapping state backends
    - Not yet available, but is a low hanging fruit
    - Requires unification of the savepoint formats between different state backends

What have users been asking for?

? State schema migration
  - Aapting state schema to new business logic, and migrating from old schema

? Savepoint manipulation
  - Re-calculating erroneous state caused by user code bugs
  - State bootstrapping for new operators

? Upgradability dry-runs
  - Detecting upgrade incompatibilities offline; e.g. a tool that takes 2 job jars and checks between them

State Schema Migration

State Schema

- The schema of state in Flink jobs changes along with business logic
  - Schema is determined by means of the state serializer

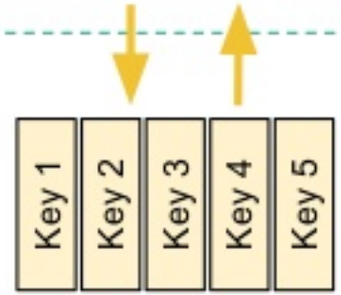
- Different serialization behaviours between heap-backed / out-of-core state backends complicates the process

State serialization in heap-based backends

Flink job user code

```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV1()
    );
```

Local state backends



Persisted state bytes

State serialization in heap-based backends

Flink job user code

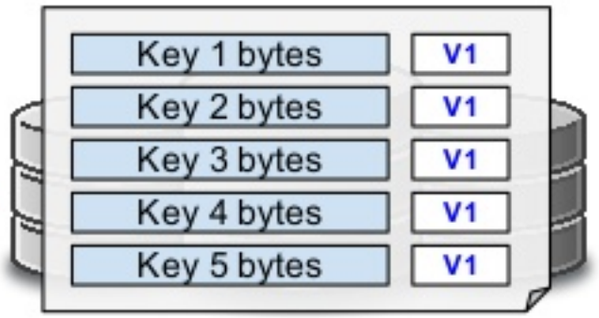
Local state backends

Persisted state bytes

```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV1()
    );
```

Serialized by V1 serializer

... 010101110



 |

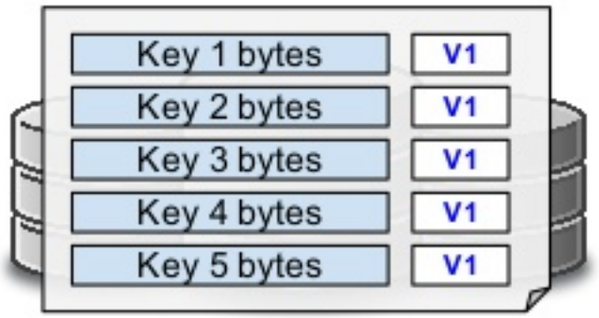
State serialization in heap-based backends

Flink job user code

```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV2()
    );
```

Local state backends

Persisted state bytes



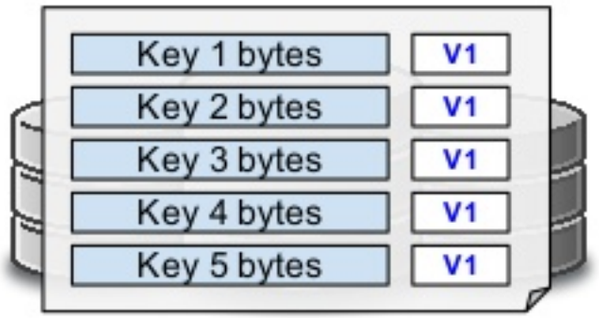
State serialization in heap-based backends

Flink job user code

```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV2()
    );
```

Local state backends

Persisted state bytes



010101110 …

Requires V1 serializer

 |

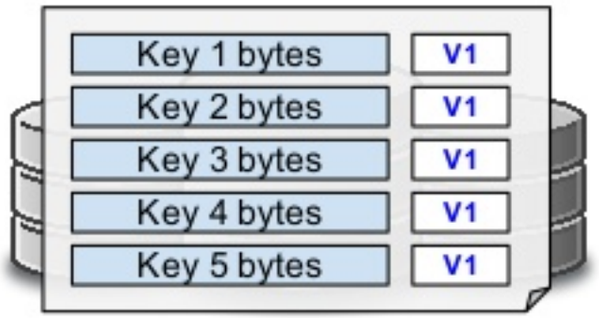
State serialization in heap-based backends

Flink job user code

```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV2()
    );
```

Local state backends

Persisted state bytes



 |

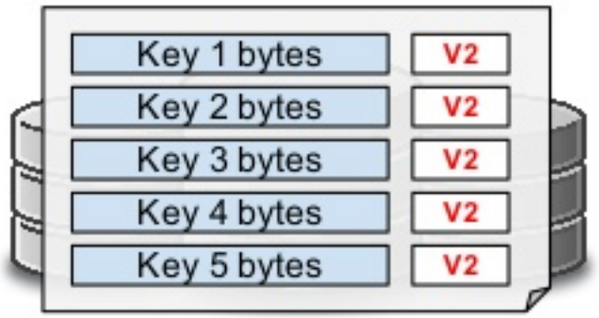
State serialization in heap-based backends

Flink job user code

```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV2()
    );
```

Local state backends

Persisted state bytes



Serialized by V2 serializer

010101110 …

Heap-based state backends

- *Lazy* serialization, *eager* deserialization
  - No state de-/serialization on access during runtime
  - Registered state serializer is only ever used on checkpoints

- By nature, the process of resuming from a savepoint + taking a new one is already a state migration process

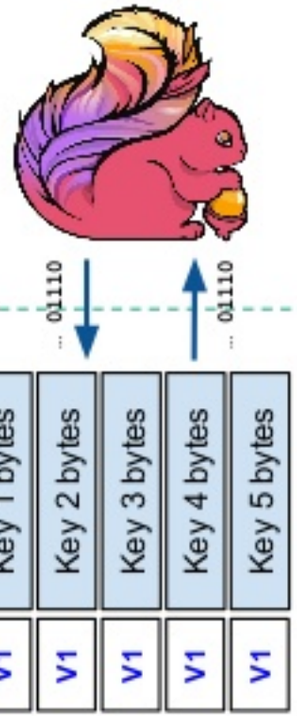
- Requirement: availability of previous state serializer at restore time
  - The new serializer is not yet available at restore time (since it is provided by user code)
  - Flink currently writes also the state serializer, using Java serialization, into savepoints as metadata
  - That serializer is deserialized from savepoints, and used to proceed with the restore

State serialization in off-heap backends

Flink job user code

Local state backends

Persisted state bytes




```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV1()
    );
```

 |

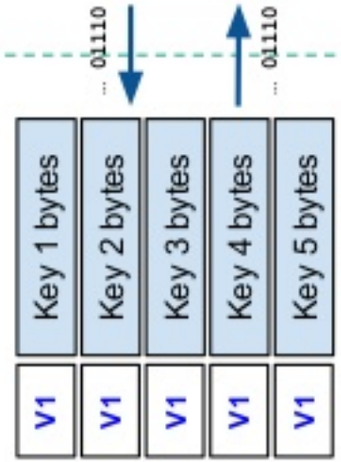
State serialization in off-heap backends

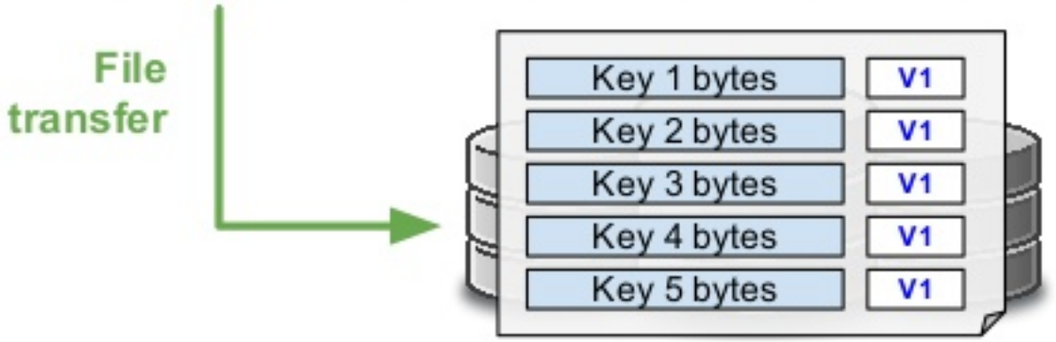
Flink job user code

Local state backends

Persisted state bytes

```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV1()
    );
```





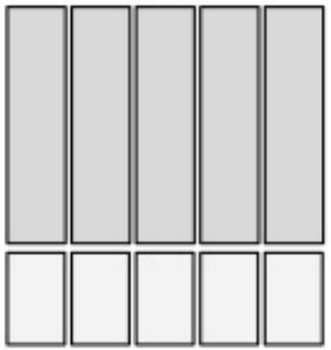
 |

State serialization in off-heap backends

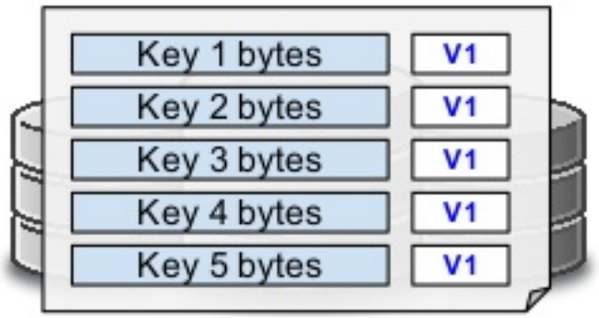
Flink job user code

```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV2()
    );
```

Local state backends



Persisted state bytes

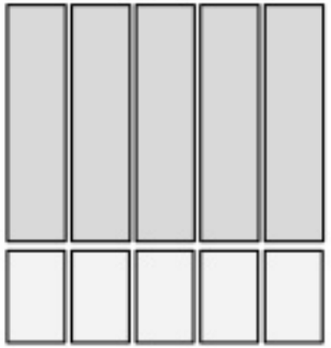


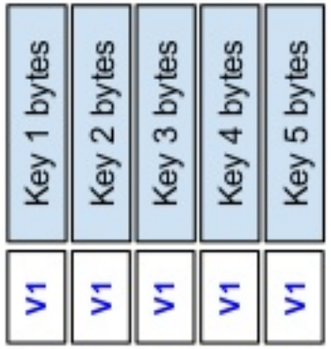
State serialization in off-heap backends

Flink job user code

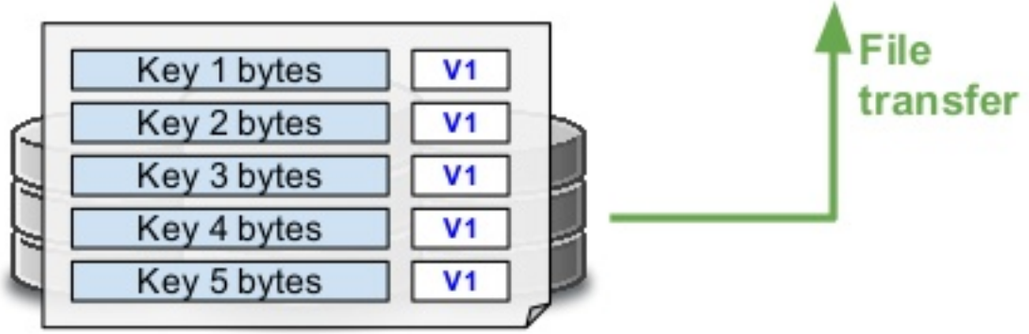
```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV2()
    );
```

Local state backends





Persisted state bytes

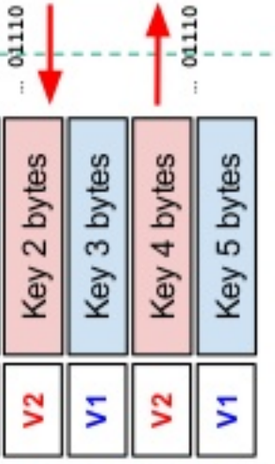


 |

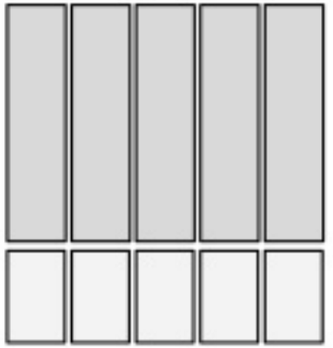
State serialization in off-heap backends

Flink job user code

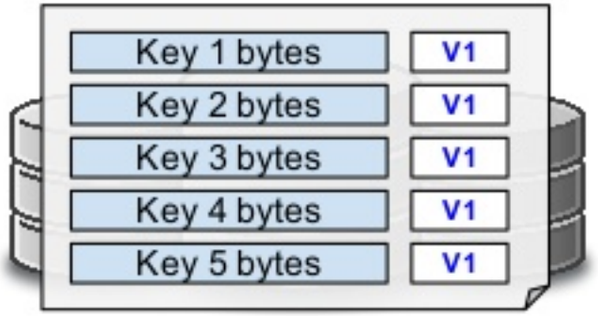
```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV2()
    );
```



Local state backends



Persisted state bytes

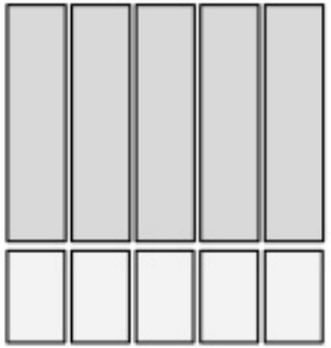


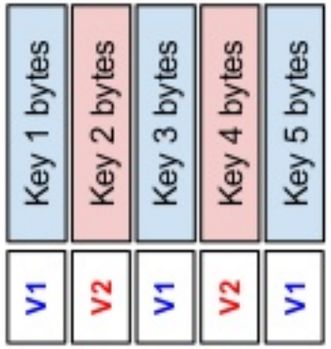
State serialization in off-heap backends

Flink job user code

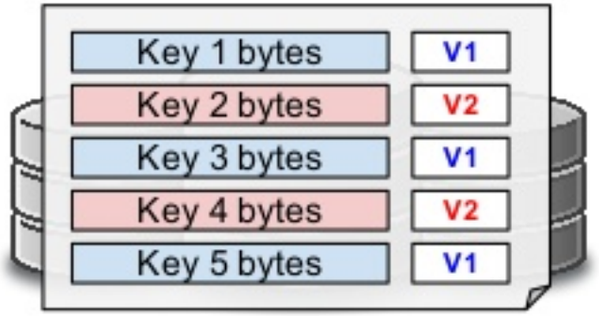
```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV2()
    );
```

Local state backends





Persisted state bytes



File transfer

Out-of-core state backends

- *Eager* serialization, *lazy* deserialization
  - State de-/serialization on every single access during runtime
  - Registered state serializer is only ever used when accessing state

- Requirement: new registered serializers need to be compatible with all previous schema versions
  - Since data will only be written with the new schema if it was accessed, it is possible that there will be multiple schema versions across different keys
  - Having a fully backwards compatible serializer ensures that the job can safely proceed

State serializer upgrade paths

Case #1: Modified state types, resulting in different Flink-generated serializers

```
ValueStateDescriptor<MyPojo> desc =
    new ValueStateDescriptor<>("my-value-state", MyPojo.class); // modified MyPojo type
```

Case #2: Modified custom serializer

```
ValueStateDescriptor<MyCustomType> desc =
    new ValueStateDescriptor<>("my-value-state", new UpgradedSerializer<MyCustomType>());
```

- The new upgraded serializer would either be compatible, or not compatible. If incompatible, state migration is required.
- *Disclaimer:* as of Flink 1.6, new upgraded serializers must be compatible, as state migration is not yet an available feature.

State serializer compatibility

- As of now, Flink's auto-generated serializers can always only write and read in a single binary format, at the same time
  - For example, adding a field to a POJO type will result in the generated `PojoSerializer` having a new binary format, and will not be able to read the old format
- Therefore, ***using custom state serializers*** is recommended when keeping ***state schema evolution*** in mind

 |

State serializer compatibility

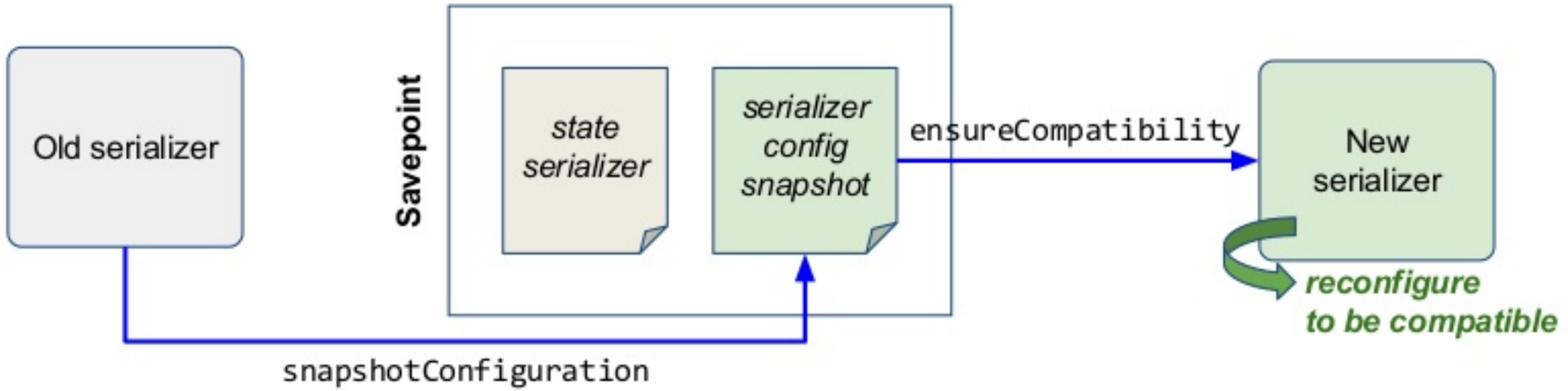
```
public abstract class TypeSerializer<T> {

    ...

     public abstract TypeSerializerConfigSnapshot snapshotConfiguration();

     public abstract CompatibilityResult<T> ensureCompatibility(
          TypeSerializerConfigSnapshot previousConfigSnapshot);

}
```



 |

Example TypeSerializer for backwards compatibility

```
public class MyCustomTypeSerializer extends TypeSerializer<MyCustomType> {

    ...

    public void serialize(DataOutputView target) {
        // write a version identifier
        target.writeInt(/** current version id **/);

        // always write in the latest binary format
    }

    public MyCustomType deserialize(DataInputView source) {
        // read the version identifier
        int versionId = source.readInt();

        switch (versionId) {
            /** deserialize path depends on version **/
        }
    }

    public TypeSerializerConfigSnapshot snapshotConfiguration() {...}
    public CompatibilityResult<T> ensureCompatibility(TypeSerializerConfigSnapshot configSnapshot) {...}
}
```

 |

Example TypeSerializer for backwards compatibility

```
public class MyCustomTypeSerializer extends TypeSerializer<MyCustomType> {

    ...

    public void serialize(DataOutputView target) {...}
    public MyCustomType deserialize(DataInputView source) {...}

    public TypeSerializerConfigSnapshot snapshotConfiguration() {
        // return a snapshot that contains information about
        //  1. the set of different versions that this serializer has handled
        //  2. information about how to handle each specific version
        //     (e.g. number of fields, field types etc., in other words the schema of each version)
    }

    public CompatibilityResult<MyCustomType> ensureCompatibility(TypeSerializerConfigSnapshot
configSnapshot) {
        // remember information about all schema versions the previous execution had handled,
        // to be used in the deserialize() method

        return CompatibilityResult.compatible();
    }
}
```

 |

What about using evolution-friendly frameworks, e.g. Avro?

- Short answer: theoretically, it works

 |

What about using evolution-friendly frameworks, e.g. Avro?

- Short answer: theoretically, it works

- Long answer: but there are some implications in Flink that needs to be solved first
  - The use of Java serialization to serialize state serializers is problematic.
  - For example, Flink's `AvroSerializer` holds a class instance of the Avro type, and that is serialized when writing the serializer.
  - Basically, this forbids any changes to the Avro type (e.g.., modifying your Avro schema)
  - See FLINK-9202

 |

Pending Improvements / WIPs

- Let Flink savepoints be completely free of Java serialization
  - This will introduce a change in the interfaces related to serializer compatibility
  - Allows Avro types and compatible evolution of Avro schema work out-of-the-box in Flink

- Introduce state migration procedures in state backends
  - A full-scan over state, for each state entry reading with previous serializer and rewriting with the new serializer
  - Allow for upgrading to incompatible serializer schemas

(Potential) New compatibility related interfaces

- See FLINK-9377

- TL;DR - Let snapshots of serializers (`TypeSerializerConfigSnapshot`) also double as a factory for instantiating the originating serializer of the snapshot
  - Eliminates the need to write serializers into savepoints
  - Availability of the restore serializer is determined at compile time
  - Overall smoothens the restore process of eagerly deserializing state backends
  - Guarantees that if state migration is really required, we always have a "copy" of the previous serializer available

(Potential) New compatibility related interfaces

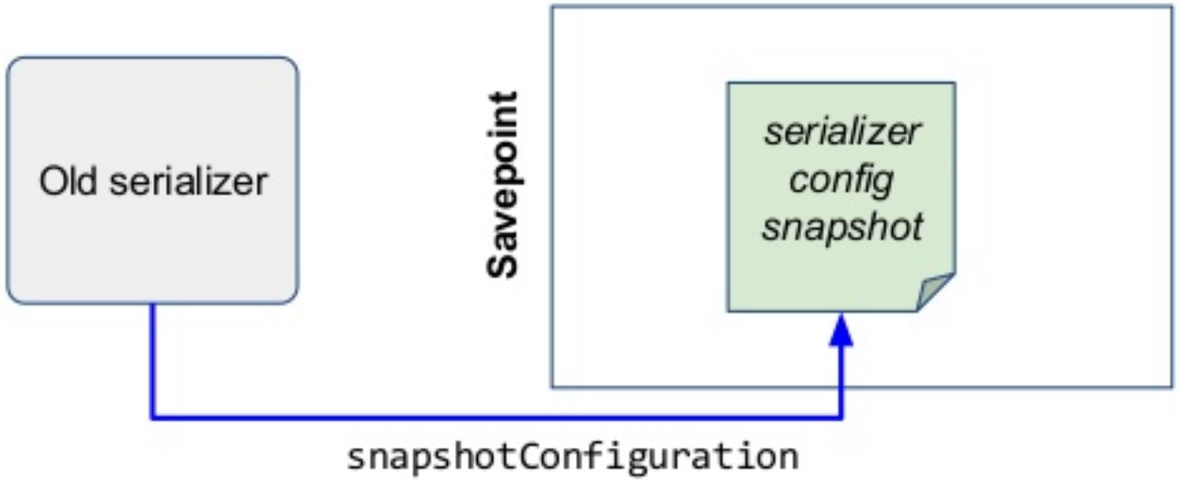
```
public abstract class TypeSerializer<T> {

      ...

      public abstract TypeSerializerConfigSnapshot<T> snapshotConfiguration();

      public abstract CompatibilityResult<T> ensureCompatibility(
            TypeSerializerConfigSnapshot previousConfigSnapshot);

}
```



 |

(Potential) New compatibility related interfaces

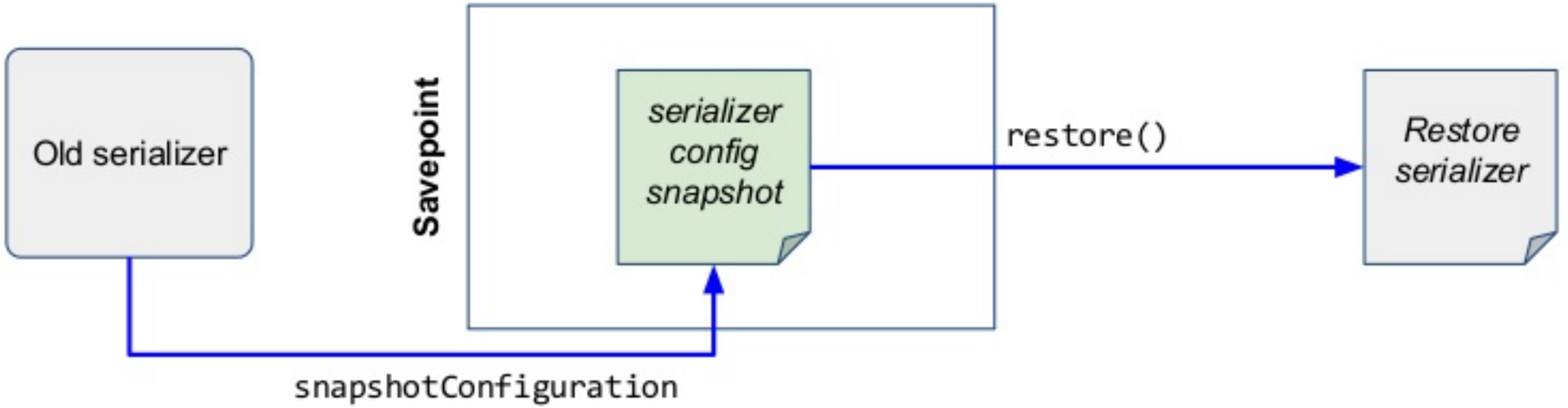
```
public abstract class TypeSerializer<T> {

      ...

      public abstract TypeSerializerConfigSnapshot<T> snapshotConfiguration();

      public abstract CompatibilityResult<T> ensureCompatibility(
            TypeSerializerConfigSnapshot previousConfigSnapshot);

}
```



 |

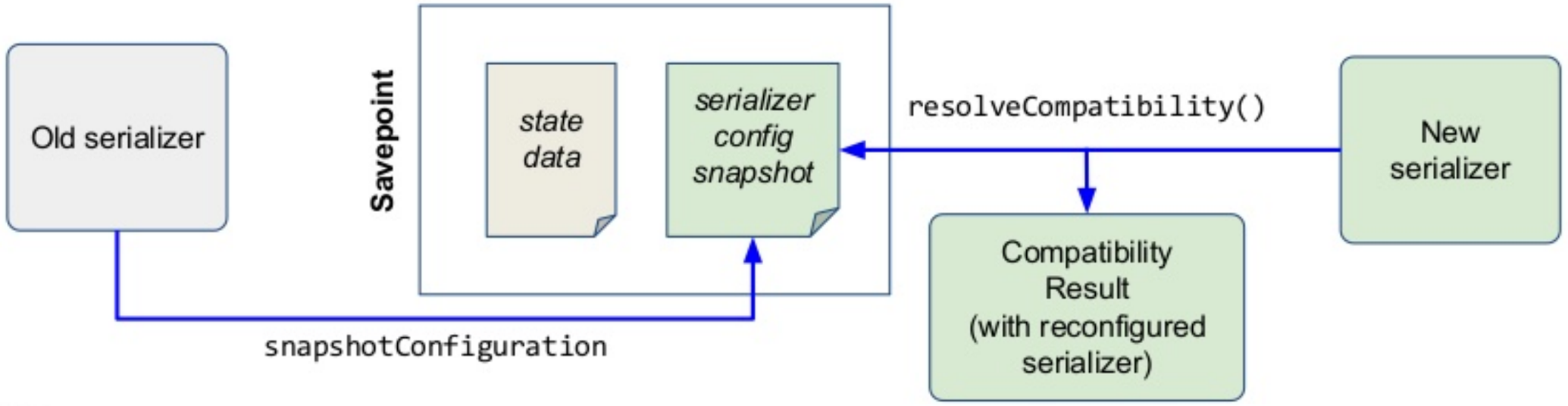
(Potential) New compatibility related interfaces

```
public abstract class TypeSerializerConfigSnapshot<T> {

      ...

      public abstract TypeSerializer<T> restore();

      public abstract CompatibilityResult<T> resolveCompatibility(
          TypeSerializer<T> newSerializer);
}
```



Savepoint Manipulation

 |

Savepoint Manipulation

- The ability to read, transform, and write to / create Flink savepoints outside of a streaming job

- Use cases -
  - Offline state migration
  - Re-calculating user-code induced errorneous states
  - Bootstrap new operators with state (e.g. from an external database)
  - Point-in-time state analytics, or even querying the state via Flink SQL
  - Change max parallelism of a job

Existing community effort - Bravo

- See the original proposal thread

- Brief description -
  - A convenient Flink savepoint reader / writer leveraging Flink's DataSet API
  - At the moment, only support RocksDBStateBackend savepoints
  - Should be useable for all backends once Flink unifies the savepoint format
  - There are plans to merge the functionality to core Flink for tighter integration

Bravo - reading and transforming keyed state

```
// First we start by taking a savepoint of our running job...
// Now it's time to load the metadata
Savepoint savepoint = StateMetadataUtils.loadSavepoint(savepointPath);

ExecutionEnvironment env = ExecutionEnvironment.getEnvironment();

// We create a KeyedStateReader for accessing the state of the operator CountPerKey
KeyedStateReader reader = new KeyedStateReader(savepoint, "CountPerKey", env);

// The reader now has access to all keyed states of the CountPerKey
// We are going to read one specific value state named Count
// The DataSet contains the key-state tuples from our state
DataSet<Tuple2<Integer, Integer>> countState = reader.readValueStates(
          ValueStateReader.forStateKVPairs("Count", new TypeHint<Tuple2<Integer, Integer>>() {}));

// We can now work with the countState dataset and analyze it however we want :)
```

 |

Bravo - reading and transforming keyed state

```
DataSet<Tuple2<Integer, Integer>> countState = ...

// We want to change our state based on some external data...
DataSet<Tuple2<Integer, Integer>> countsToAdd = environment.fromElements(
        Tuple2.of(0, 100), Tuple2.of(3, 1000),
        Tuple2.of(1, 100), Tuple2.of(2, 1000));

// These are the new count states we want to put back in the state
DataSet<Tuple2<Integer, Integer>> newCounts = countState
        .join(countsToAdd)
        .where(0)
        .equalTo(0)
        .map(new SumValues());
```


Bravo - reading and transforming keyed state

```
// We create a statetransformer that will store new checkpoint state under the newCheckpointDir base directory
StateTransformer stateBuilder = new StateTransformer(savepoint, newCheckpointDir);

// As a first step we have to serialize our Tuple K-V state with the provided utility
DataSet<KeyedStateRow> newStateRows = stateBuilder.createKeyedStateRows("CountPerKey", "Count", newCounts);

// In order not to lose the other value states in the "CountPerKey" operator we have to get the untouched rows from the reader
stateBuilder.replaceKeyedState("CountPerKey", newStateRows.union(reader.getUnparsedStateRows()));

// Last thing we do is create a new meta file that points to a valid savepoint
Path newSavepointPath = stateBuilder.writeSavepointMetadata();
```


Upgradability Dry-Runs

Upgradability Dry-Runs

- The ability to detect incompatible upgrades offline, without having to launch a full-blown job and only then discovering incompatibilities

- Common incompatibilities -
  - Incompatible topology changes
  - Incompatible state schema changes

Upgradability Dry-Runs

- Possible approach: comparing information in logical `StreamGraph`s of pipelines to detect incompatibilities
  - Topology information already included in `StreamGraph`
  - What about information about registered states?

- Information about registered states is currently only visible in state backends
  - Due to the nature of how the state declaration API works

 |

Eager State Declaration

```
public class MyMapFunction extends MapFunction<IN, OUT> {

      @KeyedState
      private final ValueState<MyStateType> valueState = StateHandleBuilder
            .valueState("state-id", new MyStateSerializer())
            .asQueryableState("queryable-state-id");

      @OperatorState( redistributionScheme = RedistributionScheme.UNION )
      private final ListState<MyStateType> unionOperatorState = StateHandleBuilder
            .listState("state-id-2", new MyStateSerializer())

      public OUT map(IN input) {
            ...
            MyStateType v = valueState.get();

      }
}
```

 |

Wrap up

 |

TL;DR

- The ability to take savepoints, change user code, and reconfigure jobs have proven to be a solid baseline for users to manage upgrading their streaming applications

- Continuing to smoothen / enable more possibilities in the upgrade process is an utmost priority in the Flink community
  - Enabling state schema migration currently being the most important
  - Reading from / transforming / writing to / creating savepoints is also a highly desired feature

- Would love to hear more about your pain points when it comes to upgrading streaming applications!

THANK YOU!

@tzulitai

@dataArtisans

@ApacheFlink

dataArtisans