DataStream API

State & Failure Recovery

Apache Flink® Training

dataArtisans

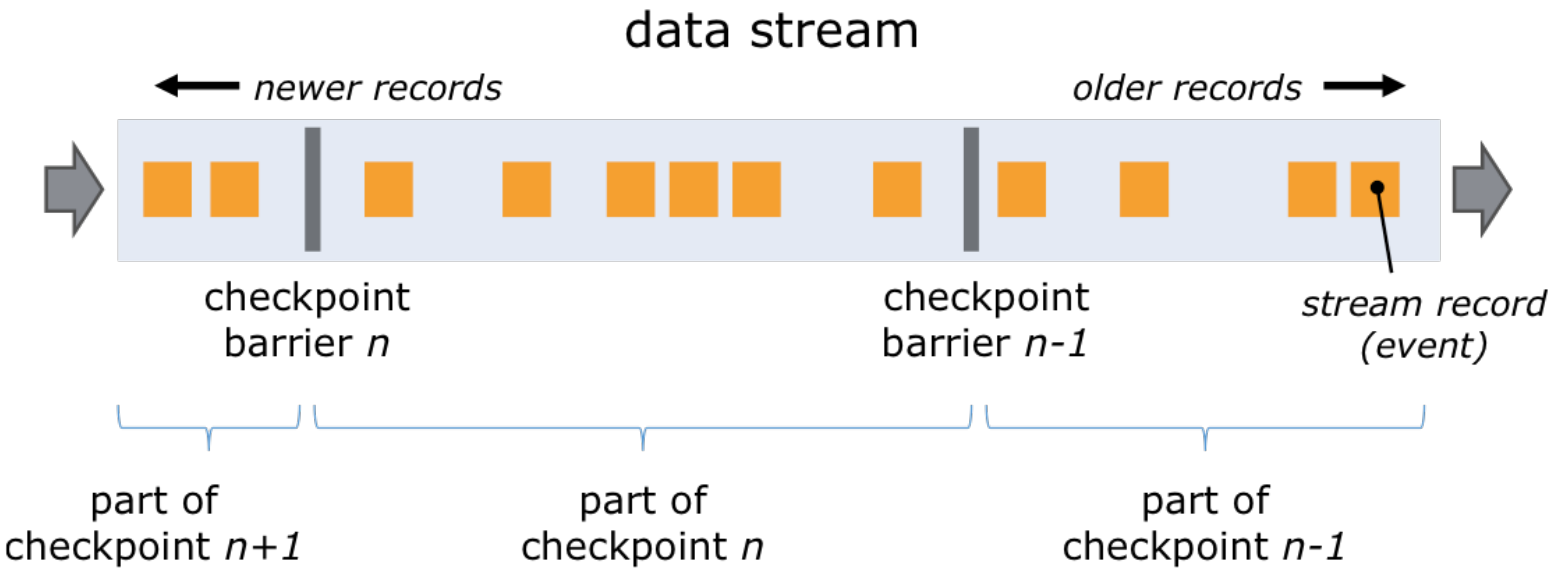
Flink v1.3 – 19.06.2017

Checkpoints

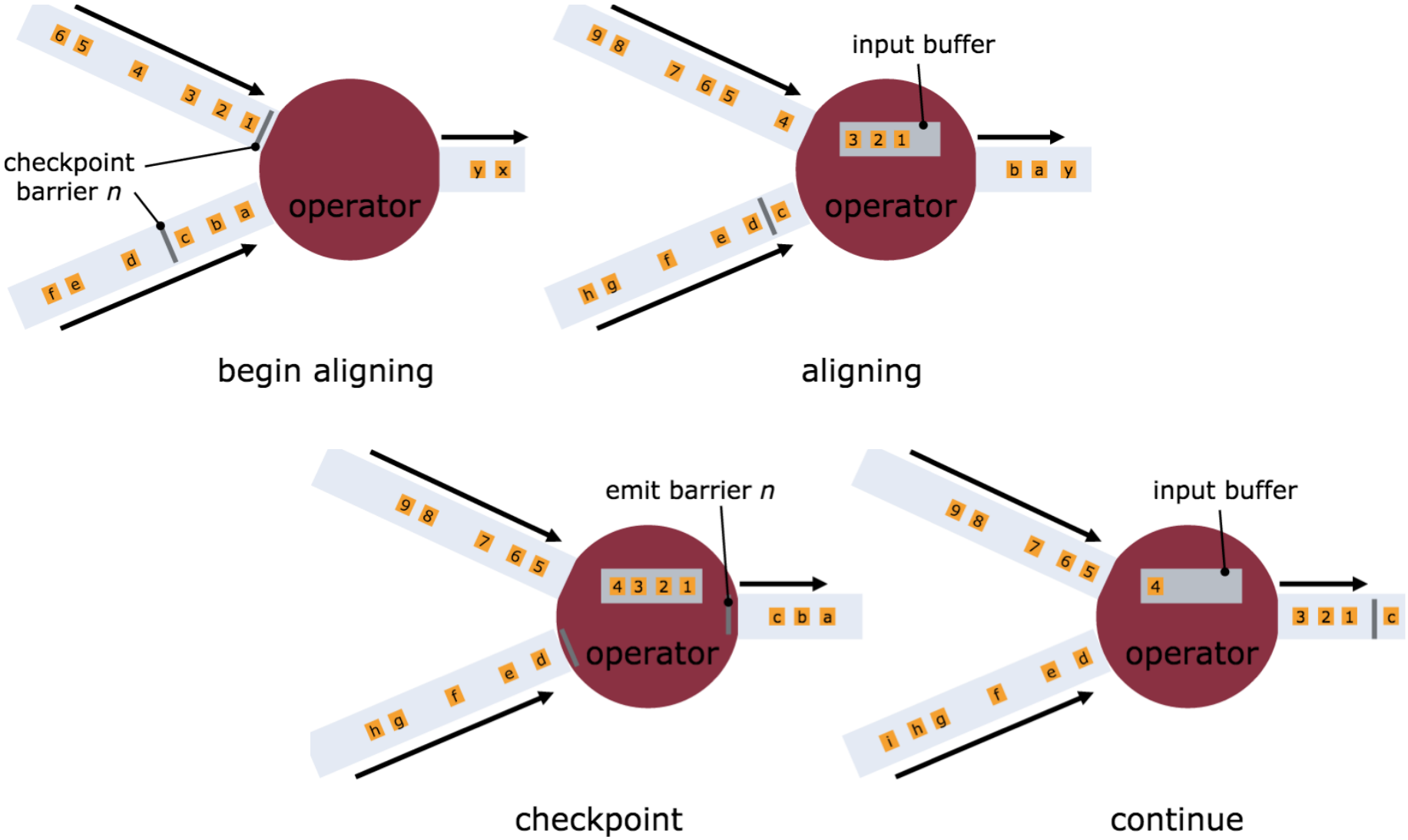
Checkpointing in Flink

- Asynchronous Barrier Snapshotting
  - checkpoint barriers are inserted into the stream and flow through the graph along with the data
  - this avoids a "global pause" during checkpointing
- Checkpoint barriers cause ...
  - replayable sources to checkpoint their offsets
  - operators to checkpoint their state
  - sinks to commit open transactions
- The program is rolled back to the latest completed checkpoint in case of a failure.

Checkpoint Barriers



Asynchronous Barrier Snapshotting



Enabling Checkpointing

- Checkpointing is disabled by default.
- Enable checkpointing with exactly once consistency:

```
// checkpoint every 5 seconds
env.enableCheckpointing(5000)
```

- Configure at least once consistency (for lower latency):

```
env.getCheckpointConfig()
   .setCheckpointingMode(CheckpointingMode.AT_LEAST_ONCE);
```

- Most applications perform well with a few seconds checkpointing interval.

Restart Strategies

- How often and fast does a job try to restart?
- Available strategies
  - No restart (default)
  - Fixed delay
  - Failure rate

```
// Fixed Delay restart strategy
env.setRestartStrategy(
  RestartStrategies.fixedDelayRestart(
    3,                          // no of restart attempts
    Time.of(10, TimeUnit.SECONDS) // restart interval
));
```

Working with State

Stateful Functions

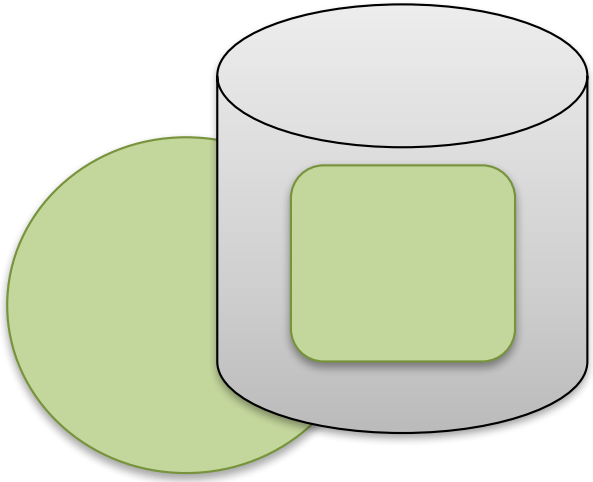
- All DataStream functions can be stateful
  - Flink manages state so that it can be redistributed/rescaled
  - State is checkpointed and restored in case of a failure (if checkpointing is enabled)

- Flink manages two types of state:
  - Operator (non-keyed) state
  - Keyed state

- Flink supports rescaling the state it manages

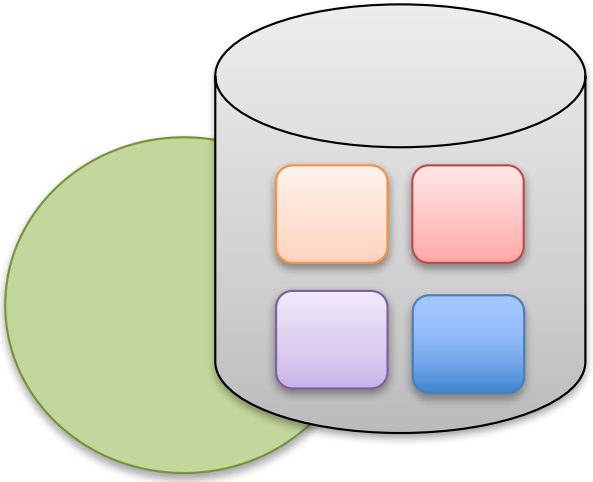
Operator vs Keyed State

Operator (non-keyed)



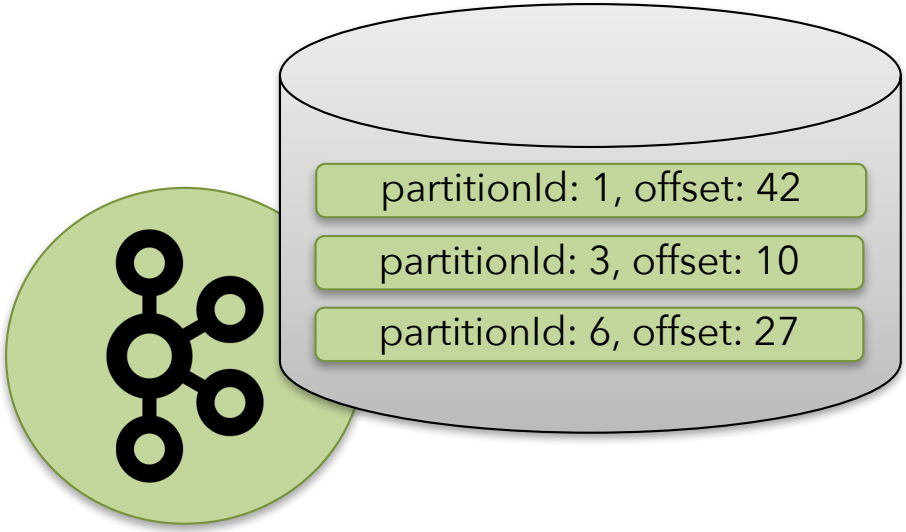
- State bound only to operator
- E.g. source state

Keyed



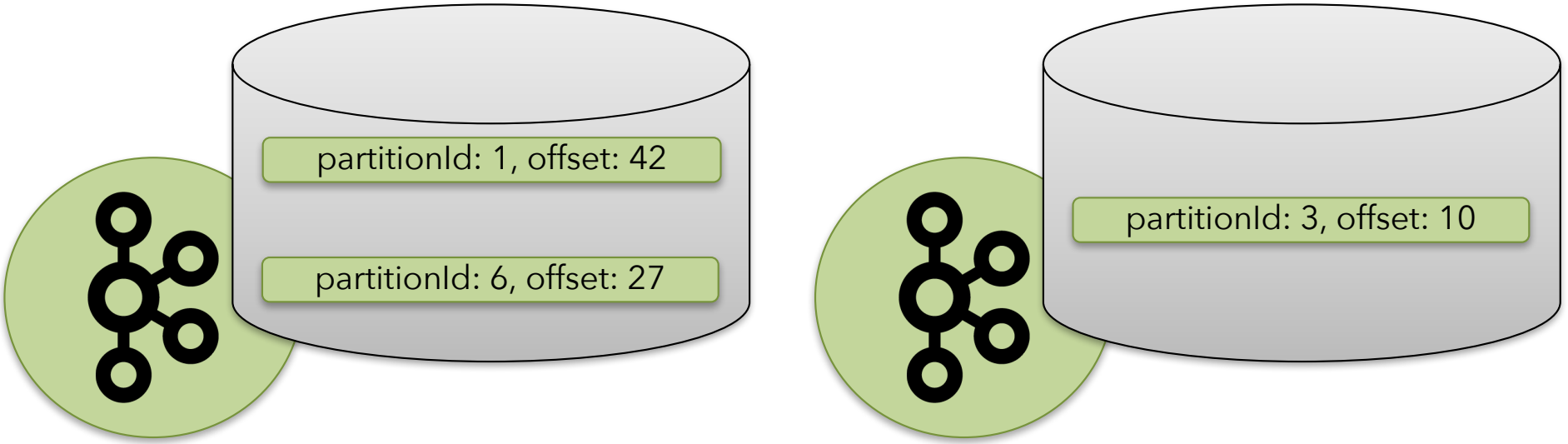
- State bound to an operator + key
- E.g. Keyed UDF and window state
- `"SELECT count(*) FROM t GROUP BY t.key"`

Repartitioning Operator State



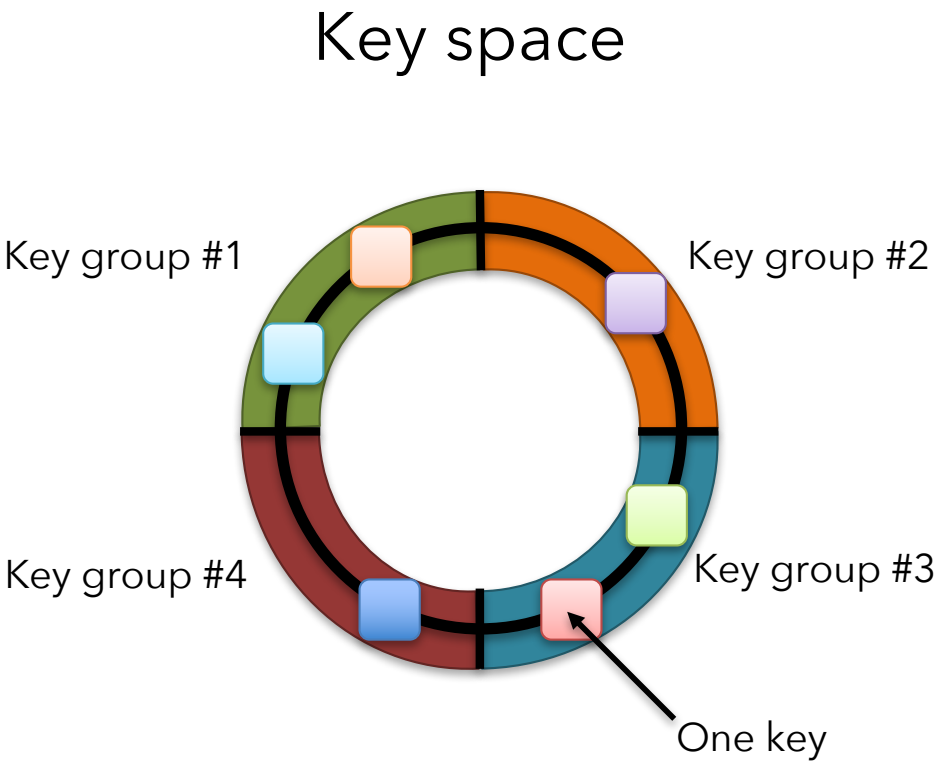
Operator state: a list of state elements which can be freely repartitioned

Scaling out

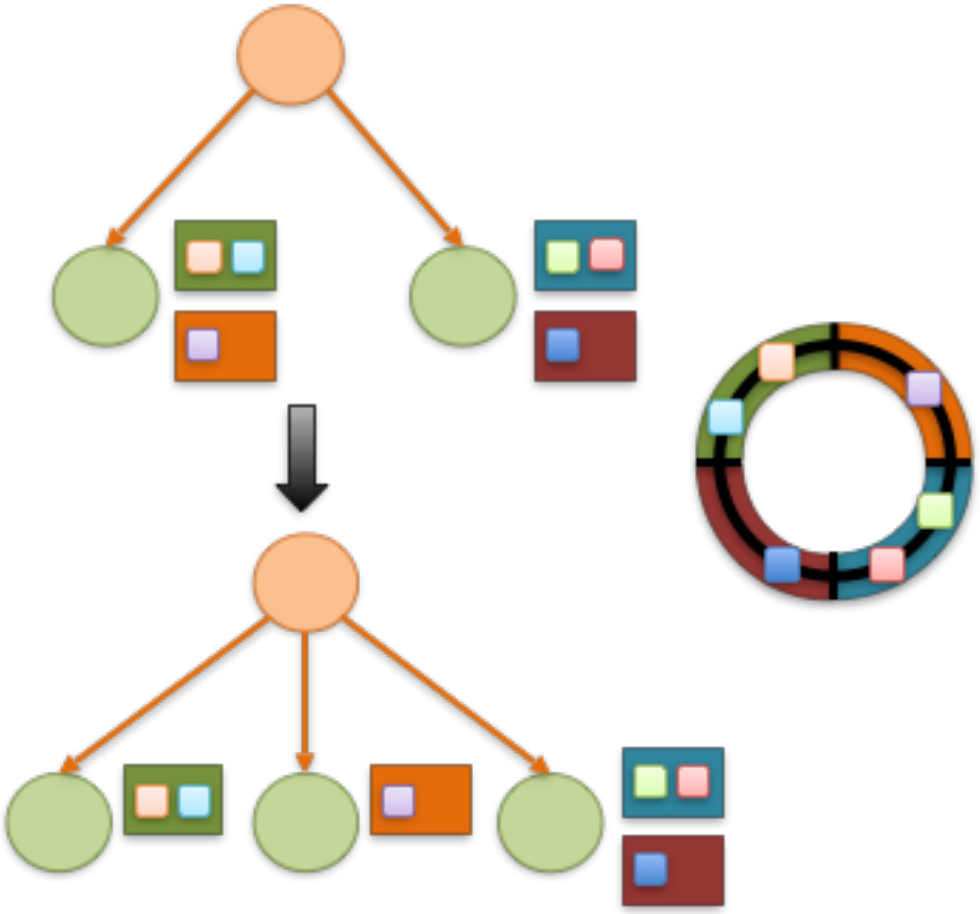

partitionId: 1, offset: 42
partitionId: 6, offset: 27
partitionId: 3, offset: 10

Repartitioning Keyed State

- Split key space into key groups
- Every key falls into exactly one key group
- Assign key groups to tasks
- Maximum parallelism defined by #key groups



Rescaling changes key group assignment



Types of Keyed State

- ValueState<T>
- ListState<T>
- ReducingState<T>
- MapState<UK, UV> *(new in 1.3)*
- ~~FoldingState~~<T> *(deprecated in 1.3)*
  - AggregatingState<IN, OUT>

Using Key-Partitioned State

```
DataStream<Tuple2<String, String>> strings = …
DataStream<Long> lengths = strings
  .keyBy(0)
  .map(new MapWithCounter());

public static class MapWithCounter extends RichMapFunction<Tuple2<String, String>, Long> {
    // state object
    private ValueState<Long> totalLengthByKey;

    @Override
    public void open (Configuration conf) {
        // obtain state object
        ValueStateDescriptor<Long> descriptor = new ValueStateDescriptor<>(
            "totalLengthByKey", Long.class, 0L);
        totalLengthByKey = getRuntimeContext().getState(descriptor);
    }

    @Override
    public Long map (Tuple2<String, String> value) throws Exception {
        long length = totalLengthByKey.value();    // fetch state for current key
        long newTotalLength = length + value.f1.length();
        totalLengthByKey.update(newTotalLength);   // update state of current key
        return totalLengthByKey.value();
    }
}
```

State Backends

State in Flink

- There are several sources of state in Flink
  - Windows
  - User functions
  - Sources and Sinks
  - Timers
- State is persisted during checkpoints, if checkpointing is enabled
- Internal representation and storage location depend on the configured State Backend

State Backends

- MemoryStateBackend (default)
  - State is held as objects on worker JVM heap
  - Checkpoints are stored on master JVM heap
  - Suitable for development and tiny state; not highly-available
- FsStateBackend
  - State is held on worker JVM heap (limited by heap size)
  - Checkpoints are written to a configured filesystem URI (hdfs, s3, file)
  - Suitable for jobs with large state and/or high-availability requirements
- RocksDBStateBackend
  - State is held in RocksDB instance on worker filesystem (limited by disk size)
  - Checkpoints are written to a configured filesystem URI (hdfs, s3, file)
  - Suitable for jobs with *very* large state and/or high-availability requirements

State Backend Configuration

- Configuration of default state backend in

  ./conf/flink-conf.yaml

- State backend configuration in job

```
env.setStateBackend(
    new FsStateBackend(
      "hdfs://namenode:40010/flink/checkpoints”
  ));
```

Savepoints

Savepoints

- A "Checkpoint" is a globally consistent point-in-time snapshot of a streaming application *(point in stream, state)*
- Savepoints are user-triggered, retained checkpoints
- Applications can be re-started from savepoints
- Savepoints are useful for
  - Application updates
  - Updating a Flink version
  - Maintenance & migration
  - A/B testing
  - Rescaling
- Currently, Flink can only restore to the same state backend that created the savepoint