Managing State in Apache Flink

Tzu-Li (Gordon) Tai
Flink Committer / PMC Member

dataArtisans

tzulitai@apache.org
@tzulitai

dataArtisans

Original creators of **Apache Flink®**

dA

PLATFORM

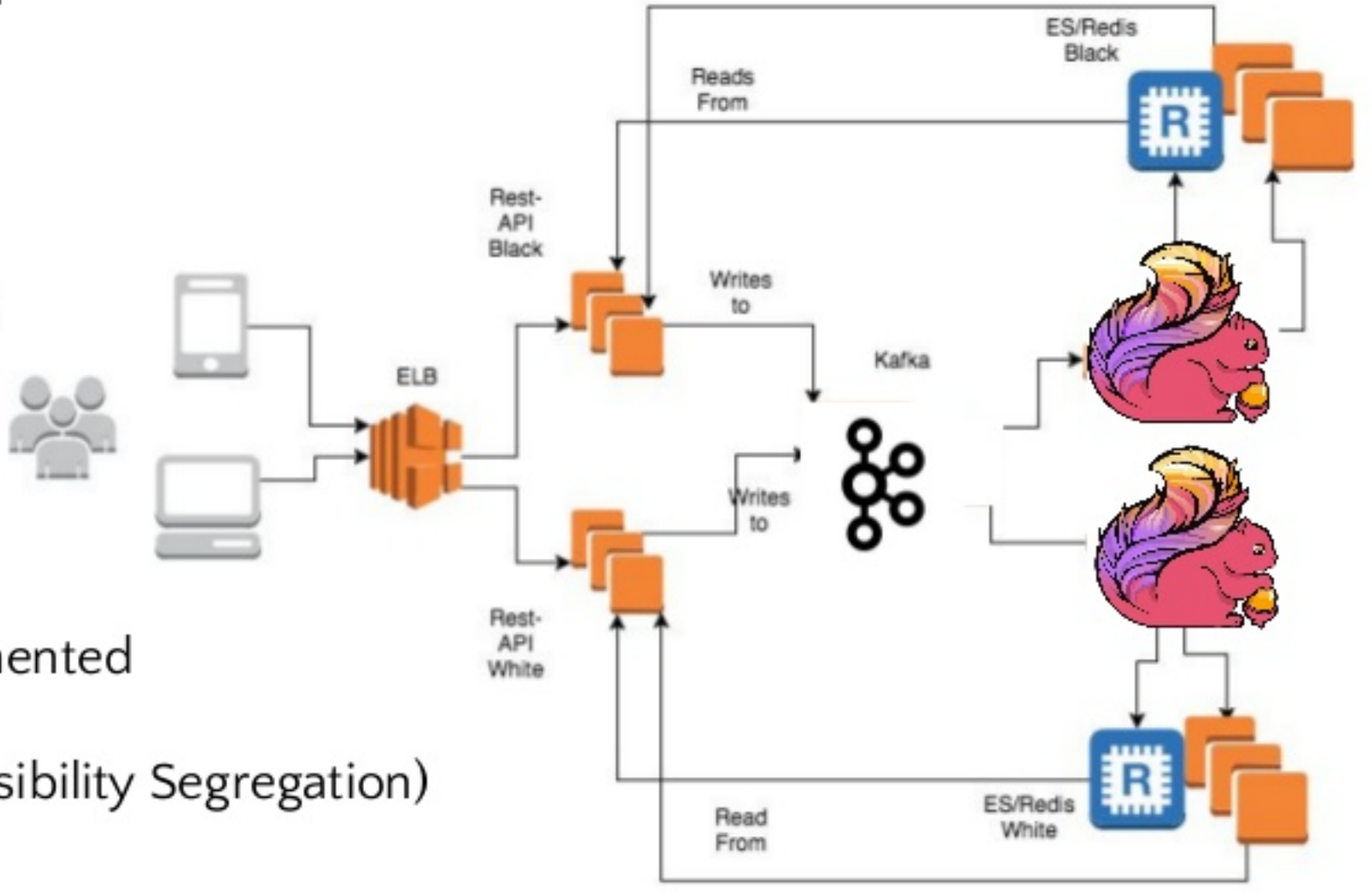
Providers of
dA Platform 2, including
open source Apache Flink +
dA Application Manager

Users are placing more and more of their most valuable assets within Flink: their application data

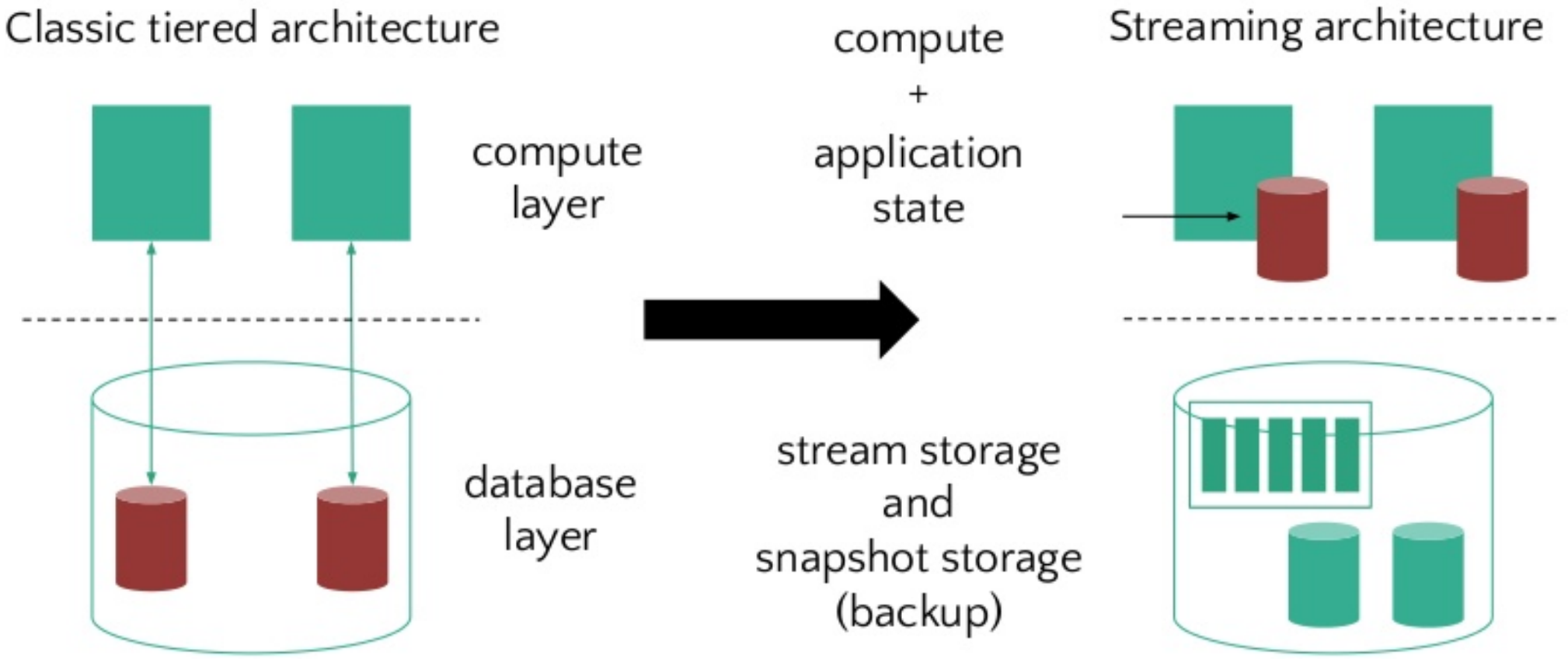
@

D_TRB

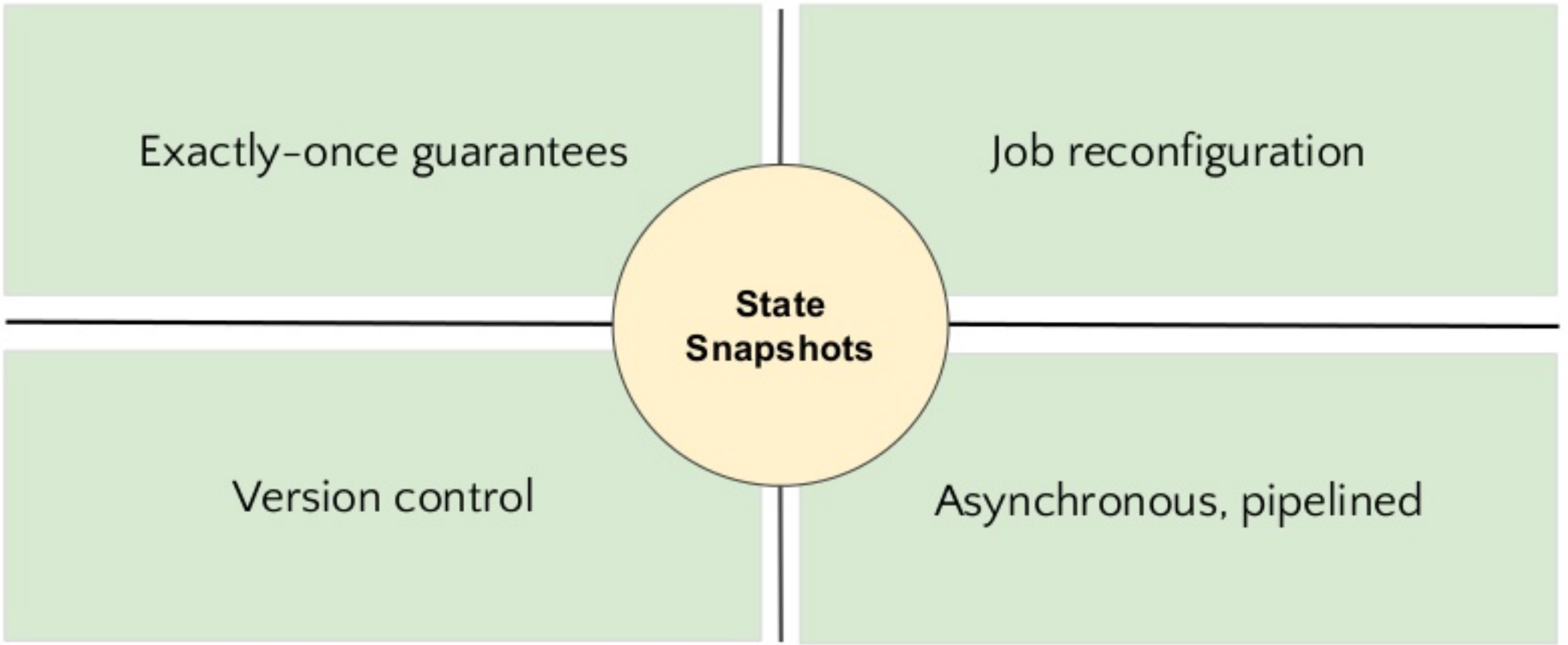
THE SOCIAL NETWORK FOR PETROLHEADS



Complete social network implemented
using event sourcing and
CQRS (Command Query Responsibility Segregation)


Classic tiered architecture
compute layer
database layer
compute + application state
stream storage and snapshot storage (backup)
Streaming architecture

Streams and Snapshots



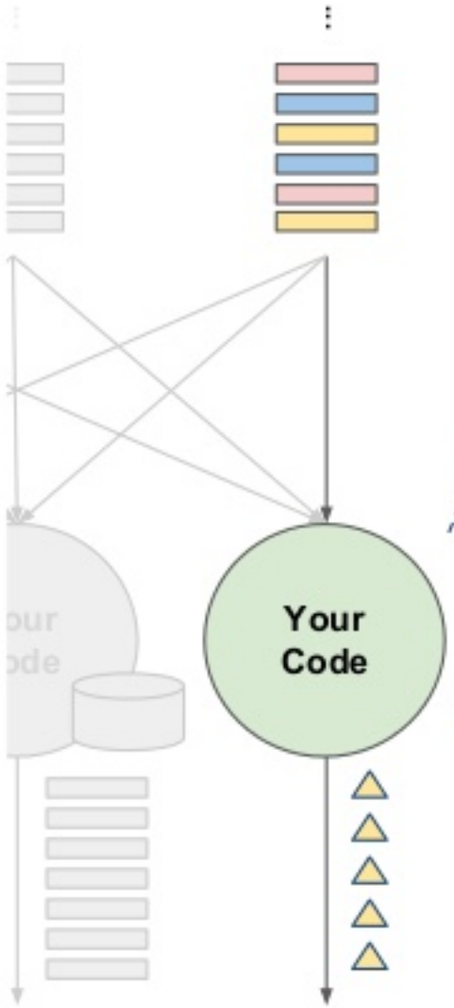
… want to ensure that users can fully entrust Flink with their state, as well as good quality of life with state management

State management FAQs ...

- State declaration best practices?
- How is my state serialized and persisted?
- Can I adapt how my state is serialized?
- Can I adapt the schema / data model of my state?

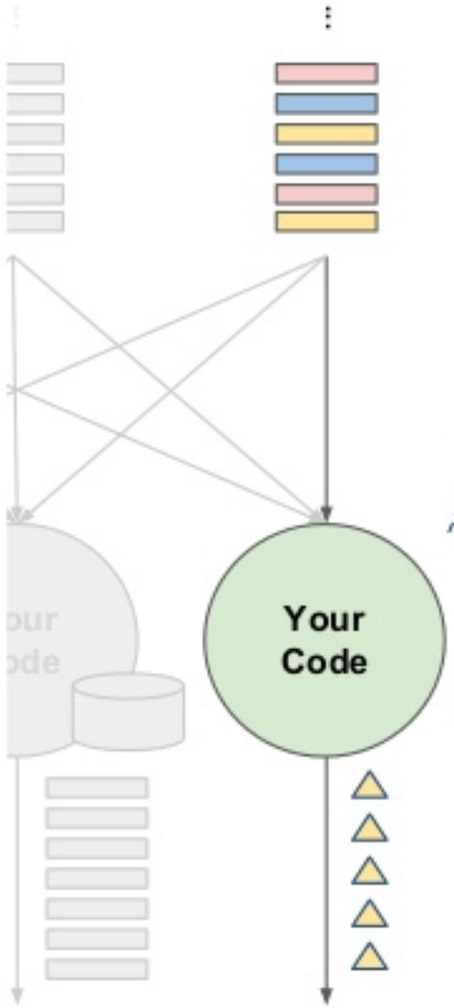
Recap: Flink's Managed State

Flink Managed State




```
ValueStateDescriptor<MyPojo> desc =
    new ValueStateDescriptor<>(
        “my-value-state”,
        MyPojo.class
    );

ValueState<MyPojo> state =
    getRuntimeContext().getState(desc);


MyPojo p = state.value();
...

state.update(...)
...
```

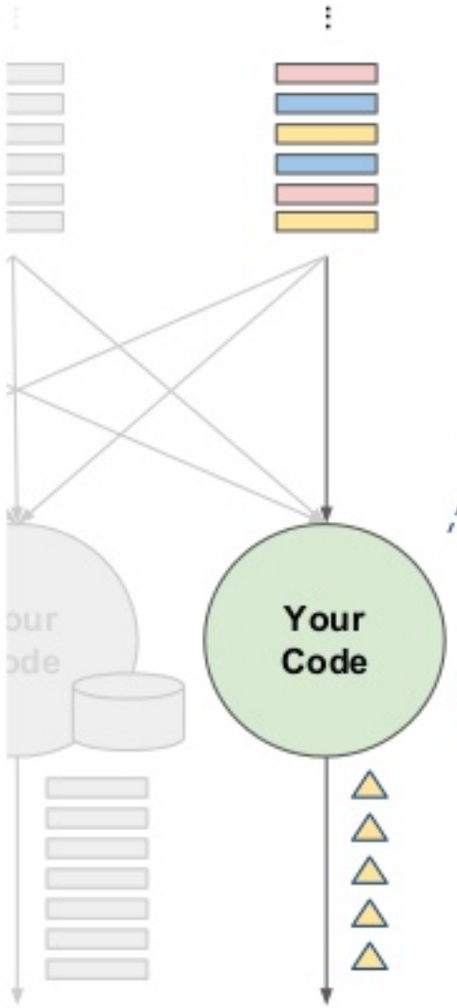
Flink Managed State




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

ValueState<MyPojo> state =
    getRuntimeContext().getState(desc);


MyPojo p = state.value();
...

state.update(...)
...
```

unique identifier for this managed state

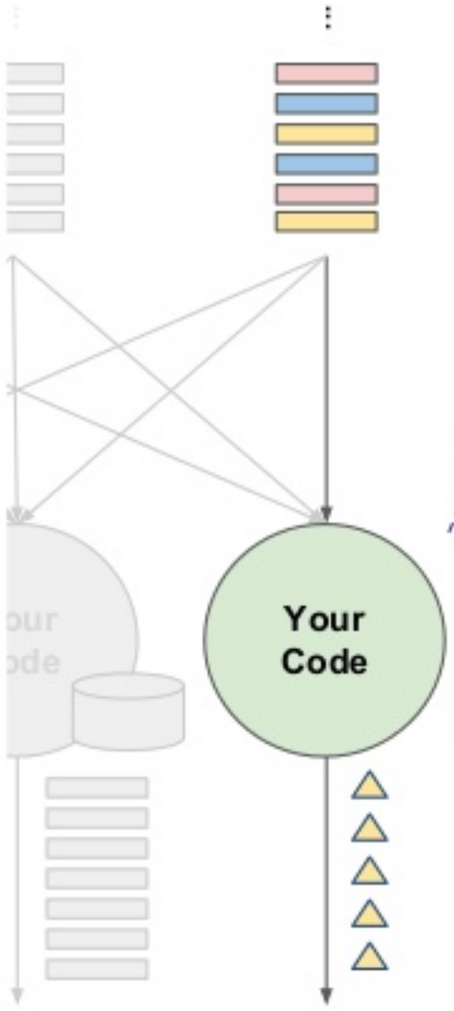
Flink Managed State




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

ValueState<MyPojo> state =
    getRuntimeContext().getState(desc);


MyPojo p = state.value();
...

state.update(...)
...
```

type information for the state

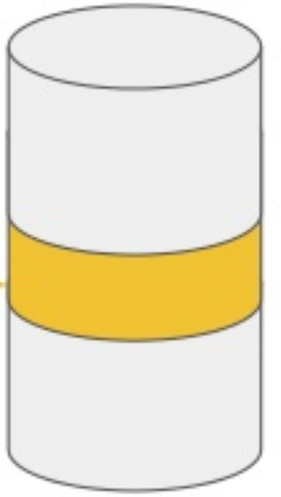
Flink Managed State



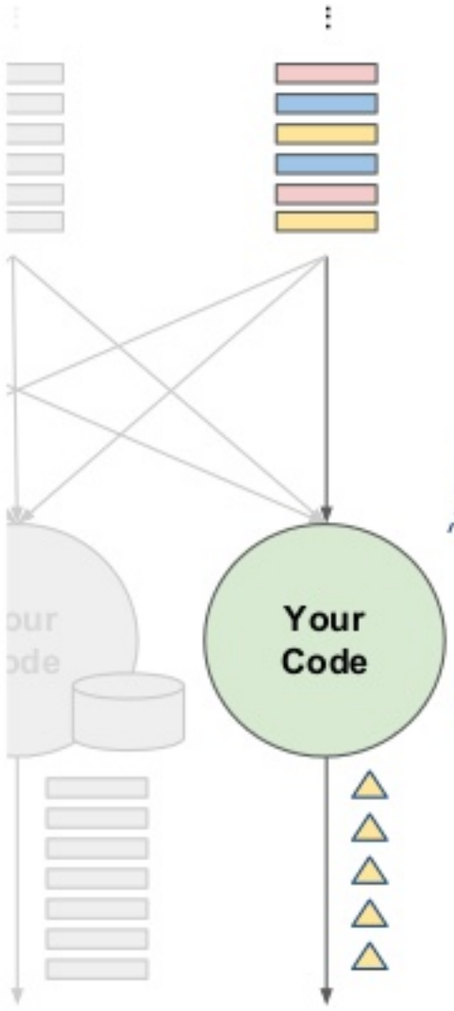

```
ValueStateDescriptor<MyPojo> desc =
    new ValueStateDescriptor<>(
        “my-value-state”,
        MyPojo.class
    );

ValueState<MyPojo> state =
  getRuntimeContext().getState(desc);


MyPojo p = state.value();
...

state.update(...)
...
```

registers state

Local
State Backend



Flink Managed State

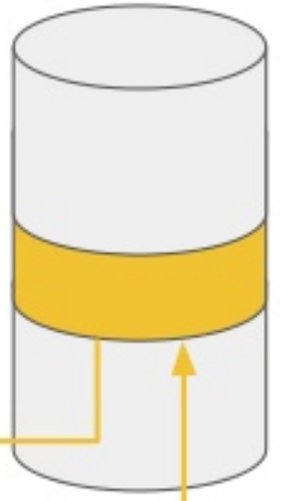



```
ValueStateDescriptor<MyPojo> desc =
    new ValueStateDescriptor<>(
        “my-value-state”,
        MyPojo.class
    );

ValueState<MyPojo> state =
    getRuntimeContext().getState(desc);

MyPojo p = state.value();
...

state.update(...)
...
```

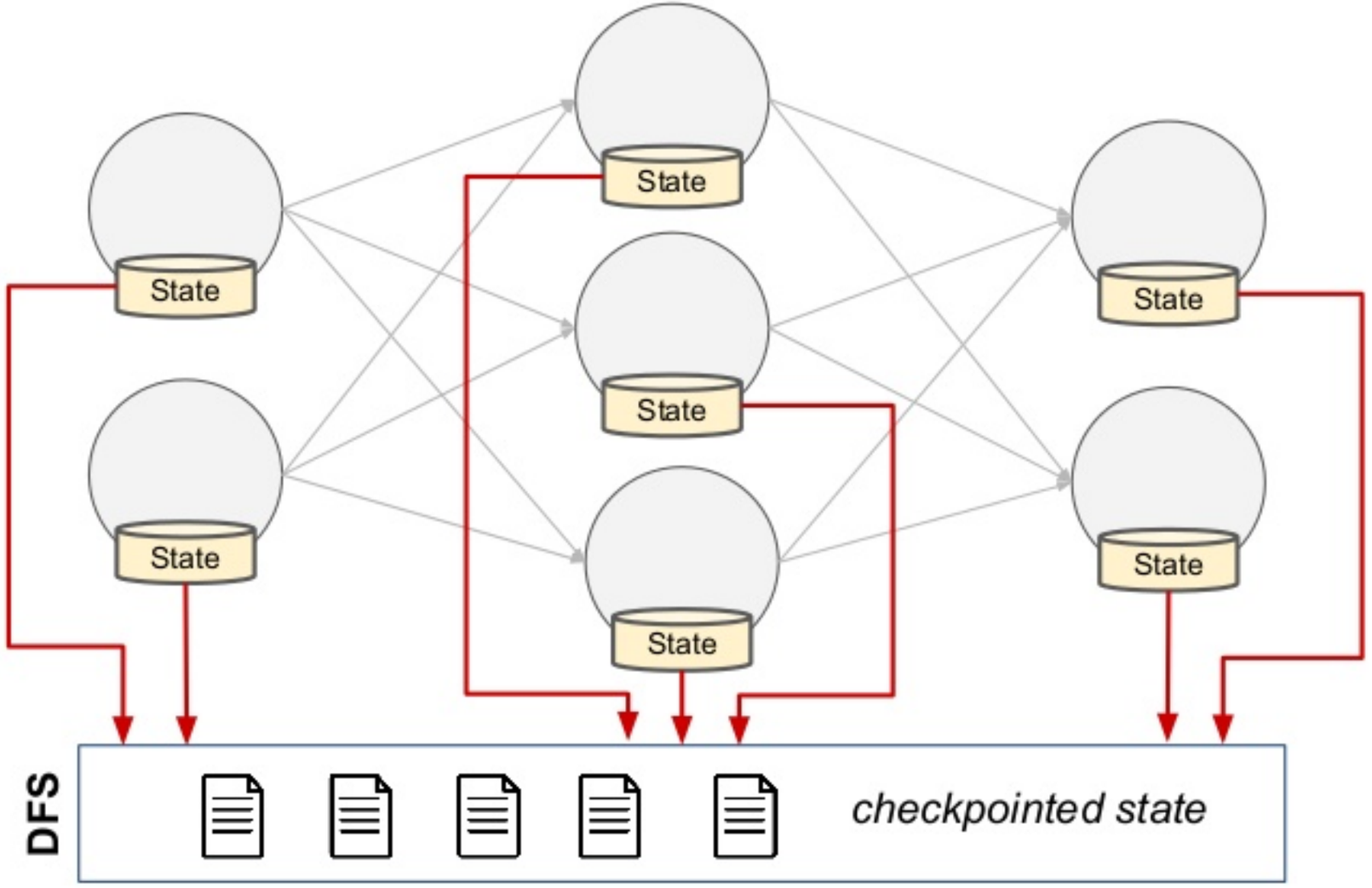
read

write

Local
State Backend



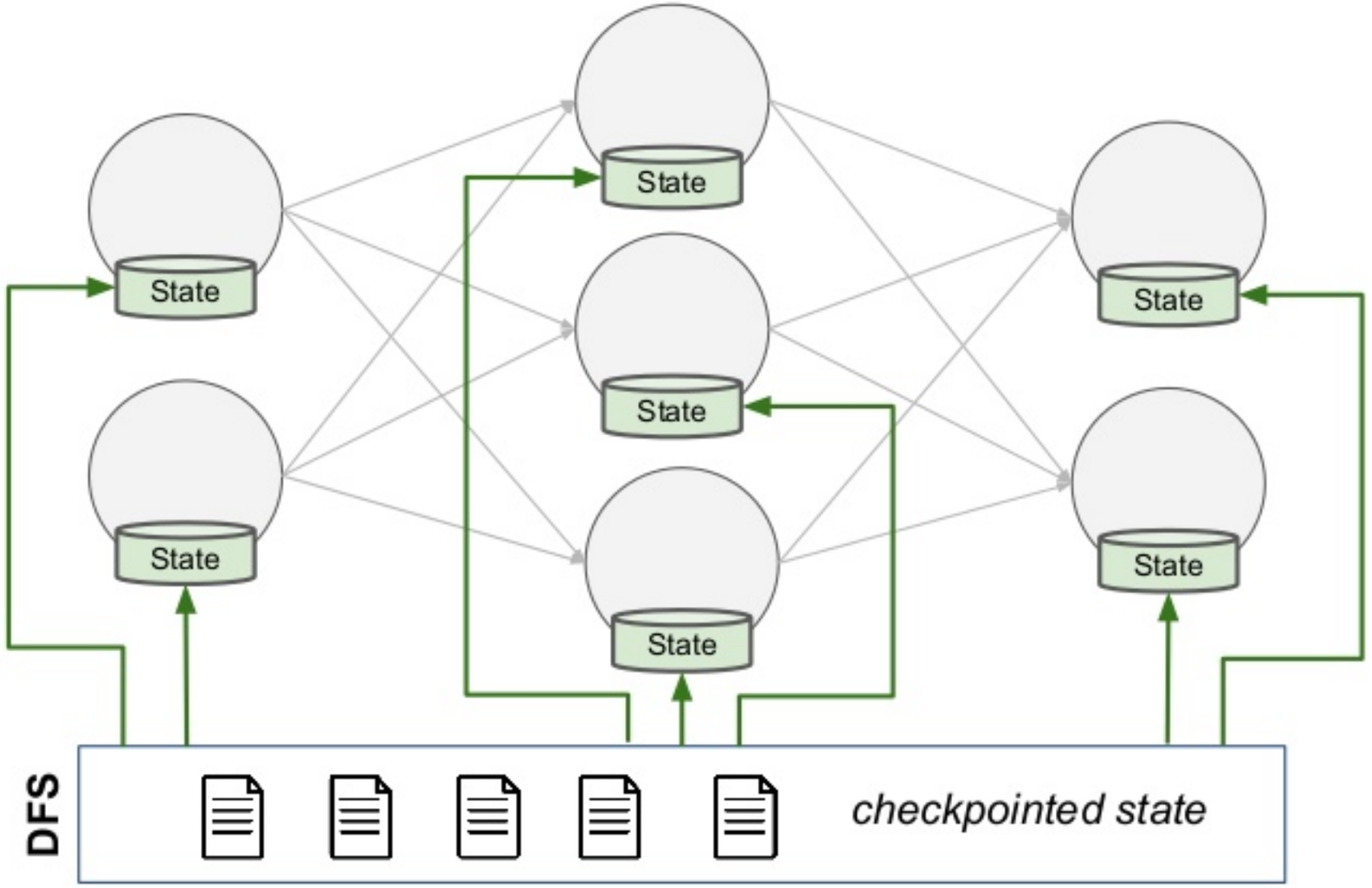
Flink Managed State (II)



Checkpoints

- Pipelined checkpoint barriers flow through the topology
- Operators asynchronously backup their state in checkpoints on a DFS

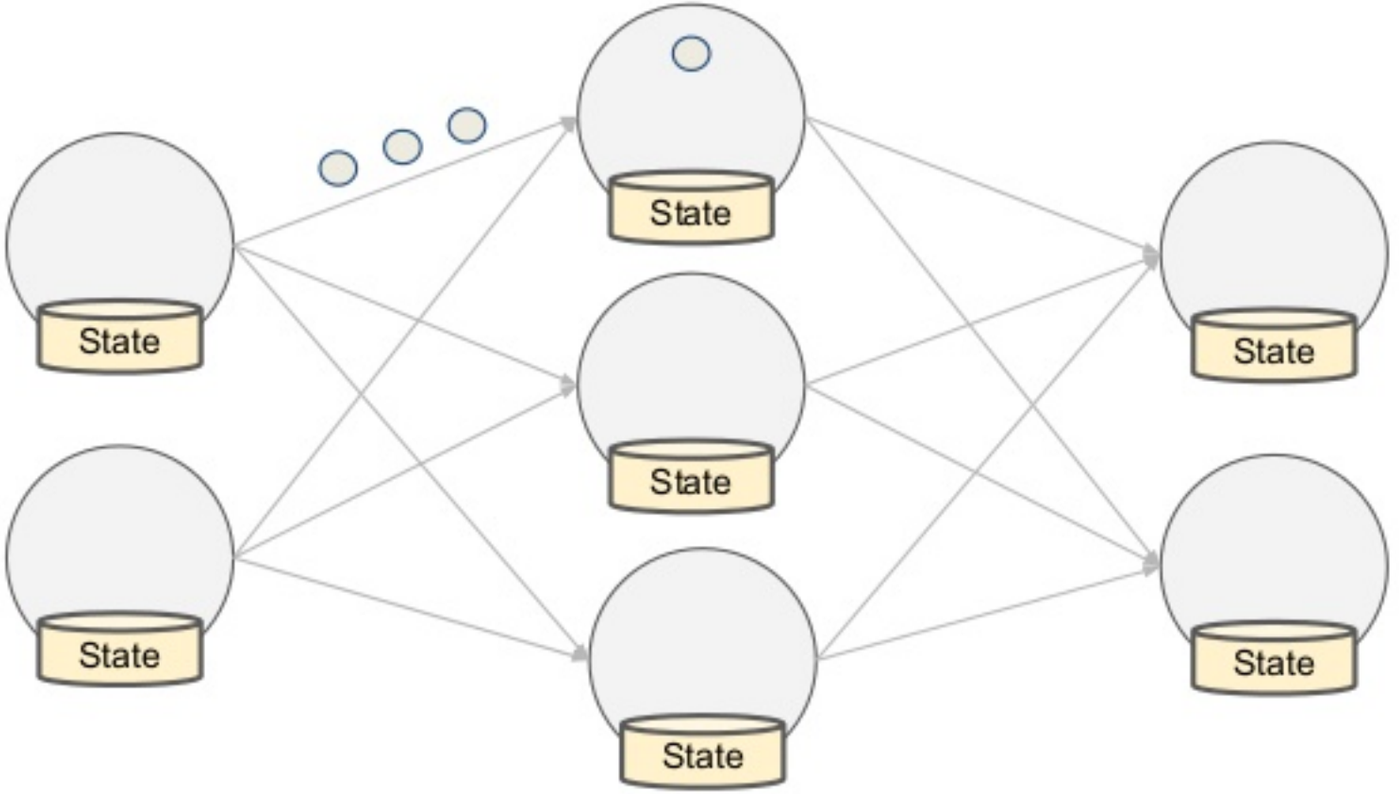
Flink Managed State (II)

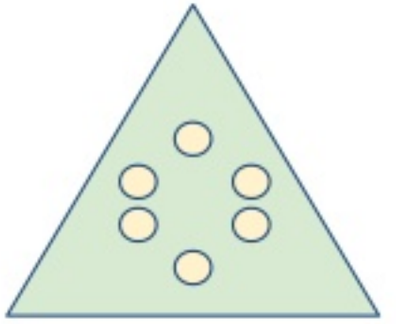


Restore

- Each operator is assigned their corresponding file handles, from which they restore their state

State Serialization Behaviours

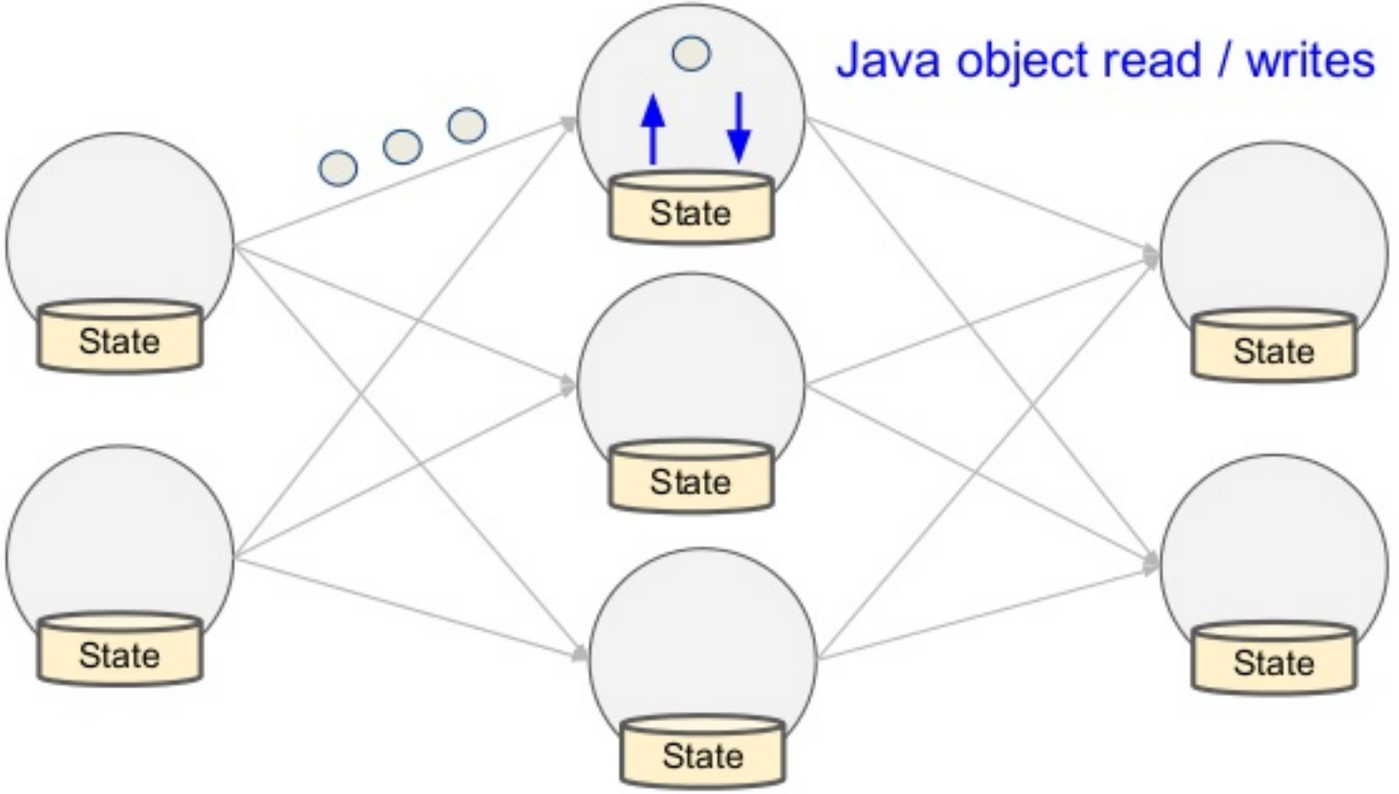


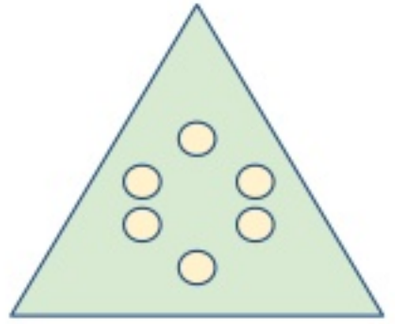


JVM Heap backed state backends (`MemoryStateBackend`, `FsStateBackend`)

⇒ ***lazy*** *serialization +* ***eager*** *deserialization*

State Serialization Behaviours

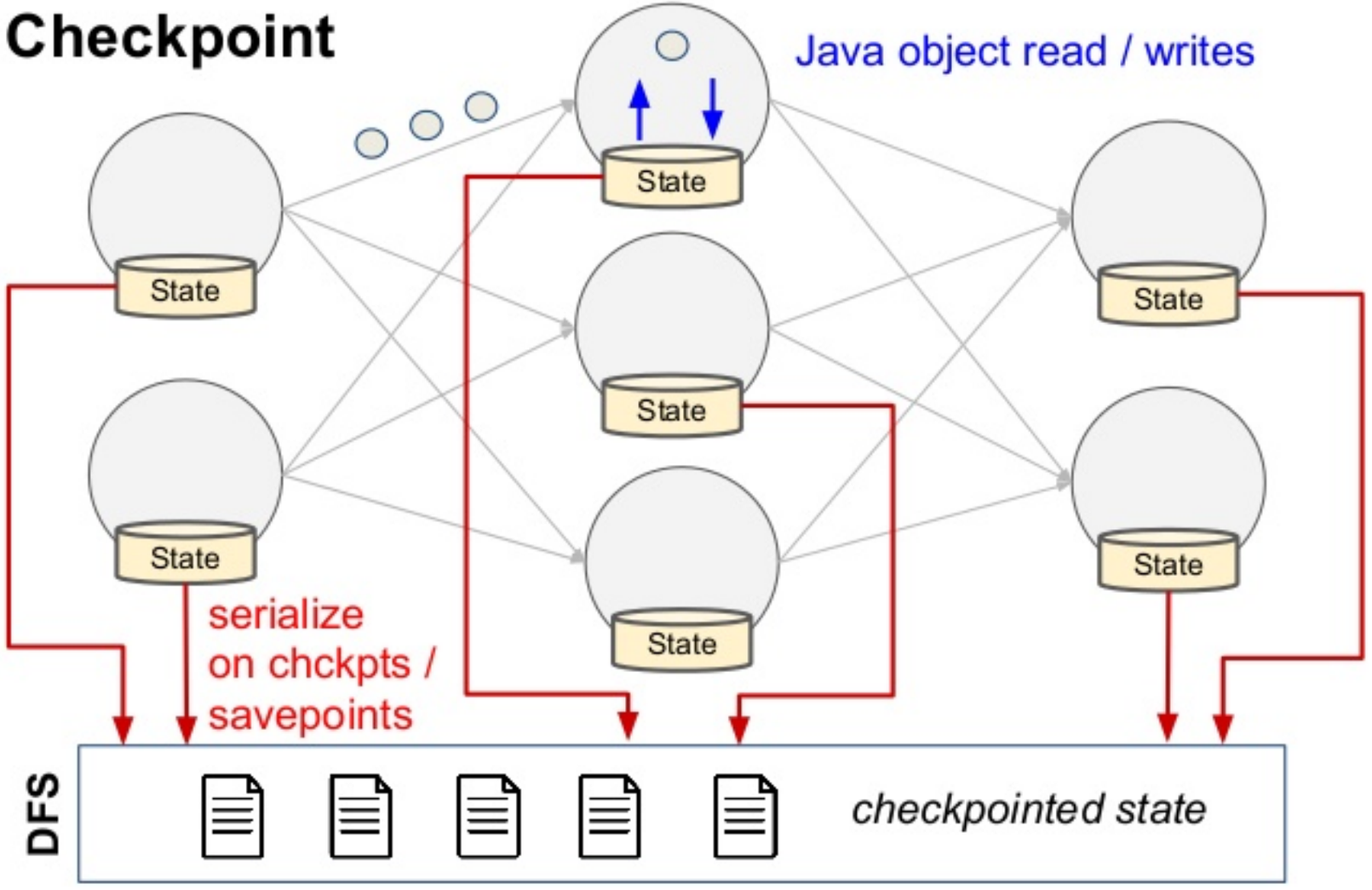


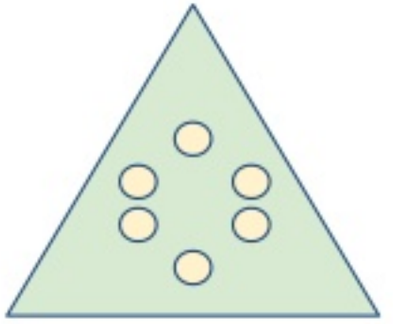


JVM Heap backed state backends (`MemoryStateBackend`, `FsStateBackend`)

⇒ ***lazy*** *serialization +* ***eager*** *deserialization*

State Serialization Behaviours

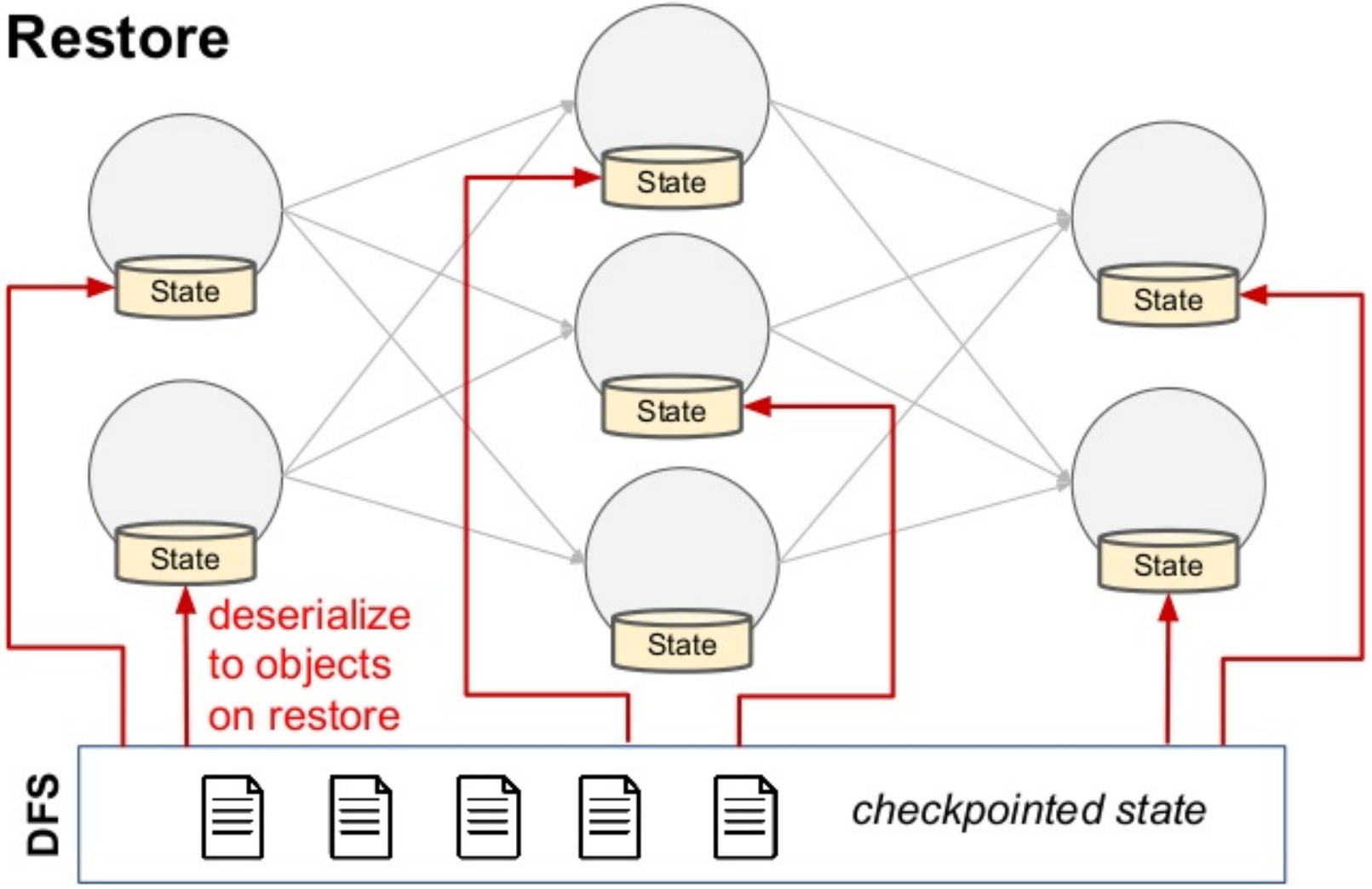


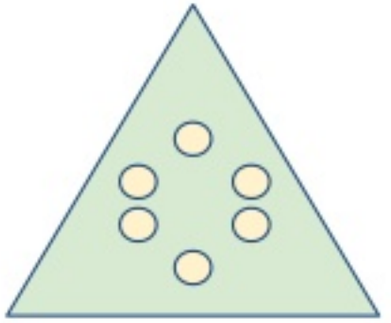


JVM Heap backed state backends (`MemoryStateBackend`, `FsStateBackend`)

⇒ ***lazy*** *serialization +* ***eager*** *deserialization*

State Serialization Behaviours

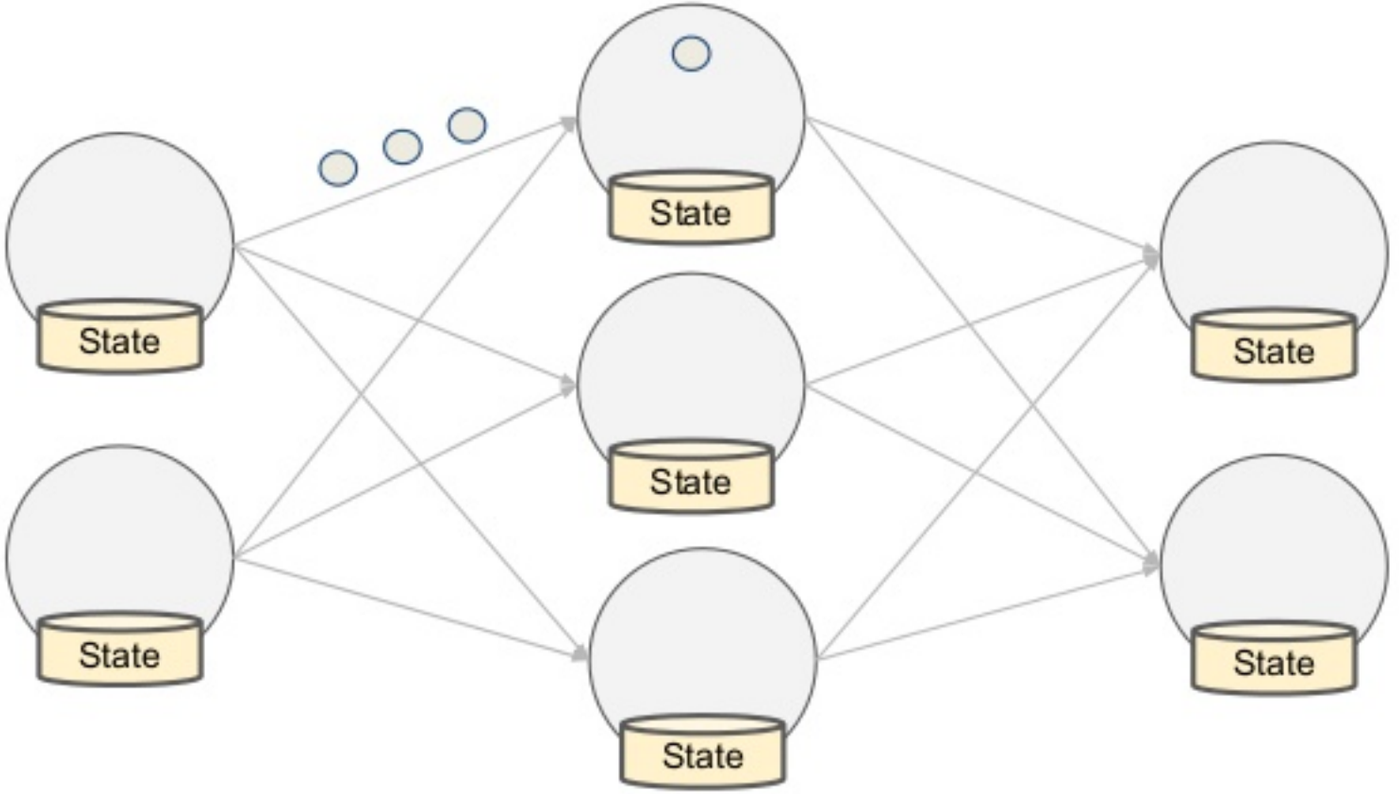




JVM Heap backed state backends (`MemoryStateBackend`, `FsStateBackend`)

⇒ ***lazy*** *serialization +* ***eager*** *deserialization*

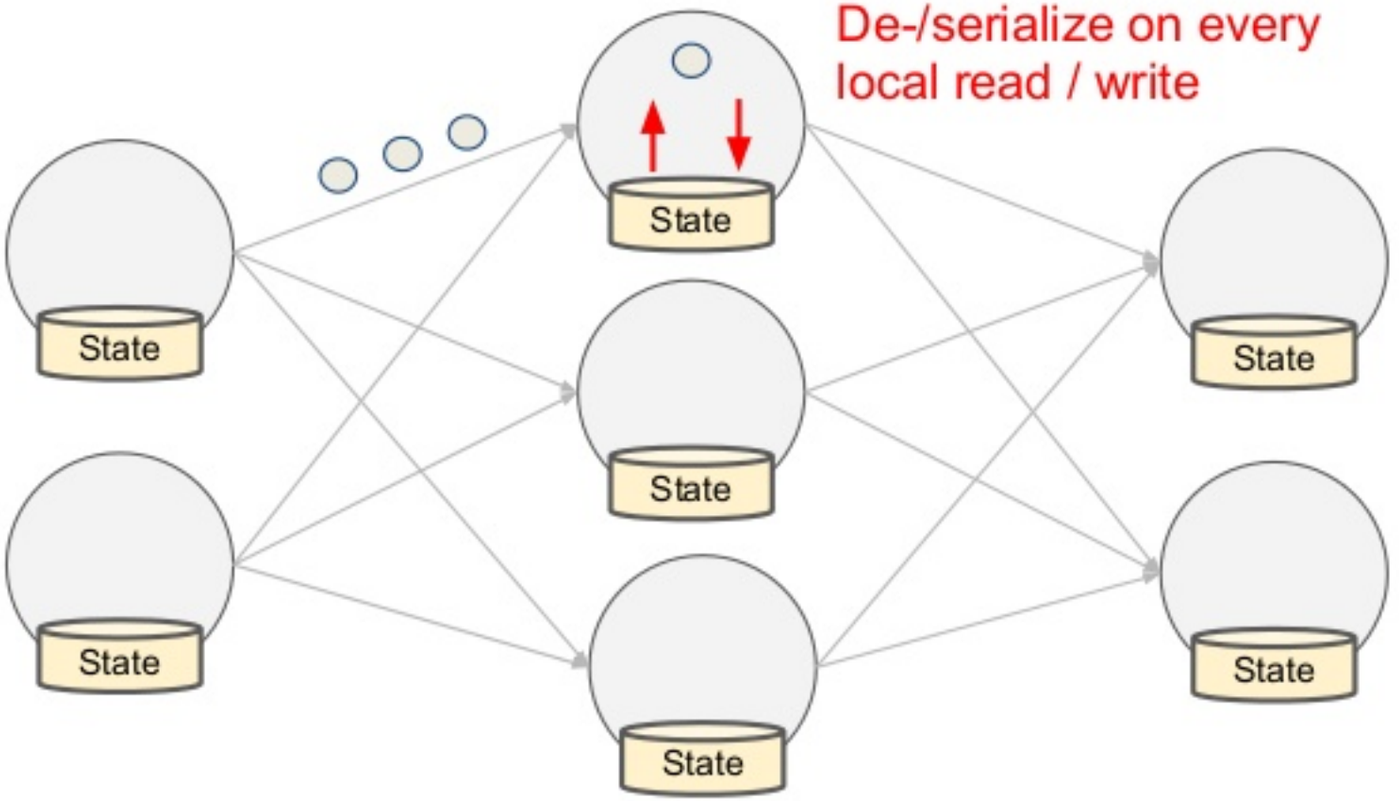
State Serialization Behaviours



Out-of-core state backends
(`RocksDBStateBackend`)

⇒ ***eager*** *serialization +*
lazy *deserialization*

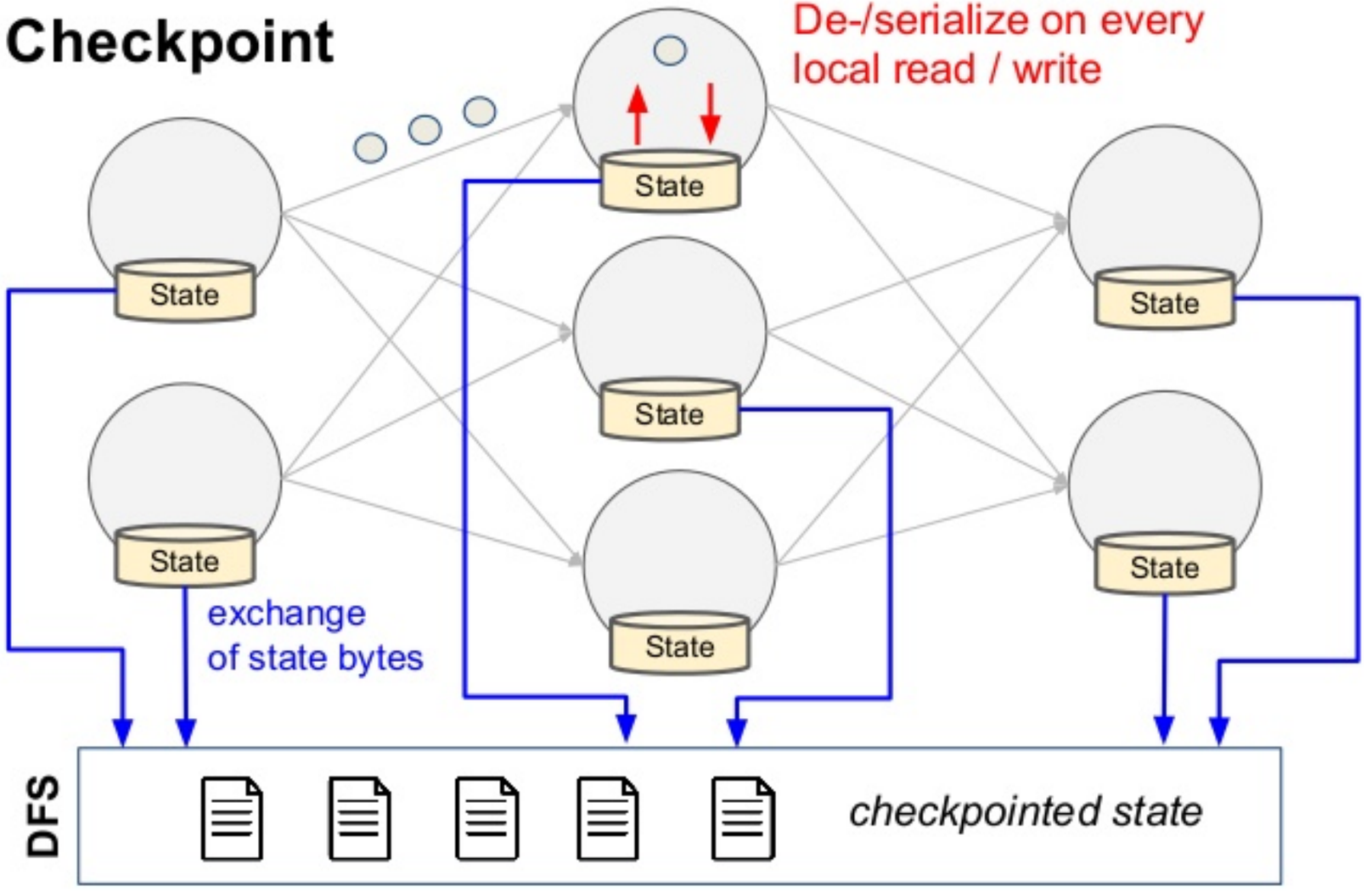
State Serialization Behaviours



Out-of-core state backends
(`RocksDBStateBackend`)

⇒ ***eager*** *serialization +*
lazy *deserialization*

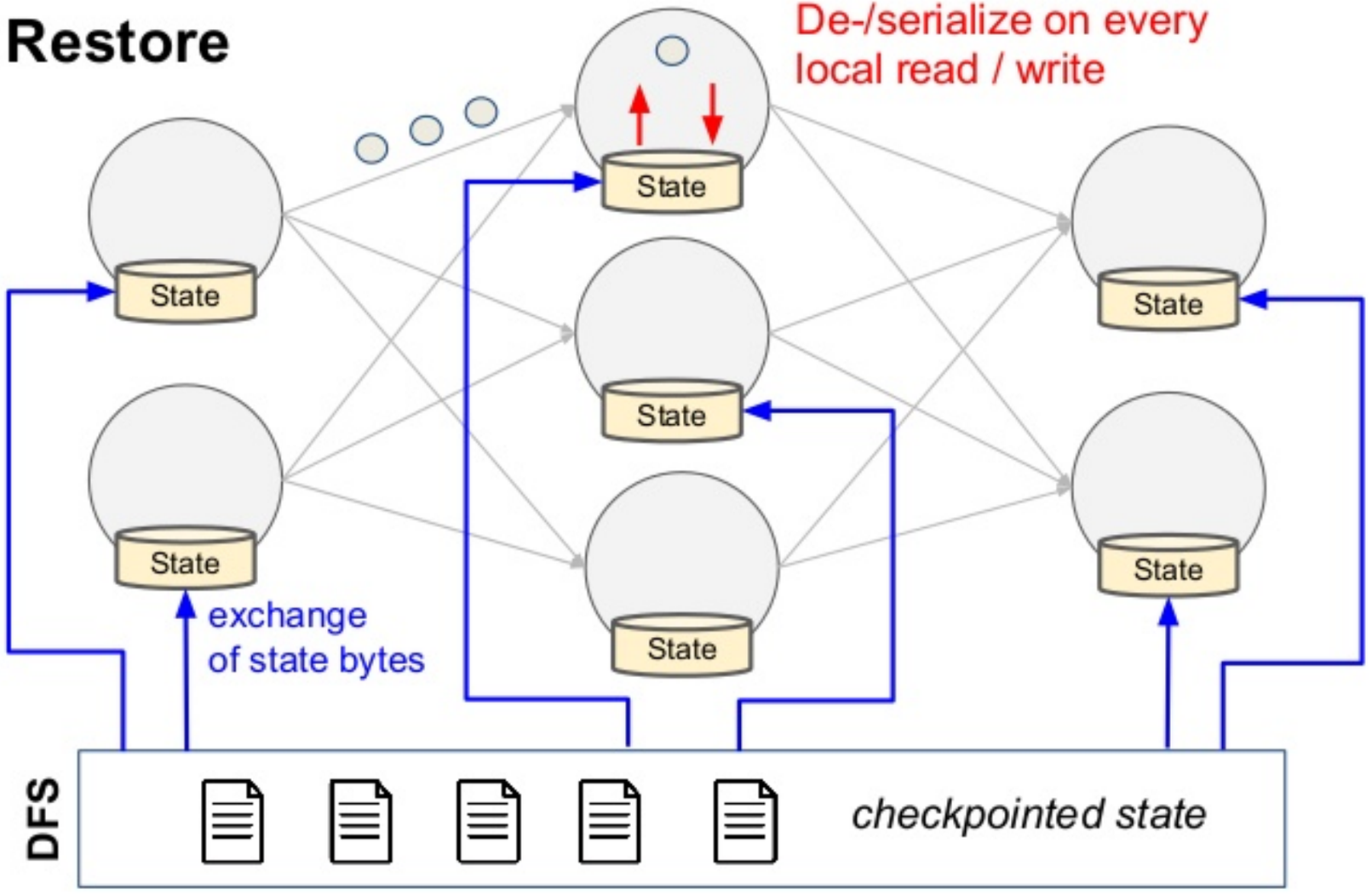
State Serialization Behaviours



Out-of-core state backends
(`RocksDBStateBackend`)

⇒ ***eager*** *serialization +*
lazy *deserialization*

State Serialization Behaviours



Out-of-core state backends
(`RocksDBStateBackend`)

⇒ ***eager*** *serialization +*
lazy *deserialization*

Matters of State (I)

State Declaration

State Access Optimizations

- Flink supports different state “structures” for efficient state access for various patterns

```
ValueStateDescriptor<Map<String, MyPojo>> desc =
    new ValueStateDescriptor<>(“my-value-state”, MyPojo.class);


ValueState<Map<String, MyPojo>> state =
    getRuntimeContext().getState(desc);


Map<String, MyPojo> map = state.value();
map.put(“someKey”, new MyPojo(...));
state.update(map);
```

X don’t do

State Access Optimizations

- Flink supports different state “structures” for efficient state access for various patterns

```
MapStateDescriptor<String, MyPojo> desc =
    new MapStateDescriptor<>(“my-value-state”, String.class, MyPojo.class);

MapState<String, MyPojo> state =
    getRuntimeContext().getMapState(desc);

MyPojo pojoVal = state.get(“someKey”);
state.put(“someKey”, new MyPojo(...));
```

optimized for random key access

Declaration Timeliness

- Try registering state as soon as possible
  - Typically, all state declaration can be done in the “open()” lifecycle of operators

Declaration Timeliness

```
public class MyStatefulMapFunction extends RichMapFunction<String, String> {

    private static ValueStateDescriptor<MyPojo> DESC =
        new ValueStateDescriptor<>("my-pojo-state", MyPojo.class);

    @Override
    public String map(String input) {
        MyPojo pojo = getRuntimeContext().getState(DESC).value();
        …
    }
}
```

Declaration Timeliness

```
public class MyStatefulMapFunction extends RichMapFunction<String, String> {

    private static ValueStateDescriptor<MyPojo> DESC =
        new ValueStateDescriptor<>("my-pojo-state", MyPojo.class);

    private ValueState<MyPojo> state;

    @Override
    public void open(Configuration config) {
        state = getRuntimeContext().getState(DESC);
    }

    @Override
    public String map(String input) {
        MyPojo pojo = state.value();
        …
    }
}
```

Eager State Declaration (under discussion, release TBD)

- Under discussion with FLIP-22
- Allows the JobManager to have knowledge on declared states of a job

Eager State Declaration

(under discussion, release TBD)

```
public class MyStatefulMapFunction extends RichMapFunction<String, String> {

    @KeyedState(
        stateId = “my-pojo-state”,
        queryableStateName = “state-query-handle”
    )
    private MapState<String, MyPojo> state;

    @Override
    public String map(String input) {
        MyPojo pojo = state.get(“someKey”);
        state.put(“someKey”, new MyPojo(...));
    }

}
```

Matters of State (II)

State Serialization

How is my state serialized?

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

- Provided state type class info is analyzed by Flink's own serialization stack, producing a tailored, efficient serializer
- Supported types:
  - Primitive types
  - Tuples, Scala Case Classes
  - POJOs (plain old Java objects)
- All non-supported types fallback to using Kryo for serialization

Avoid Kryo for State Serde

- Kryo is generally not recommended for use on persisted data
  - Unstable binary formats
  - Unfriendly for data model changes to the state
- Serialization frameworks with schema evolution support is recommended: *Avro, Thrift, etc.*
- Register custom Kryo serializers for your Flink job that uses these frameworks

Avoid Kryo for State Serde

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

// register the serializer included with
// Apache Thrift as the standard serializer for your type
env.registerTypeWithKryoSerializer(MyCustomType, TBaseSerializer.class);
```

- Also see: https://goo.gl/oU9NxJ

Custom State Serialization

- Instead of supplying type information to be analyzed, directly provide a TypeSerializer

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

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

Matters of State (III)

State Migration

State migration / evolution

- **Upgrading to more efficient serialization schemes** for performance improvements
- **Changing the schema / data model of state types**, due to evolving business logic

Upgrading State Serializers

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: New custom serializer

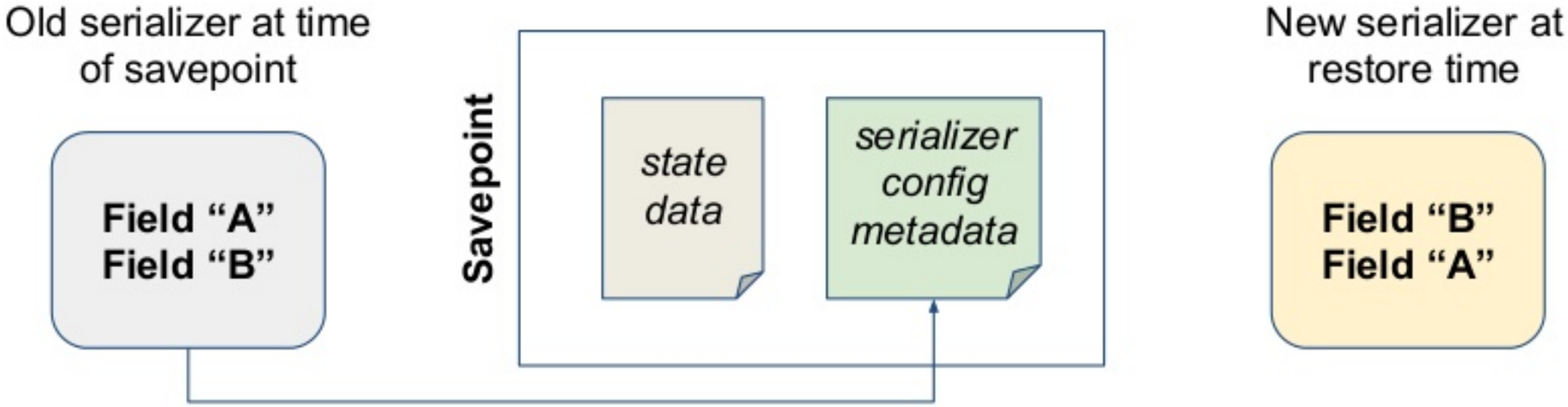
```
ValueStateDescriptor<MyCustomType> desc =
    new ValueStateDescriptor<>("my-value-state", new UpgradedSerializer<MyCustomType>());
```

- The new upgraded serializer would either be compatible, or not compatible. If incompatible, state migration is required.
- *Disclaimer:* as of Flink 1.3, upgraded serializers must be compatible, as state migration is not yet an available feature.

Upgrading State Serializers (II)

- On restore, new serializers are checked against the metadata of the previous serializer (stored together with state data in savepoints) for compatibility.

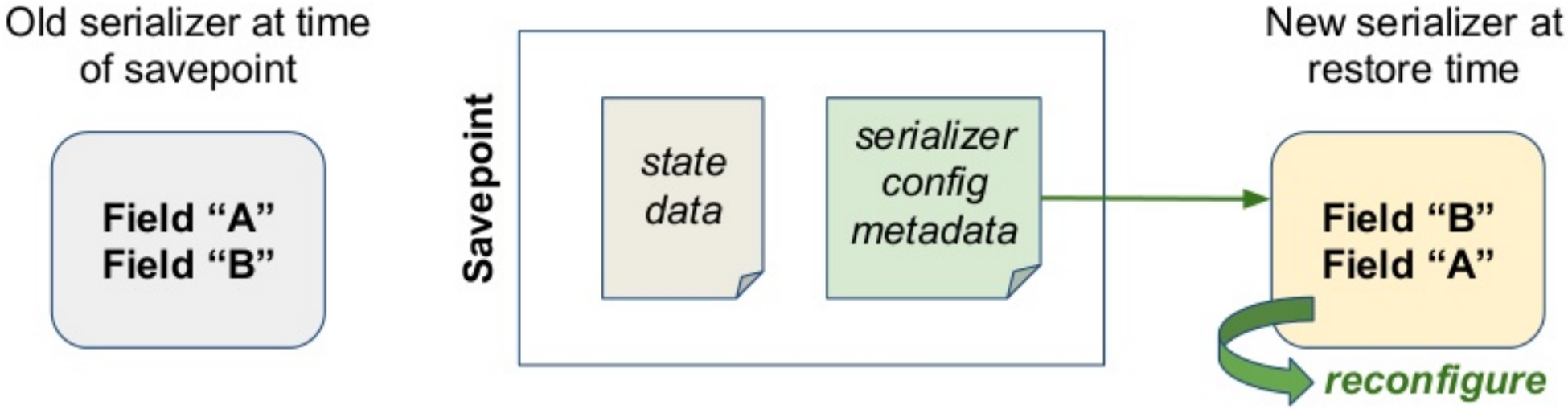
- All Flink-generated serializers define the metadata to be persisted. Newly generated serializers at restore time are *reconfigured* to be compatible.



Upgrading State Serializers (II)

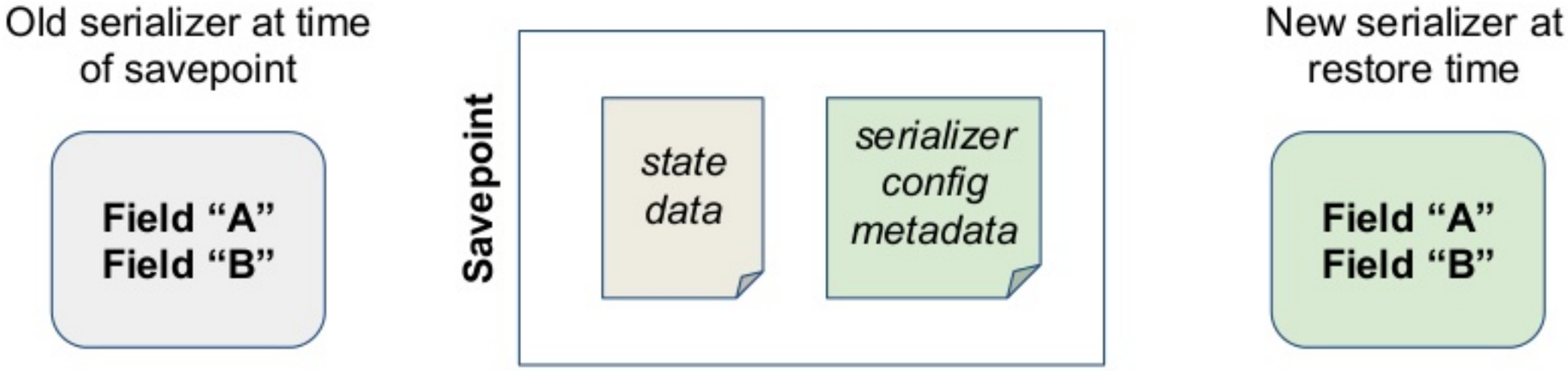
- On restore, new serializers are checked against the metadata of the previous serializer (stored together with state data in savepoints) for compatibility.

- All Flink-generated serializers define the metadata to be persisted. Newly generated serializers at restore time are *reconfigured* to be compatible.



Upgrading State Serializers (II)

- On restore, new serializers are checked against the metadata of the previous serializer (stored together with state data in savepoints) for compatibility.
- All Flink-generated serializers define the metadata to be persisted. Newly generated serializers at restore time are *reconfigured* to be compatible.



Upgrading State Serializers (III)

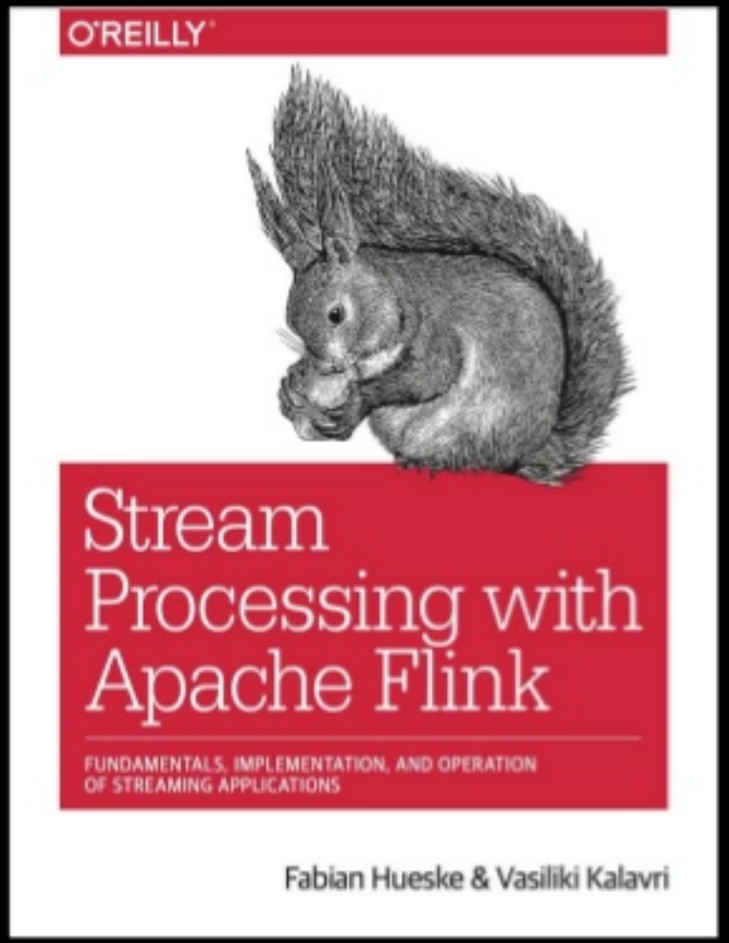
- Custom serializers must define the metadata to write: `TypeSerializerConfigSnapshot`
- Also define logic for compatibility checks

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

    ...

    TypeSerializerConfigSnapshot snapshotConfiguration();

    CompatibilityResult<MyCustomType> ensureCompatibility(
        TypeSerializerConfigSnapshot configSnapshot);
}
```

Closing

TL;DR

- The Flink community takes state management in Flink very seriously
- We try to make sure that users will feel comfortable in placing their application state within Flink and avoid any kind of lock-in.
- Upcoming changes / features to expect related to state management: Eager State Declaration & State Migration



Thank you!

@tzulitai
@ApacheFlink
@dataArtisans

We are hiring!

data-artisans.com/careers