CS 591 K1: Data Stream Processing and Analytics

Spring 2020

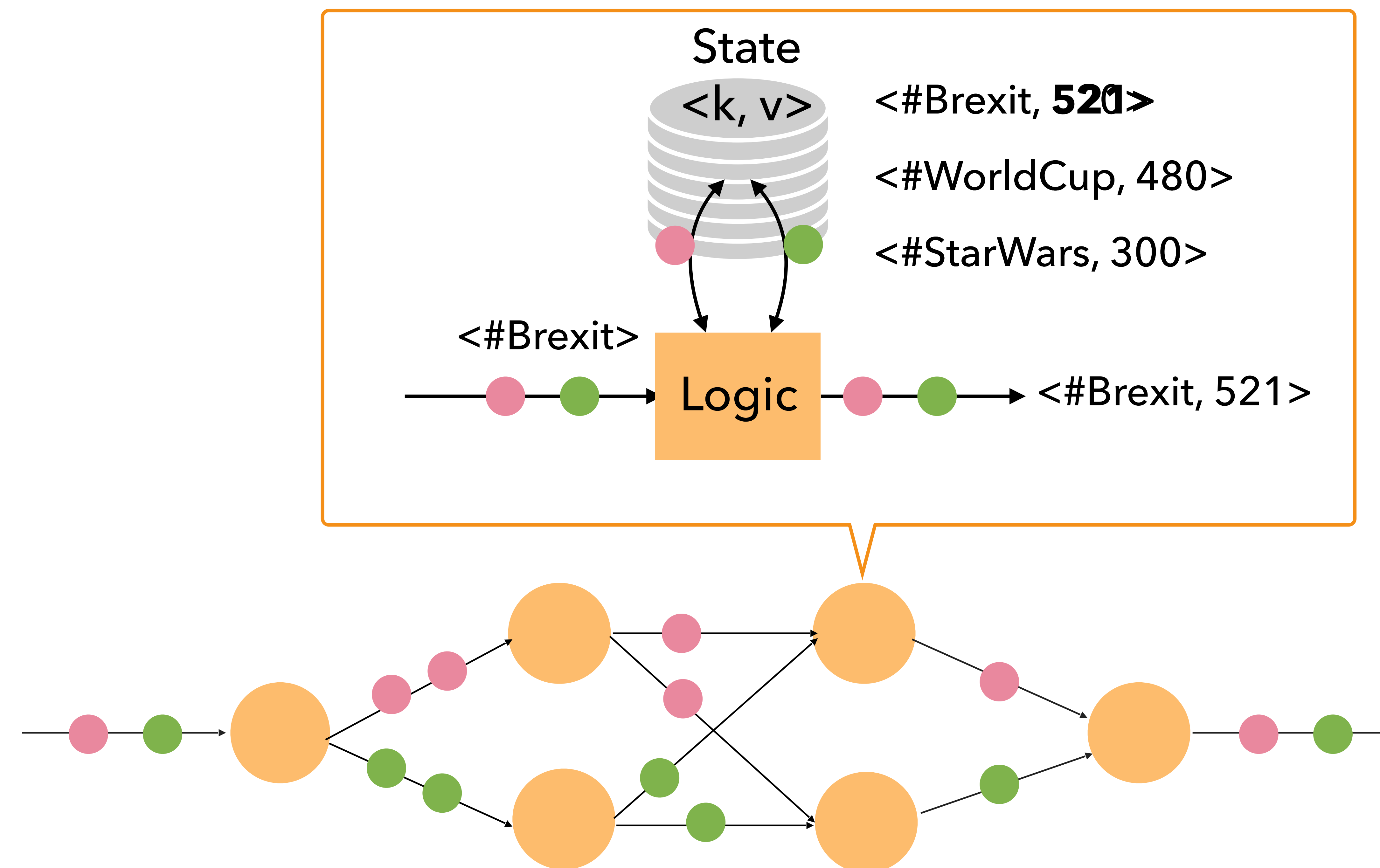
2/25: State Management

Vasiliki (Vasia) Kalavri
vkalavri@bu.edu

State in dataflow computations

Any non-trivial streaming computation maintains state:

- rolling aggregations
- window contents
- input offsets
- machine learning models



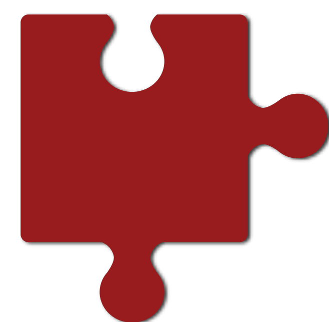
Streaming state

Unmanaged

- No explicit state primitives
- Users define state using arbitrary types
- The system is unaware of which parts of an operator constitute state

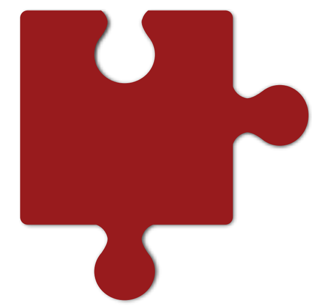
Managed

- Explicit state primitives including state types and interfaces
- The system is aware of state and can transparently checkpoint it, restore it, re-scale it



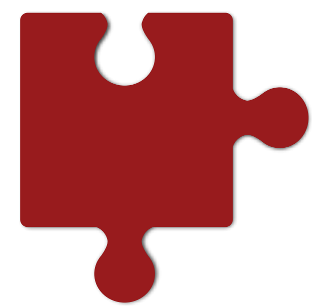
What are the advantages and disadvantages of each approach?

State operations and types



Consider you are designing a state interface. What operations should state support?

- Copy, checkpoint, restore, merge, split, query, subscribe, …



What state types can you think of?

- Count, sum, list, map, …

State management in Apache Flink

All data maintained by a task and used to compute results: a local or instance variable that is accessed by a task's business logic

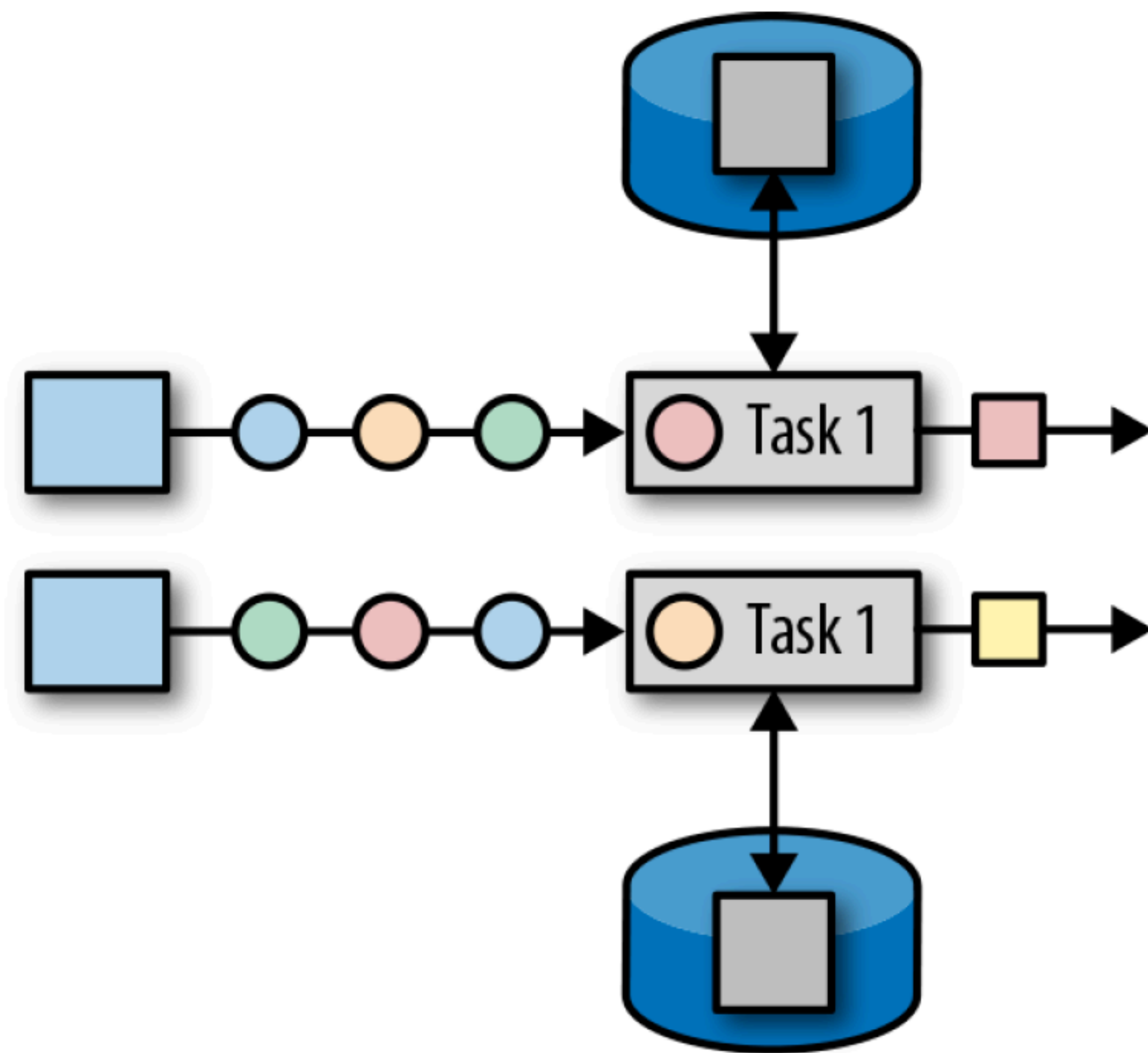
Operator state is scoped to an operator task, i.e. records processed by the same parallel task have access to the same state

- It cannot be accessed by other parallel tasks of the same or different operators

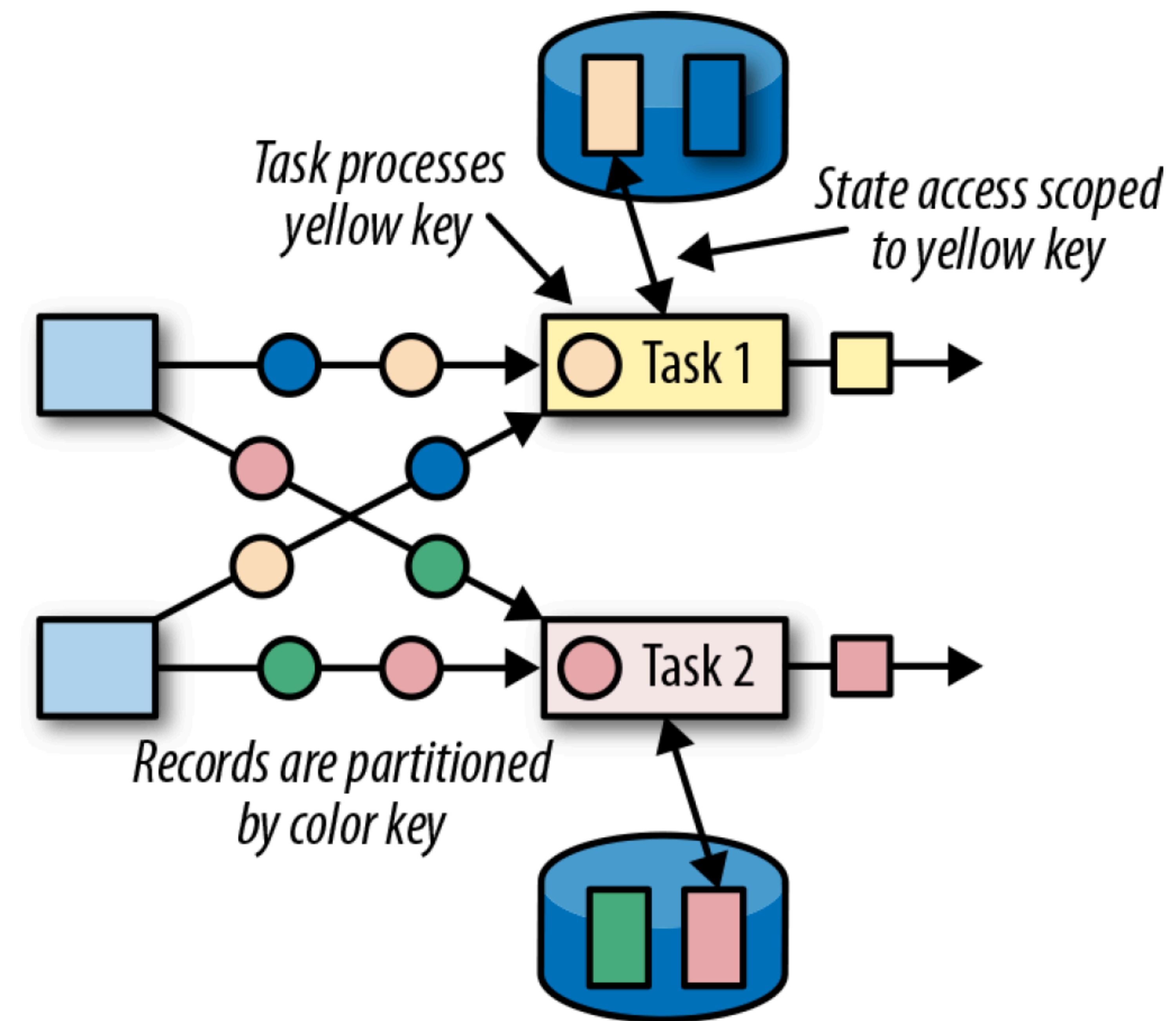
Keyed state is scoped to a key defined in the operator's input records

- Flink maintains one state instance per key value and partitions all records with the same key to the operator task that maintains the state for this key
- State access is automatically scoped to the key of the current record so that all records with the same key access the same state

State types



Operator state



Keyed state

State backends

A pluggable component that determines how state is stored, accessed, and maintained.

State backends are responsible for:

- local state management
- checkpointing state to remote and persistent storage, e.g. a distributed filesystem or a database system

- Available state backends in Flink:
  - In-memory
  - File system
  - RocksDB

Which backend to choose?

MemoryStateBackend

- Stores state as regular objects on TaskManager’s heap
- Low read/write latencies
- OutOfMemoryError if large grows too large, GC pauses
- Checkpoints sent to JobManager's heap memory, i.e. the state is lost in case of failure
- Use only for development and debugging purposes!

FsStateBackend

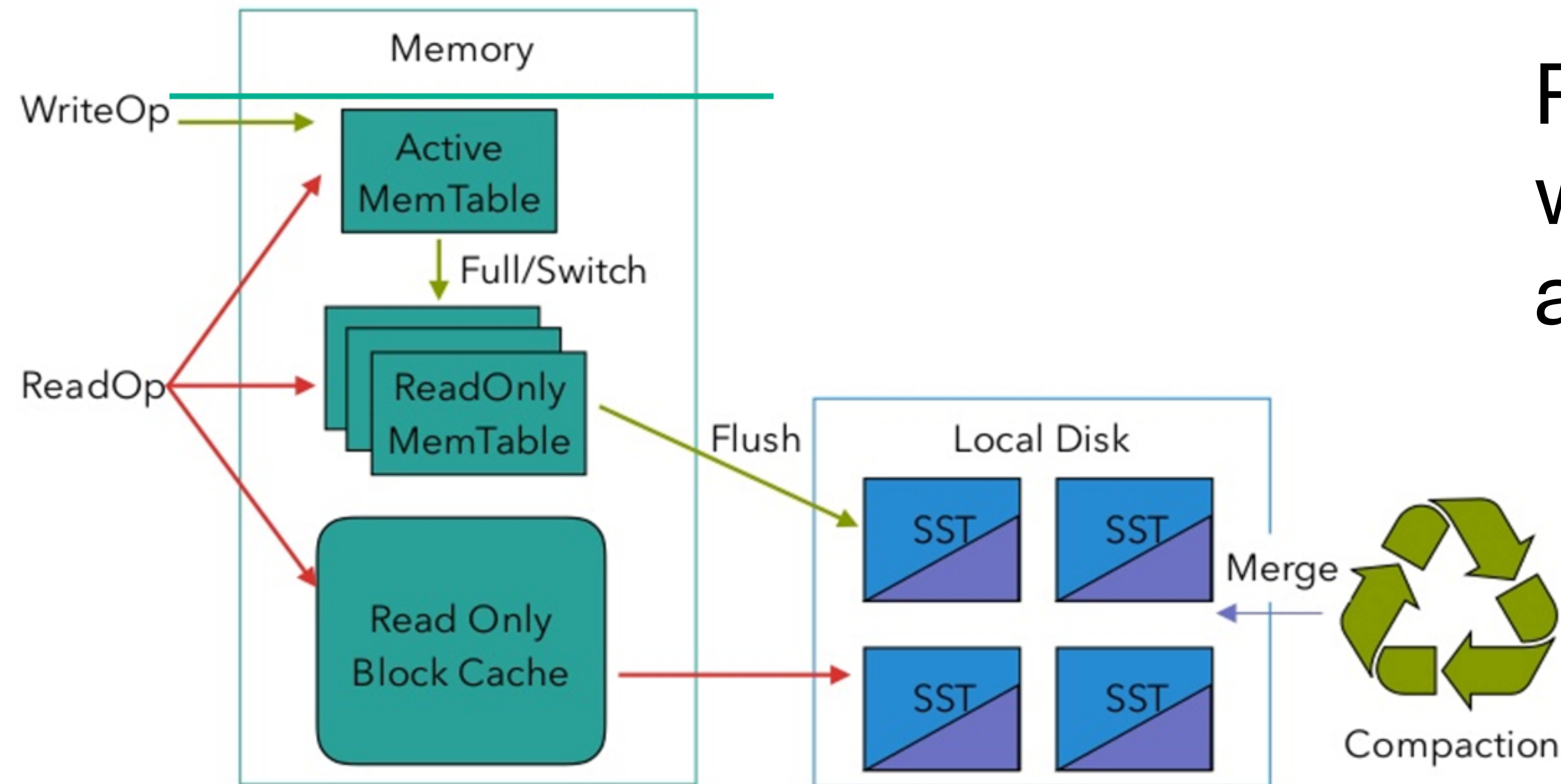
- Stores state on TaskManager’s heap but checkpoints it to a remote file system
- In-memory speed for local accesses and fault tolerance
- Limited to TaskManager’s memory and might suffer from GC pauses

Which backend to choose?

RocksDBStateBackend

- Stores all state into embedded RocksDB instances
- Accesses require de/serialization
- Checkpoints state to a remote file system and supports incremental checkpoints
- Use for applications with very large state

RocksDB



RocksDB is an LSM-tree storage engine with key/value interface, where keys and values are arbitrary byte streams.

https://rocksdb.org/

https://www.ververica.com/blog/manage-rocksdb-memory-size-apache-flink

RocksDB

- RocksDB is a *persistent* key value store: data lives on disk, state can grow larger than available memory and will not be lost upon failure.
- Keys and values are arbitrary byte arrays: serialization and deserialization is required to access the state via a Flink program.
- The keys are *ordered* according to a user-specified comparator function.

Basic operations

- **Get(key)**: fetch a single key-value from the DB
- **Put(key, val)**: insert a single key-value into the DB
- **Iterator/RangeScan**: seek to a specified key and then scan one key at a time from that point (keys are sorted)
- **Merge**: a lazy read-modify-write

Configuring the state backend

In `conf/flink.conf.yaml`:

```
# Supported backends are 'jobmanager', 'filesystem', 'rocksdb'
#
# state.backend: rocksdb
#
# Directory for checkpoints filesystem
#
# state.checkpoints.dir: path/to/checkpoint/folder/
```

In your Flink program:

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val checkpointPath: String = ???

// configure path for checkpoints on the remote filesystem
val backend = new RocksDBStateBackend(checkpointPath)

// configure the state backend
env.setStateBackend(backend)
```

Flink's state primitives

- **`ValueState[T]:`** a single value of type T
  - `ValueState.value()`
  - `ValueState.update(value: T)`

- **`ListState[T]:`** a list of elements of type T
  - `ListState.add(value: T)`
  - `ListState.addAll(values: java.util.List[T]).`
  - `List State.get(): Iterable[T]`
  - `ListState.update(values: java.util.List[T])`

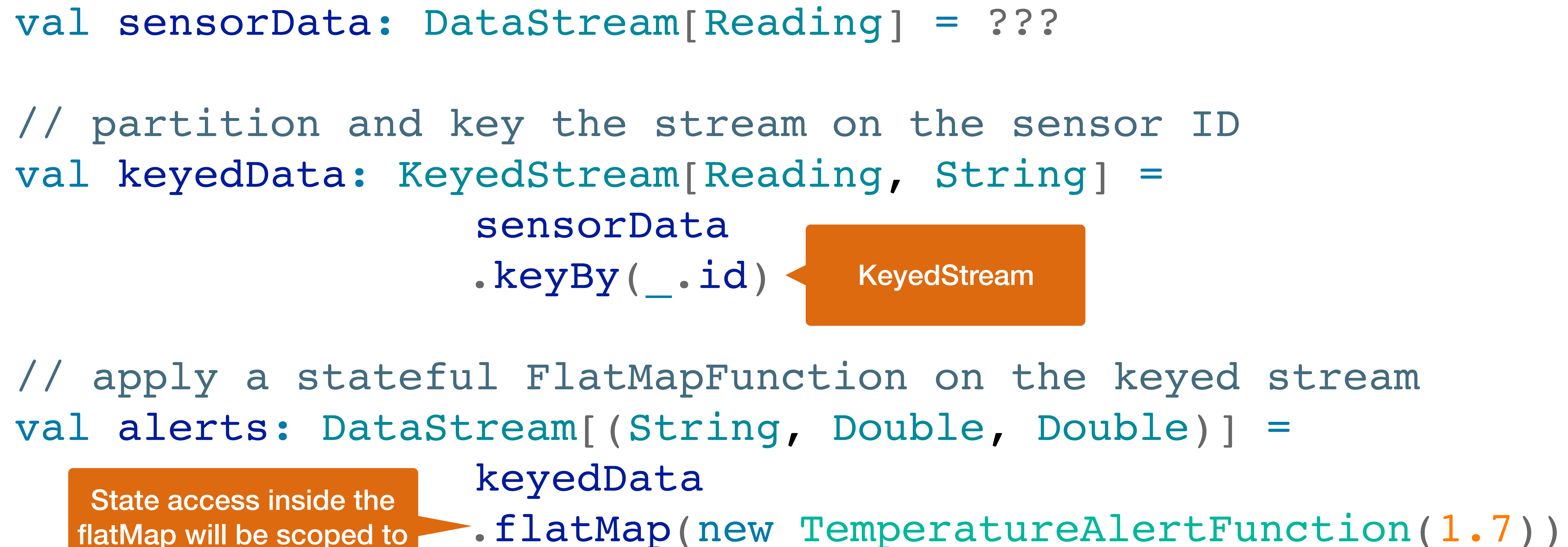
Flink's state primitives

- **`MapState[K, V]:`** a map of keys and values
  - `get(key: K), put(key: K, value: V), contains(key: K), remove(key: K)`
  - iterators over the contained entries, keys, and values

- **`ReducingState[T]:`** aggregates values using a `ReduceFunction`
  - `ReducingState.add(value: T)`
  - `ReducingState.get()`

- **`AggregatingState[I, O]:`** aggregates values using an `AggregateFunction`

Using state in Flink



Registering state

- To create a state object, we have to register a `StateDescriptor` with Flink's runtime via the `RuntimeContext`, which is exposed by **`RichFunctions`** (`RichFlatMapFunction`, `RichMapFunction`, (`Co`)`ProcessFunction`).

- The `StateDescriptor` is specific to the state primitive and includes the **name** of the state and the **data types** of the state:
  - The state name is scoped to the operator so that a function can have more than one state object by registering multiple state descriptors.
  - The data types handled by the state are specified as `Class` or `TypeInformation` objects.

Using state in Flink

```
class TemperatureAlertFunction(val threshold: Double)
  extends RichFlatMapFunction[Reading, (String, Double, Double)] {

1. // the state handle object
   private var lastTempState: ValueState[Double] = _

   override def open(parameters: Configuration): Unit = {
    // create state descriptor
2.  val lastTempDescriptor =
     new ValueStateDescriptor[Double]("lastTemp", classOf[Double])
    // obtain the state handle
    lastTempState = getRuntimeContext.getState[Double](lastTempDescriptor)
   }
   …

}
```

declare state handle

In the operator (FlatMap) class

assign name and get the state handle

In the open() method

Vasiliki Kalavri | Boston University 2020

Using state in Flink

```
class TemperatureAlertFunction(val threshold: Double)
  extends RichFlatMapFunction[SensorReading, (String, Double, Double)] {
 …

 override def flatMap(
   reading: SensorReading,
   out: Collector[(String, Double, Double)]): Unit = {

  // fetch the last temperature from state
3.val lastTemp = lastTempState.value() get state value
  // check if we need to emit an alert
  val tempDiff = (reading.temperature - lastTemp).abs
  if (tempDiff > threshold) {
   // temperature changed by more than the threshold
   out.collect((reading.id, reading.temperature, tempDiff))
  }
  // update lastTemp state     update state
4. this.lastTempState.update(reading.temperature)
 }
}
```

This is the state of the current key (sensor id)

Vasiliki Kalavri | Boston University 2020

Keyed state scope

Use keyed state to store and access state in the context of a key attribute:

- For each distinct value of the key attribute, Flink maintains one state instance.
- The keyed state instances of a function are distributed across all parallel tasks of the function's operator.

Keyed state can only be used by functions that are applied on a `KeyedStream`:

- When the processing method of a function with keyed input is called, Flink's runtime automatically puts all keyed state objects of the function into the context of the key of the record that is passed by the function call.
- A function **can only access the state that belongs to the record it currently processes**.

Java example

```
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

// taxi ride events (start, end)
DataStream<TaxiRide> rides = env.addSource(…)).keyBy("rideId");

// taxi fare events (payment, tip)
DataStream<TaxiFare> fares = env.addSource(…).keyBy("rideId");

// match ride and fare events
DataStream<Tuple2<TaxiRide, TaxiFare>> connectedRides = rides
    .connect(fares)
    .flatMap(new MatchFunction());
```

Java example (cont.)

```java
public static class EnrichmentFunction extends RichCoFlatMapFunction<TaxiRide, TaxiFare, Tuple2<TaxiRide, TaxiFare>> {
   // define the state primitives here
   private ValueState<TaxiRide> rideState;
   private ValueState<TaxiFare> fareState;

   @Override
   public void open(Configuration config) {
      // initialize the state descriptors here
      rideState = getRuntimeContext().getState(new ValueStateDescriptor<>("saved ride", TaxiRide.class));
      fareState = getRuntimeContext().getState(new ValueStateDescriptor<>("saved fare", TaxiFare.class));
   }

   @Override
   public void flatMap1(TaxiRide ride, Collector<Tuple2<TaxiRide, TaxiFare>> out) throws Exception {
      TaxiFare fare = fareState.value();
      if (fare != null) { // a matching fare exists
         fareState.clear(); // always clear the state you don’t need anymore!
         out.collect(new Tuple2(ride, fare));
      } else {
         rideState.update(ride); // no matching fare -> store the ride
      }
   }

   @Override
   public void flatMap2(TaxiFare fare, Collector<Tuple2<TaxiRide, TaxiFare>> out) throws Exception {
     // similar logic for processing fare events
      }
   }
}
```

Operator state

- A function can work with operator list state by implementing the `ListCheckpointed` interface
- `snapshotState()` is invoked when Flink triggers a checkpoint of the stateful function.
- `restoreState()` is always invoked when the job is started or in the case of a failure.

```
List<T> snapshotState(long checkpointId, long timestamp)
void restoreState(List<T> state)
```

A stateful source

```java
public static class CounterSource extends RichParallelSourceFunction<Long> implements ListCheckpointed<Long> {

    /**  current offset for exactly once semantics */
    private Long offset = 0L;
    private volatile boolean isRunning = true;

    @Override
    public void run(SourceContext<Long> ctx) {
        final Object lock = ctx.getCheckpointLock();

        while (isRunning) {
            // output and state update are atomic
            synchronized (lock) {
                ctx.collect(offset);
                offset += 1;
            }
        }
    }

    @Override
    public List<Long> snapshotState(long checkpointId, long checkpointTimestamp) {
        return Collections.singletonList(offset);
    }

    @Override
    public void restoreState(List<Long> state) {
        for (Long s : state)
            offset = s;
    }
}
```

get a lock to make output and state update atomic

Further resources

- Working with State: https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/stream/state/state.html
- Managing State in Apache Flink - Tzu-Li (Gordon) Tai: https://www.youtube.com/watch?v=euFMWFDThiE
- Webinar: Deep Dive on Apache Flink State - Seth Wiesman: https://www.youtube.com/watch?v=9GF8Hwqzwnk