**CHAPTER 7**

**Stateful Operators and Applications**

Stateful operators and user functions are common building blocks of stream process‐ ing applications. In fact, most nontrivial operations need to memorize records or par‐ tial results because data is streamed and arrives over time.[1](#_bookmark21) Many of Flink’s built-in DataStream operators, sources, and sinks are stateful and buffer records or maintain partial results or metadata. For instance, a window operator collects input records for

a ProcessWindowFunction or the result of applying a ReduceFunction, a Process Function memorizes scheduled timers, and some sink functions maintain state about transactions to provide exactly-once functionality. In addition to built-in operators

and provided sources and sinks, Flink’s DataStream API exposes interfaces to register, maintain, and access state in user-defined functions.

Stateful stream processing has implications on many aspects of a stream processor such as failure recovery and memory management as well as the maintenance of streaming applications. Chapters [2](#_bookmark0) and [3](#_bookmark0) discussed the foundations of stateful stream processing and related details of Flink’s architecture, respectively. [Chapter 9](#_bookmark0) explains how to set up and configure Flink to reliably process stateful applications. [Chapter 10](#_bookmark0) gives guidance on how to operate stateful applications—taking and restoring from application savepoints, rescaling applications, and performing application upgrades.

This chapter focuses on the implementation of stateful user-defined functions and discusses the performance and robustness of stateful applications. Specifically, we explain how to define and interact with different types of state in user-defined func‐ tions. We also discuss performance aspects and how to control the size of function

1. This differs from batch processing where user-defined functions, such as GroupReduceFunction, are called when all data to be processed has been collected.

**153**

state. Finally, we show how to configure keyed state as queryable and how to access it from an external application.

# Implementing Stateful Functions

In [“State Management”](#_bookmark1), we explained that functions can have two types of state, keyed state and operator state. Flink provides multiple interfaces to define stateful functions. In this section, we show how functions with keyed and operator state are implemented.

## Declaring Keyed State at RuntimeContext

User functions can employ 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. That means each parallel instance of the function is responsible for a subrange of the key domain and maintains the corresponding state instances. Hence, keyed state is very similar to a distributed key-value map. Consult [“State Management” on page 53](#_bookmark1) for more details on keyed state.

Keyed state can only be used by functions that are applied on a KeyedStream. A Keyed Stream is constructed by calling the DataStream.keyBy() method that defines a key on a stream. A KeyedStream is partitioned on the specified key and remembers the key definition. An operator that is applied on a KeyedStream is applied in the context

of its key definition.

Flink provides multiple primitives for keyed state. A state primitive defines the struc‐ ture of the state for an individual key. The choice of the right state primitive depends on how the function interacts with the state. The choice also affects the performance of a function because each state backend provides its own implementations for these primitives. The following state primitives are supported by Flink:

* + ValueState[T] holds a single value of type T. The value can be read using Value State.value() and updated with ValueState.update(value: T).
  + ListState[T] holds a list of elements of type T. New elements can be appended to the list by calling ListState.add(value: T) or ListState.addAll(values: java.util.List[T]). The state elements can be accessed by calling List State.get(), which returns an Iterable[T] over all state elements. It is not pos‐ sible to remove individual elements from ListState, but the list can be updated by calling ListState.update(values: java.util.List[T]). A call to this

method will replace existing values with the given list of values.

* + MapState[K, V] holds a map of keys and values. The state primitive offers many of the methods of a regular Java Map such as get(key: K), put(key: K, value:

V), contains(key: K), remove(key: K), and iterators over the contained entries, keys, and values.

* + ReducingState[T] offers the same methods as ListState[T] (except for addAll() and update()), but instead of appending values to a list, Reducing State.add(value: T) immediately aggregates value using a ReduceFunction. The iterator returned by get() returns an Iterable with a single entry, which is

the reduced value.

* + AggregatingState[I, O] behaves similar to ReducingState. However, it uses the more general AggregateFunction to aggregate values. Aggregating State.get() computes the final result and returns it as an Iterable with a single element.

All state primitives can be cleared by calling State.clear().

[Example 7-1](#_bookmark3) shows how to apply a FlatMapFunction with a keyed ValueState on a stream of sensor measurements. The example application emits an alert event if the

temperature measured by a sensor changes by more than a threshold since the last measurement.

*Example 7-1. Applying a FlatMapFunction with a keyed ValueState*

**val** sensorData**: DataStream**[**SensorReading**] **=** ???

*// partition and key the stream on the sensor ID*

**val** keyedData**: KeyedStream**[**SensorReading**, **String**] **=** sensorData

.keyBy(**\_**.id)

*// apply a stateful FlatMapFunction on the keyed stream which*

*// compares the temperature readings and raises alerts*

**val** alerts**: DataStream**[(**String**, **Double**, **Double**)] **=** keyedData

.flatMap(**new TemperatureAlertFunction**(1.7))

A function with keyed state must be applied on a KeyedStream. We need to specify the key by calling keyBy() on the input stream before we apply the function. When the processing method of a function with keyed input is called, Flink’s runtime auto‐

matically puts all keyed state objects of the function into the context of the key of the record that is passed by the function call. Therefore, a function can only access the state that belongs to the record it currently processes.

[Example 7-2](#_bookmark1) shows the implementation of a FlatMapFunction with a keyed Value State that checks whether the measured temperature changed more than a config‐ ured threshold.

*Example 7-2. Implementing a FlatMapFunction with a keyed ValueState*

**class TemperatureAlertFunction**(**val** threshold**: Double**)

**extends RichFlatMapFunction**[**SensorReading**, (**String**, **Double**, **Double**)] {

*// the state handle object*

**private var** lastTempState**: ValueState**[**Double**] **= \_**

**override def** open(parameters**: Configuration**)**: Unit** = {

*// create state descriptor*

**val** lastTempDescriptor **=**

**new ValueStateDescriptor**[**Double**]("lastTemp", classOf[**Double**])

*// obtain the state handle*

lastTempState **=** getRuntimeContext.getState[**Double**](lastTempDescriptor)

}

**override def** flatMap( reading**: SensorReading**,

out**: Collector**[(**String**, **Double**, **Double**)])**: Unit** = {

*// fetch the last temperature from state*

**val** lastTemp **=** lastTempState.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*

**this**.lastTempState.update(reading.temperature)

}

}

To create a state object, we have to register a StateDescriptor with Flink’s runtime via the RuntimeContext, which is exposed by RichFunction (see [“Implementing](#_bookmark1) [Functions” on page 105](#_bookmark1) for a discussion of the RichFunction interface). The StateDe scriptor is specific to the state primitive and includes the name of the state and the data types of the state. The descriptors for ReducingState and AggregatingState also need a ReduceFunction or AggregateFunction object to aggregate the added

values. 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 (see [“Types” on page](#_bookmark16) [97](#_bookmark16) for a discussion of Flink’s type handling). The data type must be specified because

Flink needs to create a suitable serializer. Alternatively, it is also possible to explicitly

specify a TypeSerializer to control how state is written into a state backend, check‐ point, and savepoint.[2](#_bookmark22)

Typically, the state handle object is created in the open() method of RichFunction. open() is called before any processing methods, such as flatMap() in the case of a FlatMapFunction, are called. The state handle object (lastTempState in [Example 7-2](#_bookmark1)) is a regular member variable of the function class.

The state handle object only provides access to the state, which is stored an maintained in the state backend. The handle does not hold the state itself.

When a function registers a StateDescriptor, Flink checks if the state backend has data for the function and a state with the given name and type. This might happen if

the stateful function is restarted to recover from a failure or when an application is started from a savepoint. In both cases, Flink links the newly registered state handle object to the existing state. If the state backend does not contain state for the given descriptor, the state that is linked to the handle is initialized as empty.

The Scala DataStream API offers syntactic shortcuts to define map and flatMap func‐ tions with a single ValueState. [Example 7-3](#_bookmark10) shows how to implement the previous example with the shortcut.

*Example 7-3. Scala DataStream API shortcut for a FlatMap with a keyed ValueState*

**val** alerts**: DataStream**[(**String**, **Double**, **Double**)] **=** keyedData

.flatMapWithState[(**String**, **Double**, **Double**), **Double**] {

**case** (in**: SensorReading**, **None**) **=>**

*// no previous temperature defined; just update the last temperature*

(**List**.empty, **Some**(in.temperature))

**case** (r**: SensorReading**, lastTemp**: Some**[**Double**]) **=>**

*// compare temperature difference with threshold* **val** tempDiff **=** (r.temperature - lastTemp.get).abs **if** (tempDiff > 1.7) {

*// threshold exceeded; emit an alert and update the last temperature*

(**List**((r.id, r.temperature, tempDiff)), **Some**(r.temperature))

} **else** {

*// threshold not exceeded; just update the last temperature*

(**List**.empty, **Some**(r.temperature))

}

}

1. The serialization format of state is an important aspect when updating an application and is discussed later in this chapter.

The flatMapWithState() method expects a function that accepts a Tuple2. The first field of the tuple holds the input record to flatMap, and the second field holds an Option of the retrieved state for the key of the processed record. Option is not defined if the state has not been initialized yet. The function also returns a Tuple2. The first field is a list of the flatMap results, and the second field is the new value of the state.

## Implementing Operator List State with the ListCheckpointed Interface

Operator state is managed per parallel instance of an operator. All events that are pro‐ cessed in the same parallel task of an operator have access to the same state. In [“State](#_bookmark1) [Management” on page 53](#_bookmark1), we discussed that Flink supports three types of operator state: list state, list union state, and broadcast state.

A function can work with operator list state by implementing the ListCheckpointed interface. The ListCheckpointed interface does not work with state handles like Val ueState or ListState, which are registered at the state backend. Instead, functions implement operator state as regular member variables and interact with the state backend via callback functions of the ListCheckpointed interface. The interface pro‐ vides two methods:

*// returns a snapshot the state of the function as a list*

snapshotState(checkpointId**: Long**, timestamp**: Long**)**: java.util.List**[**T**]

*// restores the state of the function from the provided list*

restoreState(java.util.**List**[**T**] state)**: Unit**

The snapshotState() method is invoked when Flink triggers a checkpoint of the stateful function. The method has two parameters, checkpointId, which is a unique, monotonically increasing identifier for checkpoints, and timestamp, which is the wall-clock time when the master initiated the checkpoint. The method has to return

the operator state as a list of serializable state objects.

The restoreState() method is always invoked when the state of a function needs to be initialized—when the job is started (from a savepoint or not) or in the case of a

failure. The method is called with a list of state objects and has to restore the state of the operator based on these objects.

[Example 7-4](#_bookmark23) shows how to implement the ListCheckpointed interface for a function that counts temperature measurements that exceed a threshold per partition, for each

parallel instance of the function.

*Example 7-4. A RichFlatMapFunction with operator list state*

**class HighTempCounter**(**val** threshold**: Double**)

**extends RichFlatMapFunction**[**SensorReading**, (**Int**, **Long**)]

**with ListCheckpointed**[**java.lang.Long**] {

*// index of the subtask*

**private lazy val** subtaskIdx **=** getRuntimeContext

.getIndexOfThisSubtask

*// local count variable*

**private var** highTempCnt **=** 0L

**override def** flatMap( in**: SensorReading**,

out**: Collector**[(**Int**, **Long**)])**: Unit** = {

**if** (in.temperature > threshold) {

*// increment counter if threshold is exceeded*

highTempCnt += 1

*// emit update with subtask index and counter*

out.collect((subtaskIdx, highTempCnt))

}

}

**override def** restoreState(

state**: util.List**[**java.lang.Long**])**: Unit** = { highTempCnt **=** 0

*// restore state by adding all longs of the list*

**for** (cnt **<-** state.asScala) { highTempCnt += cnt

}

}

**override def** snapshotState( chkpntId**: Long**,

ts**: Long**)**: java.util.List**[**java.lang.Long**] **=** {

*// snapshot state as list with a single count*

java.util.**Collections**.singletonList(highTempCnt)

}

}

The function in the above example counts per parallel instance how many tempera‐ ture measurements exceeded a configured threshold. The function uses operator state and has a single state variable for each parallel operator instance that is checkpointed

and restored using the methods of the ListCheckpointed interface. Note that the ListCheckpointed interface is implemented in Java and expects a java.util.List instead of a Scala native list.

Looking at the example, you might wonder why operator state is handled as a list of state objects. As discussed in [“Scaling Stateful Operators” on page 56](#_bookmark9), the list struc‐ ture supports changing the parallelism of functions with operator state. In order to increase or decrease the parallelism of a function with operator state, the operator state needs to be redistributed to a larger or smaller number of task instances. This requires splitting or merging of state objects. Since the logic for splitting and merging

of state is custom for every stateful function, this cannot be automatically done for arbitrary types of state.

By providing a list of state objects, functions with operator state can implement this logic using the snapshotState() and restoreState() methods. The snapshot State() method splits the operator state into multiple parts and the restoreState()

method assembles the operator state from possibly multiple parts. When the state of a

function is restored, the parts of the state are distributed among all parallel instances of the function and handed to the restoreState() method. If there are more parallel subtasks than state objects, some subtasks are started with no state, and the restoreState() method is called with an empty list.

Looking again at the HighTempCounter function in [Example 7-4](#_bookmark23), we see that each par‐ allel instance of the operator exposes its state as a list with a single entry. If we

increased the parallelism of this operator, some of the new subtasks would be initial‐ ized with an empty state, and start counting from zero. In order to achieve better state

distribution behavior when the HighTempCounter function is rescaled, we can imple‐ ment the snapshotState() method so that it splits its count into multiple partial counts as shown in [Example 7-5](#_bookmark4).

*Example 7-5. Split operator list state for better distribution during rescaling*

**override def** snapshotState( chkpntId**: Long**,

ts**: Long**)**: java.util.List**[**java.lang.Long**] **=** {

*// split count into ten partial counts*

**val** div **=** highTempCnt / 10

**val** mod **=** (highTempCnt % 10).toInt

*// return count as ten parts*

(**List**.fill(mod)(**new** java.lang.**Long**(div + 1)) ++

**List**.fill(10 - mod)(**new** java.lang.**Long**(div))).asJava

}

### ListCheckpointed Interface Uses Java Serialization

The ListCheckpointed interface uses Java serialization to serialize and deserialize the list of state objects. This can be a problem if you

need to update your application because Java serialization does not allow for migrating or configuring a custom serializer. Implement

CheckpointedFunction instead of the ListCheckpointed interface

if you need to ensure a function’s operator state can be evolved.

## Using Connected Broadcast State

A common requirement in streaming applications is to distribute the same informa‐ tion to all parallel instances of a function and maintain it as recoverable state. An

example is a stream of rules and a stream of events on which the rules are applied. The function that applies the rules ingests two input streams, the event stream and the rules stream. It remembers the rules in an operator state in order to apply them to all events of the event stream. Since each parallel instance of the function must hold all rules in its operator state, the rules stream needs to be broadcasted to ensure each instance of the function receives all rules.

In Flink, such a state is called a broadcast state. Broadcast state can be combined with a regular DataStream or KeyedStream. [Example 7-6](#_bookmark5) shows how to implement a tem‐ perature alert application with thresholds that can be dynamically configured via a

broadcasted stream.

*Example 7-6. Connecting a broadcast stream and keyed event stream*

**val** sensorData**: DataStream**[**SensorReading**] **=** ???

**val** thresholds**: DataStream**[**ThresholdUpdate**] **=** ???

**val** keyedSensorData**: KeyedStream**[**SensorReading**, **String**] **=** sensorData.keyBy(**\_**.id)

*// the descriptor of the broadcast state*

**val** broadcastStateDescriptor **=**

**new MapStateDescriptor**[**String**, **Double**]( "thresholds", classOf[**String**], classOf[**Double**])

**val** broadcastThresholds**: BroadcastStream**[**ThresholdUpdate**] **=** thresholds

.broadcast(broadcastStateDescriptor)

*// connect keyed sensor stream and broadcasted rules stream*

**val** alerts**: DataStream**[(**String**, **Double**, **Double**)] **=** keyedSensorData

.connect(broadcastThresholds)

.process(**new UpdatableTemperatureAlertFunction**())

A function with broadcast state is applied on two streams in three steps:

1. You create a BroadcastStream by calling DataStream.broadcast() and provid‐ ing one or more MapStateDescriptor objects. Each descriptor defines a separate broadcast state of the function that is later applied on the BroadcastStream.
2. You connect the BroadcastStream with a DataStream or KeyedStream. The

BroadcastStream must be put as an argument in the connect() method.

1. You apply a function on the connected streams. Depending on whether the other stream is keyed or not, a KeyedBroadcastProcessFunction or BroadcastPro cessFunction can be applied.

[Example 7-7](#_bookmark1) shows the implementation of a KeyedBroadcastProcessFunction that supports the dynamic configuration of sensor thresholds at runtime.

*Example 7-7. Implementing a KeyedBroadcastProcessFunction*

**class UpdatableTemperatureAlertFunction**() **extends KeyedBroadcastProcessFunction**

[**String**, **SensorReading**, **ThresholdUpdate**, (**String**, **Double**, **Double**)] {

*// the descriptor of the broadcast state* **private lazy val** thresholdStateDescriptor **= new MapStateDescriptor**[**String**, **Double**](

"thresholds", classOf[**String**], classOf[**Double**])

*// the keyed state handle*

**private var** lastTempState**: ValueState**[**Double**] **= \_**

**override def** open(parameters**: Configuration**)**: Unit** = {

*// create keyed state descriptor*

**val** lastTempDescriptor **= new ValueStateDescriptor**[**Double**]( "lastTemp", classOf[**Double**])

*// obtain the keyed state handle*

lastTempState **=** getRuntimeContext.getState[**Double**](lastTempDescriptor)

}

**override def** processBroadcastElement( update**: ThresholdUpdate**,

ctx**: KeyedBroadcastProcessFunction**

[**String**, **SensorReading**, **ThresholdUpdate**, (**String**, **Double**, **Double**)]**#Context**, out**: Collector**[(**String**, **Double**, **Double**)])**: Unit** = {

*// get broadcasted state handle*

**val** thresholds **=** ctx.getBroadcastState(thresholdStateDescriptor)

**if** (update.threshold != 0.0d) {

*// configure a new threshold for the sensor*

thresholds.put(update.id, update.threshold)

} **else** {

*// remove threshold for the sensor*

thresholds.remove(update.id)

}

}

**override def** processElement( reading**: SensorReading**,

readOnlyCtx**: KeyedBroadcastProcessFunction** [**String**, **SensorReading**, **ThresholdUpdate**, (**String**, **Double**, **Double**)]**#ReadOnlyContext**,

out**: Collector**[(**String**, **Double**, **Double**)])**: Unit** = {

*// get read-only broadcast state*

**val** thresholds **=** readOnlyCtx.getBroadcastState(thresholdStateDescriptor)

*// check if we have a threshold*

**if** (thresholds.contains(reading.id)) {

*// get threshold for sensor*

**val** sensorThreshold**: Double** = thresholds.get(reading.id)

*// fetch the last temperature from state*

**val** lastTemp **=** lastTempState.value()

*// check if we need to emit an alert*

**val** tempDiff **=** (reading.temperature - lastTemp).abs

**if** (tempDiff > sensorThreshold) {

*// temperature increased by more than the threshold*

out.collect((reading.id, reading.temperature, tempDiff))

}

}

*// update lastTemp state*

**this**.lastTempState.update(reading.temperature)

}

}

BroadcastProcessFunction and KeyedBroadcastProcessFunction differ from a regular CoProcessFunction because the element processing methods are not sym‐ metric. The methods, processElement() and processBroadcastElement(), are called with different context objects. Both context objects offer a method getBroad castState(MapStateDescriptor) that provides access to a broadcast state handle. However, the broadcast state handle that is returned in the processElement()

method provides read-only access to the broadcast state. This is a safety mechanism

to ensure the broadcast state holds the same information in all parallel instances. In addition, both context objects also provide access to the event-time timestamp, the current watermark, the current processing time, and the side outputs, similar to the context objects of other process functions.

The BroadcastProcessFunction and KeyedBroadcastProcess Function differ from each other as well. BroadcastProcessFunc tion does not expose a timer service to register timers and consequently does not offer an onTimer() method. Note that you should not access keyed state from the processBroadcastEle ment() method of KeyedBroadcastProcessFunction. Since the

broadcast input does not specify a key, the state backend cannot

access a keyed value and will throw an exception. Instead, the context of the KeyedBroadcastProcessFunction.processBroad castElement() method provides a method applyToKeyed State(StateDescriptor, KeyedStateFunction) to apply a

KeyedStateFunction to the value of each key in the keyed state ref‐ erenced by the StateDescriptor.

### Broadcasted Events Might Not Arrive in Deterministic Order

The order in which the broadcasted events arrive at the different parallel tasks of the broadcast state operator might differ if the operator that emits the broadcasted messages runs with a parallel‐ ism larger than 1.

Consequently, you should either ensure the value of the broadcast state does not depend on the order in which the broadcasted mes‐ sages are received or ensure the parallelism of the broadcasting operator is set to 1.

## Using the CheckpointedFunction Interface

The CheckpointedFunction interface is the lowest-level interface to specify stateful functions. It provides hooks to register and maintain keyed state and operator state

and is the only interface that gives access to operator list union state—the operator state that is fully replicated in the case of a recovery or savepoint restart.[3](#_bookmark24)

The CheckpointedFunction interface defines two methods, initializeState() and snapshotState(), which work similar to the methods of the ListCheckpointed interface for operator list state. The initializeState() method is called when a par‐ allel instance of CheckpointedFunction is created. This happens when an application

is started or when a task is restarted due to a failure. The method is called with a FunctionInitializationContext object that provides access to an Operator StateStore and a KeyedStateStore object. The state stores are responsible for regis‐

tering function state with Flink’s runtime and returning the state objects, such as ValueState, ListState, or BroadcastState. Each state is registered with a name that must be unique for the function. When a function registers state, the state store tries

to initialize the state by checking if the state backend holds state for the function reg‐ istered under the given name. If the task is restarted due to a failure or from a save‐ point, the state will be initialized from the saved data. If the application is not started from a checkpoint or savepoint, the state will be initially empty.

The snapshotState() method is called immediately before a checkpoint is taken and receives a FunctionSnapshotContext object as the parameter. FunctionSnapshotCon text gives access to the unique identifier of the checkpoint and the timestamp when the JobManager initiates the checkpoint. The purpose of the snapshotState()

method is to ensure all state objects are updated before the checkpoint is done. More‐ over, in combination with the CheckpointListener interface, the snapshotState() method can be used to consistently write data to external datastores by synchronizing

with Flink’s checkpoints.

1. See [Chapter 3](#_bookmark0) for details on how operator list union state is distributed.

[Example 7-8](#_bookmark2) shows how the CheckpointedFunction interface is used to create a func‐ tion with keyed and operator state that counts per key and operator instance how

many sensor readings exceed a specified threshold.

*Example 7-8. A function implementing the CheckpointedFunction interface*

**class HighTempCounter**(**val** threshold**: Double**)

**extends FlatMapFunction**[**SensorReading**, (**String**, **Long**, **Long**)] **with CheckpointedFunction** {

*// local variable for the operator high temperature cnt*

**var** opHighTempCnt**: Long** = 0

**var** keyedCntState**: ValueState**[**Long**] **= \_ var** opCntState**: ListState**[**Long**] **= \_**

**override def** flatMap( v**: SensorReading**,

out**: Collector**[(**String**, **Long**, **Long**)])**: Unit** = {

*// check if temperature is high*

**if** (v.temperature > threshold) {

*// update local operator high temp counter*

opHighTempCnt += 1

*// update keyed high temp counter*

**val** keyHighTempCnt **=** keyedCntState.value() + 1 keyedCntState.update(keyHighTempCnt)

*// emit new counters*

out.collect((v.id, keyHighTempCnt, opHighTempCnt))

}

}

**override def** initializeState(initContext**: FunctionInitializationContext**)**: Unit** = {

*// initialize keyed state*

**val** keyCntDescriptor **= new ValueStateDescriptor**[**Long**]("keyedCnt", classOf[**Long**]) keyedCntState **=** initContext.getKeyedStateStore.getState(keyCntDescriptor)

*// initialize operator state*

**val** opCntDescriptor **= new ListStateDescriptor**[**Long**]("opCnt", classOf[**Long**]) opCntState **=** initContext.getOperatorStateStore.getListState(opCntDescriptor)

*// initialize local variable with state*

opHighTempCnt **=** opCntState.get().asScala.sum

}

**override def** snapshotState(

snapshotContext**: FunctionSnapshotContext**)**: Unit** = {

*// update operator state with local state* opCntState.clear() opCntState.add(opHighTempCnt)

}

}

## Receiving Notifications About Completed Checkpoints

Frequent synchronization is a major reason for performance limitations in dis‐ tributed systems. Flink’s design aims to reduce synchronization points. Checkpoints are implemented based on barriers that flow with the data and therefore avoid global synchronization across all operators of an application.

Due to its checkpointing mechanism, Flink can achieve very good performance. However, another implication is that the state of an application is never in a consis‐ tent state except for the logical points in time when a checkpoint is taken. For some operators it can be important to know whether a checkpoint completed or not. For example, sink functions that aim to write data to external systems with exactly-once guarantees must only emit records that were received before a successful checkpoint to ensure the received data will not be recomputed in the case of a failure.

As discussed in [“Checkpoints, Savepoints, and State Recovery” on page 58](#_bookmark11), a check‐ point is only successful if all operator tasks successfully checkpointed their states to the checkpoint storage. Hence, only the JobManager can determine whether a check‐ point is successful or not. Operators that need to be notified about completed check‐

points can implement the CheckpointListener interface. This interface provides the notifyCheckpointComplete(long chkpntId) method, which might be called when the JobManager registers a checkpoint as completed—when all operators successfully

copied their state to the remote storage.

Note that Flink does not guarantee that the notifyCheckpointCom plete() method is called for each completed checkpoint. It is pos‐ sible that a task misses the notification. This needs to be taken into

account when implementing the interface.

# Enabling Failure Recovery for Stateful Applications

Streaming applications are supposed to run continuously and must recover from fail‐ ures, such as failing machines or processes. Most streaming applications require that failures not affect the correctness of the computed results.

In [“Checkpoints, Savepoints, and State Recovery”](#_bookmark11), we explained Flink’s mechanism to create consistent checkpoints of a stateful application, a snapshot of the state of all built-in and user-defined stateful functions at a point in time when all operators pro‐ cessed all events up to a specific position in the application’s input streams. In order to provide fault tolerance for an application, the JobManager initiates checkpoints at regular intervals.

Applications need to explicitly enable the periodic checkpointing mechanism via the

StreamExecutionEnvironment as shown in [Example 7-9](#_bookmark1).

*Example 7-9. Enabling checkpointing for an application*

**val** env **= StreamExecutionEnvironment**.getExecutionEnvironment

*// set checkpointing interval to 10 seconds (10000 milliseconds)*

env.enableCheckpointing(10000L)

The checkpointing interval is an important parameter that affects the overhead of the checkpointing mechanism during regular processing and the time it takes to recover from a failure. A shorter checkpointing interval causes higher overhead during regu‐ lar processing but can enable faster recovery because less data needs to be reproc‐ essed.

Flink provides more tuning knobs to configure the checkpointing behavior, such as the choice of consistency guarantees (exactly-once or at-least-once), the number of concurrent checkpoints, and a timeout to cancel long-running checkpoints, as well as several state backend–specific options. We discuss these options in more detail in [“Tuning Checkpointing and Recovery” on page 263](#_bookmark12).

# Ensuring the Maintainability of Stateful Applications

The state of an application that was running for several weeks can be expensive or even impossible to recompute. At the same time, long-running applications need to be maintained. Bugs need to be fixed, functionality adjusted, added, or removed, or the parallelism of the operator needs to be adjusted to account for higher or lower data rates. Therefore, it is important that application state can be migrated to a new version of the application or be redistributed to more or fewer operator tasks.

Flink features savepoints to maintain applications and their states. However, it requires that all stateful operators of the initial version of an application specify two parameters to ensure the application can be properly maintained in the future. These parameters are a unique operator identifier and the maximum parallelism (for opera‐ tors with keyed state). In the following we describe how to set these parameters.

### Operator Unique Identifiers and Maximum Parallelism Are Baked into Savepoints

The unique identifier and maximum parallelism of operators are baked into a savepoint and cannot be changed. It is not possible to start an application from a previously taken savepoint if the identi‐ fiers or the maximum parallelism of operators were changed.

Once you change operator identifiers or the maximum parallelism, you cannot start an application from a savepoint but have to start it from scratch without any state initialization.

**Specifying Unique Operator Identifiers**

Unique identifiers should be specified for every operator of an application. The iden‐ tifiers are written into a savepoint as metadata with the actual state data of an opera‐ tor. When an application is started from a savepoint, the identifiers are used to map a state in the savepoint to the corresponding operator of the started application. Save‐ point state can only be restored to an operator of a started application if their identifi‐ ers are identical.

If you do not explicitly set unique identifiers to the operators of your stateful applica‐ tion, you will face significant limitations when you have to evolve the application. We discuss the importance of unique operator identifiers and the mapping of savepoint state in more detail in [“Savepoints” on page 66](#_bookmark13).

We strongly recommend assigning unique identifiers to every operator of an applica‐ tion. You can set the identifier with the uid() method as shown in [Example 7-10](#_bookmark6).

*Example 7-10. Setting a unique identifier for an operator*

**val** alerts**: DataStream**[(**String**, **Double**, **Double**)] **=** keyedSensorData

.flatMap(**new TemperatureAlertFunction**(1.1))

.uid("TempAlert")

## Defining the Maximum Parallelism of Keyed State Operators

The maximum parallelism parameter of an operator defines the number of key groups into which the keyed state of the operator is split. The number of key groups limits the maximum number of parallel tasks to which keyed state can be scaled. [“Scaling Stateful Operators” on page 56](#_bookmark9) discusses key groups and how keyed state is scaled out and in. The maximum parallelism can be set for all operators of an applica‐

tion via the StreamExecutionEnvironment or per operator using the setMaxParallel ism() method as shown in [Example 7-11](#_bookmark14).

*Example 7-11. Setting the maximum parallelism of operators*

**val** env **= StreamExecutionEnvironment**.getExecutionEnvironment

*// set the maximum parallelism for this application*

env.setMaxParallelism(512)

**val** alerts**: DataStream**[(**String**, **Double**, **Double**)] **=** keyedSensorData

.flatMap(**new TemperatureAlertFunction**(1.1))

*// set the maximum parallelism for this operator and*

*// override the application-wide value*

.setMaxParallelism(1024)

The default maximum parallelism of an operator depends on the operator’s parallel‐ ism in the application’s first version:

* + If the parallelism is less than or equal to 128, the maximum parallelism is 128.
  + If the operator’s parallelism is larger than 128, the maximum parallelism is com‐ puted as the minimum of nextPowerOfTwo(parallelism + (parallelism / 2)) and 2^15.

# Performance and Robustness of Stateful Applications

The way operators interact with state has implications on the robustness and perfor‐ mance of an application. There are several aspects that affect the behavior of an appli‐ cation such as the choice of the state backend that locally maintains the state and performs checkpoints, the configuration of the checkpointing algorithm, and the size of the application’s state. In this section, we discuss aspects that need to be taken into account to ensure robust execution behavior and consistent performance of long- running applications.

## Choosing a State Backend

In [“State Backends” on page 55](#_bookmark17), we explained that Flink maintains application state in a state backend. The state backend is responsible for storing the local state of each task instance and persisting it to remote storage when a checkpoint is taken. Because local state can be maintained and checkpointed in different ways, state backends are pluggable—two applications can use different state backend implementations to maintain their states. The choice of the state backend has implications on the robust‐ ness and performance of a stateful application. Each state backend provides imple‐

mentations for the different state primitives, such as ValueState, ListState, and

MapState.

Currently, Flink offers three state backends, the MemoryStateBackend, the FsState Backend, and the RocksDBStateBackend:

* + MemoryStateBackend stores state as regular objects on the heap of the TaskMan‐ ager JVM process. For example, MapState is backed by a Java HashMap object. While this approach provides very low latencies to read or write state, it has

implications on the robustness of an application. If the state of a task instance grows too large, the JVM and all task instances running on it can be killed due to

an OutOfMemoryError. Moreover, this approach can suffer from garbage collec‐

tion pauses because it puts many long-lived objects on the heap. When a check‐ point is taken, MemoryStateBackend sends the state to the JobManager, which stores it in its heap memory. Hence, the total state of an application must fit into

the JobManager’s memory. Since its memory is volatile, the state is lost in case of a JobManager failure. Due to these limitations, MemoryStateBackend is only rec‐ ommended for development and debugging purposes.

* + FsStateBackend stores the local state on the TaskManager’s JVM heap, just like MemoryStateBackend. However, instead of checkpointing the state to the JobMa‐ nager’s volatile memory, FsStateBackend writes the state to a remote and persis‐ tent file system. Hence, FsStateBackend provides in-memory speed for local

accesses and fault tolerance in the case of failures. However, it is limited by the

size of the TaskManager memory and might suffer from garbage collection pau‐ ses.

* + RocksDBStateBackend stores all state into local RocksDB instances. RocksDB is an embedded key-value store that persists data to the local disk. In order to read

and write data from and to RocksDB, it needs to be de/serialized. The RocksDBStateBackend also checkpoints the state to a remote and persistent file system. Because it writes data to disk and supports incremental checkpoints

(more on this in [“Checkpoints, Savepoints, and State Recovery” on page 58](#_bookmark11)), RocksDBStateBackend is a good choice for applications with very large state. Users have reported applications with state sizes of multiple terabytes leveraging RocksDBStateBackend. However, reading and writing data to disk and the over‐ head of de/serializing objects result in lower read and write performance com‐

pared to maintaining state on the heap.

Since StateBackend is a public interface, it is also possible to implement a custom state backend. [Example 7-12](#_bookmark15) shows how to configure a state backend (here, RocksDBStateBackend) for an application and all its stateful functions.

*Example 7-12. Configuring RocksDBStateBackend for an application*

**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)

We discuss in [“Tuning Checkpointing and Recovery” on page 263](#_bookmark12) how to use and configure state backends in your application.

## Choosing a State Primitive

The performance of a stateful operator (built-in or user-defined) depends on several aspects, including the data types of the state, the state backend of the application, and the chosen state primitives.

For state backends that de/serialize state objects when reading or writing, such as RocksDBStateBackend, the choice of the state primitive (ValueState, ListState, or MapState) can have a major impact on the performance of an application. For

instance, ValueState is completely deserialized when it is accessed and serialized when it is updated. The ListState implementation of RocksDBStateBackend deseri‐ alizes all list entries before constructing Iterable to read the values. However, adding a single value to ListState—appending it to the end of the list—is a cheap operation because only the appended value is serialized. MapState of RocksDBStateBackend

allows reading and writing values per key—only those keys and values are de/serial‐ ized that are read or written. When iterating over the entry set of MapState, the seri‐ alized entries are prefetched from RocksDB and only deserialized when a key or value

is actually accessed.

For example, with RocksDBStateBackend it is more efficient to use MapState[X, Y] instead of ValueState[HashMap[X, Y]]. ListState[X] has an advantage over Value State[List[X]] if elements are frequently appended to the list and the elements of the list are less frequently accessed.

Another good practice is to update state only once per function call. Since check‐ points are synchronized with function invocations, multiple state updates do not pro‐ vide any benefits but can cause additional serialization overhead when updating state several times in a single function call.

## Preventing Leaking State

Streaming applications are often designed to run continuously for months or years. If the state of an application is continuously increasing, it will at some point grow too large and kill the application unless action is taken to scale the application to more resources. In order to prevent increasing resource consumption of an application over time, it is important that the size of the operator state be controlled. Since the handling of state directly affects the semantics of an operator, Flink cannot automati‐ cally clean up state and free storage. Instead, all stateful operators must control the size of their state and have to ensure it is not infinitely growing.

A common reason for growing state is keyed state on an evolving key domain. In this scenario, a stateful function receives records with keys that are only active for a cer‐ tain period of time and are never received after that. A typical example is a stream of click events where clicks have a session id attribute that expires after some time. In such a case, a function with keyed state would accumulate state for more and more

keys. As the key space evolves, the state of expired keys becomes stale and useless. A solution for this problem is to remove the state of expired keys. However, a function with keyed state can only access the state of a key if it received a record with that key. In many cases, a function does not know if a record will be the last one for a key. Hence, it will not be able to evict the state for the key because it might receive another record for the key.

This problem does not only exist for custom stateful functions but also for some of the built-in operators of the DataStream API. For example, computing running

aggregates on a KeyedStream, either with the built-in aggregations functions such as min, max, sum, minBy, or maxBy or with a custom ReduceFunction or AggregateFunc tion, keeps the state for each key and never discards it. Consequently, these functions should only be used if the key values are from a constant and bounded domain. Other

examples are windows with count-based triggers, which process and clean their state when a certain number of records has been received. Windows with time-based trig‐ gers (both processing time and event time) are not affected by this because they trig‐ ger and purge their state based on time.

This means that you should take application requirements and the properties of its input data, such as key domain, into account when designing and implementing stateful operators. If your application requires keyed state for a moving key domain, it should ensure the state of keys is cleared when it is not needed anymore. This can be done by registering timers for a point of time in the future.[4](#_bookmark24) Similar to state, timers are registered in the context of the currently active key. When the timer fires, a call‐ back method is called and the context of timer’s key is loaded. Hence, the callback method has full access to the key’s state and can also clear it. The functions that offer

support to register timers are the Trigger interface for windows and the process function. Both were covered in [Chapter 6](#_bookmark0).

[Example 7-13](#_bookmark18) shows a KeyedProcessFunction that compares two subsequent tem‐ perature measurements and raises an alert if the difference is greater than a certain

threshold. This is the same use case as in the keyed state example before, but the KeyedProcessFunction also clears the state for keys (i.e., sensors) that have not pro‐ vided any new temperature measurements within one hour of event time.

*Example 7-13. A stateful KeyedProcessFunction that cleans its state*

**class SelfCleaningTemperatureAlertFunction**(**val** threshold**: Double**)

**extends KeyedProcessFunction**[**String**, **SensorReading**, (**String**, **Double**, **Double**)] {

*// the keyed state handle for the last temperature*

**private var** lastTempState**: ValueState**[**Double**] **= \_**

1. Timers can be based on event time or processing time.

*// the keyed state handle for the last registered timer*

**private var** lastTimerState**: ValueState**[**Long**] **= \_**

**override def** open(parameters**: Configuration**)**: Unit** = {

*// register state for last temperature*

**val** lastTempDesc **= new ValueStateDescriptor**[**Double**]("lastTemp", classOf[**Double**]) lastTempState **=** getRuntimeContext.getState[**Double**](lastTempDescriptor)

*// register state for last timer*

**val** lastTimerDesc **= new ValueStateDescriptor**[**Long**]("lastTimer", classOf[**Long**]) lastTimerState **=** getRuntimeContext.getState(timestampDescriptor)

}

**override def** processElement( reading**: SensorReading**, ctx**: KeyedProcessFunction**

[**String**, **SensorReading**, (**String**, **Double**, **Double**)]**#Context**, out**: Collector**[(**String**, **Double**, **Double**)])**: Unit** = {

*// compute timestamp of new clean up timer as record timestamp + one hour*

**val** newTimer **=** ctx.timestamp() + (3600 \* 1000)

*// get timestamp of current timer*

**val** curTimer **=** lastTimerState.value()

*// delete previous timer and register new timer* ctx.timerService().deleteEventTimeTimer(curTimer) ctx.timerService().registerEventTimeTimer(newTimer)

*// update timer timestamp state*

lastTimerState.update(newTimer)

*// fetch the last temperature from state*

**val** lastTemp **=** lastTempState.value()

*// check if we need to emit an alert*

**val** tempDiff **=** (reading.temperature - lastTemp).abs

**if** (tempDiff > threshold) {

*// temperature increased by more than the threshold*

out.collect((reading.id, reading.temperature, tempDiff))

}

*// update lastTemp state*

**this**.lastTempState.update(reading.temperature)

}

**override def** onTimer( timestamp**: Long**,

ctx**: KeyedProcessFunction**

[**String**, **SensorReading**, (**String**, **Double**, **Double**)]**#OnTimerContext**, out**: Collector**[(**String**, **Double**, **Double**)])**: Unit** = {

*// clear all state for the key* lastTempState.clear() lastTimerState.clear()

}

}

The state-cleaning mechanism implemented by the above KeyedProcessFunction works as follows. For each input event, the processElement() method is called. Before comparing the temperature measurements and updating the last temperature,

the method updates the clean-up timer by deleting the previous timer and registering a new one. The clean-up time is computed by adding one hour to the timestamp of the current record. In order to be able to delete the currently registered timer, its

timestamp is stored in an additional ValueState[Long] called lastTimerState. After

that, the method compares the temperatures, possibly emits an alert, and updates its

state.

Since our KeyedProcessFunction always updates the registered timer by deleting the current timer and registering a new one, only a single timer is registered per key. Once that timer fires, the onTimer() method is called. The method clears all state associated with the key, the last temperature and the last timer state.

# Evolving Stateful Applications

It is often necessary to fix a bug or to evolve the business logic of a long-running stateful streaming application. Consequently, a running application needs to be replaced by an updated version usually without losing the state of the application.

Flink supports such updates by taking a savepoint of a running application, stopping it, and starting a new version of the application from the savepoint.[5](#_bookmark22) However, updat‐ ing an application while preserving its state is only possible for certain application changes—the original application and its new version need to be savepoint compati‐ ble. In the following, we explain how applications can be evolved while preserving savepoint compatibility.

In [“Savepoints”](#_bookmark13), we explained that each state in a savepoint can be addressed by a composite identifier consisting of a unique operator identifier and the state name declared by the state descriptor.

### Implement Your Applications With Evolution in Mind

It is important to understand that the initial design of an applica‐ tion determines if and how it can be modified later on in a savepoint-compatible way. Many changes will not be possible if the original version was not designed with updates in mind. Assigning unique identifiers to operators is mandatory for most application changes.

1. [Chapter 10](#_bookmark0) explains how to take savepoints of running applications and how to start a new application from an existing savepoint.

When an application is started from a savepoint, the operators of the started applica‐ tion are initialized by looking up the corresponding states from the savepoint using operator identifiers and state names. From a savepoint-compatibility point of view this means an application can be evolved in three ways:

1. Updating or extending the logic of an application without changing or removing an existing state. This includes adding of stateful or stateless operators to the application.
2. Removing a state from the application.
3. Modifying the state of an existing operator by changing the state primitive or data type of the state.

In the following sections, we discuss these three cases.

**Updating an Application without Modifying Existing State**

If an application is updated without removing or changing existing state, it is always savepoint compatible and can be started from a savepoint of an earlier version.

If you add a new stateful operator to the application or a new state to an existing operator, the state will be initialized as empty when the application is started from a savepoint.

### Changing the Input Data Type of Built-in Stateful Operators

Note that changing the input data type of built-in stateful opera‐ tors, such as window aggregation, time-based joins, or asyncronous functions, often modifies the type of their internal state. Therefore, such changes are not safepoint compatible even though they look unobtrusive.

## Removing State from an Application

Instead of adding new states to an application, you might also want to adjust an appli‐ cation by removing state—either by removing a complete stateful operator or just a state from a function. When the new version of the application is started from a save‐ point of the previous version, the savepoint contains state that cannot be mapped to the restarted application. This is also the case if the unique identifier of an operator or the name of a state was changed.

By default, Flink will not start applications that do not restore all states that are con‐ tained in a savepoint to avoid losing the state in the savepoint. However, it is possible to disable this safety check as described in [“Running and Managing Streaming Appli‐](#_bookmark19)

[cations” on page 245](#_bookmark19). Hence, it is not difficult to update an application by removing stateful operators or state from an existing operator.

## Modifying the State of an Operator

While adding or removing state from an application is rather easy and does not affect savepoint compatibility, modifying the state of an existing operator is more involved. There are two ways state can be modified:

* + By changing the data type of a state, such as changing a ValueState[Int] to a

ValueState[Double]

* + By changing the type of a state primitive, as for example by changing a Value State[List[String]] to a ListState[String]

Changing the data type of a state is possible in a few specific cases. However, Flink currently does not support changing the primitive (or structure) of a state. There are some ideas to support this case by offering an offline tool to convert savepoints. How‐ ever, as of Flink 1.7 no such tool exists. In the following we focus on changing the data type of a state.

In order to grasp the problem of modifying the data type of a state, we have to under‐ stand how state data is represented within a savepoint. A savepoint consists mainly of serialized state data. The serializers that convert the state JVM objects into bytes are generated and configured by Flink’s type system. This conversion is based on the data

type of the state. For example, if you have a ValueState[String], Flink’s type system generates a StringSerializer to convert String objects into bytes. The serializer is also used to convert the raw bytes back into JVM objects. Depending on whether the

state backend stores the data serialized (like the RocksDBStateBackend) or as objects on the heap (like the FSStateBackend), this happens when the state is read by a func‐ tion or when an application is restarted from a savepoint.

Since Flink’s type system generates serializers depending on the data type of a state, the serializers are likely to change when the data type of a state changes. For example,

if you changed the ValueState[String] to a ValueState[Double], Flink would cre‐ ate a DoubleSerializer to access the state. It is not surprising that using a DoubleSer ializer to deserialize the binary data generated by serializing a String with a StringSerializer will fail. Hence, changing the data type of a state is only supported

in very specific cases.

In Flink 1.7, changing the data type of a state is supported if the data type was defined as an Apache Avro type and if the new data type is also an Avro type that was evolved from the original type according to Avro’s schema evolution rules. Flink’s type system will automatically generate serializers that can read previous versions of the data type.

State evolution and migration is an important topic in the Flink community and receives a lot of attention. You can expect improved support for these scenarios in future versions of Apache Flink. Despite all these efforts, we recommend always dou‐ ble checking if an application can be evolved as planned before putting it into pro‐ duction.

# Queryable State

Many stream processing applications need to share their results with other applica‐ tions. A common pattern is to write results into a database or key-value store and have other applications retrieve the result from that datastore. Such an architecture implies that a separate system needs to be set up and maintained, which can be a major effort, especially if this needs to be a distributed system as well.

Apache Flink features queryable state to address use cases that usually would require an external datastore to share data. In Flink, any keyed state can be exposed to exter‐ nal applications as queryable state and act as a read-only key-value store. The stateful streaming application processes events as usual and stores and updates its intermedi‐ ate or final results in a queryable state. External applications can request the state for a key while the streaming application is running.

Note that only key point queries are supported. It is not possible to request key ranges or even run more complex queries.

Queryable state does not address all use cases that require an external datastore. For example, the queryable state is only accessible while the application is running. It is not accessible while the application is restarted due to an error, for rescaling the application, or to migrate it to another cluster. However, it makes many applications much easier to realize, such as real-time dashboards or other monitoring applica‐ tions.

In the following, we discuss the architecture of Flink’s queryable state service and explain how streaming applications can expose queryable state and external applica‐ tions can query it.

## Architecture and Enabling Queryable State

Flink’s queryable state service consists of three processes:

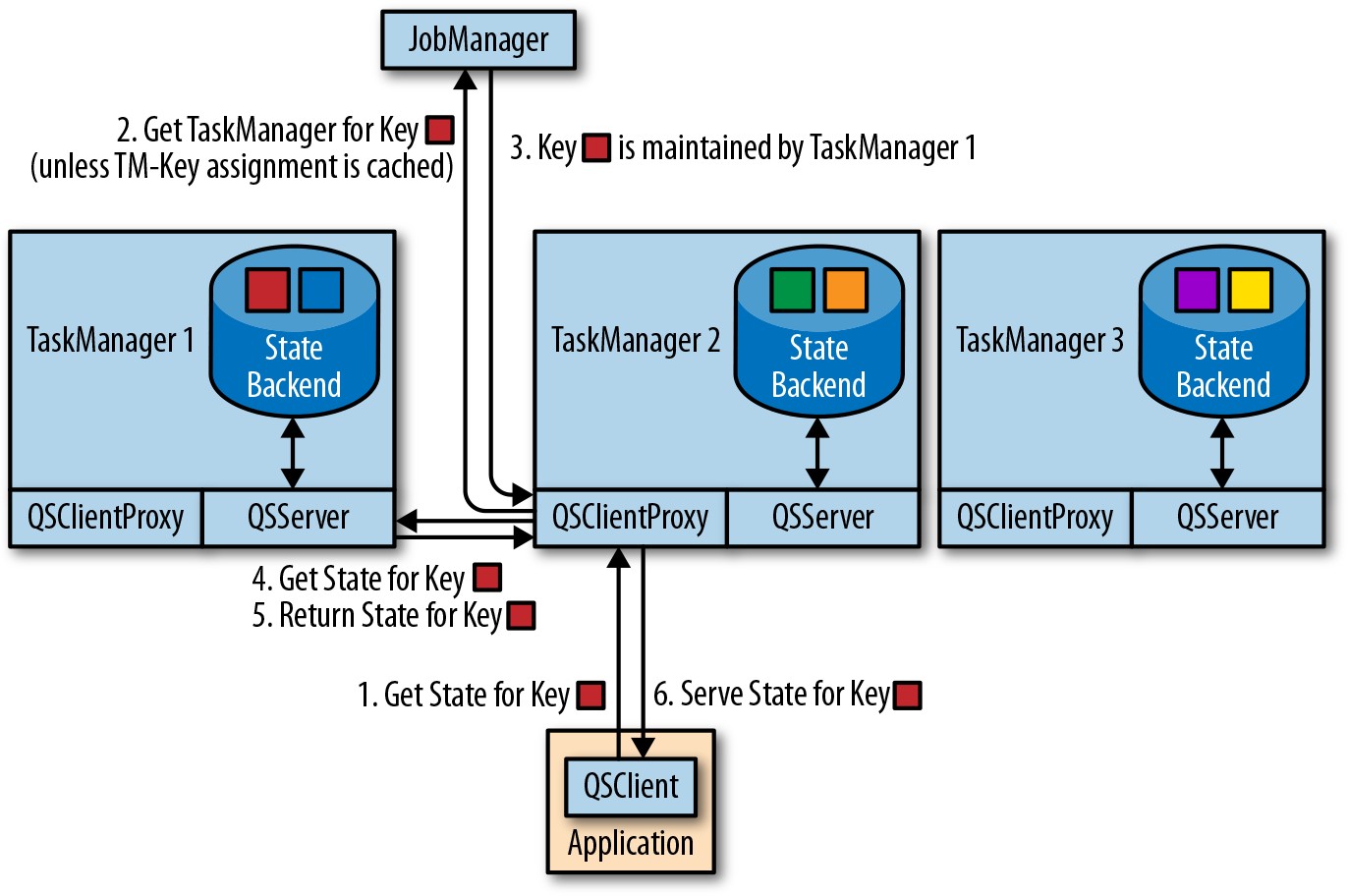
* + The QueryableStateClient is used by an external application to submit queries and retrieve results.
  + The QueryableStateClientProxy accepts and serves client requests. Each Task‐ Manager runs a client proxy. Since keyed state is distributed across all parallel

instances of an operator, the proxy needs to identify the TaskManager that main‐ tains the state for the requested key. This information is requested from the Job‐ Manager that manages the key group assignment, and is cached once received.[6](#_bookmark24) The client proxy retrieves the state from the state server of the respective Task‐ Manager and serves the result to the client.

* + The QueryableStateServer serves the requests of a client proxy. Each TaskMan‐ ager runs a state server that fetches the state of a queried key from the local state

backend and returns it to the requesting client proxy.

[Figure 7-1](#_bookmark7) shows the architecture of the queryable state service.



*Figure 7-1. Architecture of Flink’s queryable state service*

In order to enable the queryable state service in a Flink setup—to start client proxy and server threads within the TaskManagers—you need to add the *flink-queryable- state-runtime* JAR file to the classpath of the TaskManager process. This is done by copying it from the *./opt* folder of your installation into the *./lib* folder. When the JAR file is in the classpath, the queryable state threads are automatically started and can

1. Key groups are discussed in [Chapter 3](#_bookmark0).

serve requests of the queryable state client. When properly configured, you will find the following log message in the TaskManager logs:

Started the Queryable State Proxy Server @ …

The ports used by the client proxy and server and additional parameters can be con‐ figured in the *./conf/flink-conf.yaml* file.

## Exposing Queryable State

Implementing a streaming application with queryable state is easy. All you have to do is define a function with keyed state and make the state queryable by calling the set Queryable(String) method on the StateDescriptor before obtaining the state han‐ dle. [Example 7-14](#_bookmark8) shows how to make lastTempState queryable to illustrate the usage of the keyed state.

*Example 7-14. Configuring keyed state to be queryable*

**override def** open(parameters**: Configuration**)**: Unit** = {

*// create state descriptor*

**val** lastTempDescriptor **=**

**new ValueStateDescriptor**[**Double**]("lastTemp", classOf[**Double**])

*// enable queryable state and set its external identifier*

lastTempDescriptor.setQueryable("lastTemperature")

*// obtain the state handle*

lastTempState **=** getRuntimeContext

.getState[**Double**](lastTempDescriptor)

}

The external identifier that is passed with the setQueryable() method can be freely chosen and is only used to configure the queryable state client.

In addition to the generic way of enabling queries on any type of keyed state, Flink also offers shortcuts to define stream sinks that store the events of a stream in a queryable state. [Example 7-15](#_bookmark20) shows how to use a queryable state sink.

*Example 7-15. Writing a DataStream into a queryable state sink*

**val** tenSecsMaxTemps**: DataStream**[(**String**, **Double**)] **=** sensorData

*// project to sensor id and temperature*

.map(r **=>** (r.id, r.temperature))

*// compute every 10 seconds the max temperature per sensor*

.keyBy(**\_**.\_1)

.timeWindow(**Time**.seconds(10))

.max(1)

*// store max temperature of the last 10 secs for each sensor*

*// in a queryable state*

tenSecsMaxTemps

*// key by sensor id*

.keyBy(**\_**.\_1)

.asQueryableState("maxTemperature")

The asQueryableState() method appends a queryable state sink to the stream. The type of the queryable state is ValueState, which holds values of the type of the input stream—our example (String, Double). For each received record, the queryable state sink upserts the record into ValueState, so that the latest event per key is always

stored.

An application with a function that has a queryable state is executed just like any other application. You only have to ensure that the TaskManagers are configured to start their queryable state services as discussed in the previous section.

## Querying State from External Applications

Any JVM-based application can query the queryable state of a running Flink applica‐ tion by using QueryableStateClient. This class is provided by the flink- queryable-state-client-java dependency, which you can add to your project as

follows:

**<dependency>**

**<groupid>**org.apache.flink**</groupid>**

**<artifactid>**flink-queryable-state-client-java\_2.12**</artifactid>**

**<version>**1.7.1**</version>**

**</dependency>**

The QueryableStateClient is initialized with the hostname of any TaskManager and the port on which the queryable state client proxy is listening. By default, the client

proxy listens on port 9067, but the port can be configured in the *./conf/flink- conf.yaml* file:

**val** client**: QueryableStateClient** =

**new QueryableStateClient**(tmHostname, proxyPort)

Once you obtain a state client, you can query the state of an application by calling the getKvState() method. The method takes several parameters, such as the JobID of the running application, the state identifier, the key for which the state should be

fetched, the TypeInformation for the key, and the StateDescriptor of the queried state. The JobID can be obtained via the REST API, the Web UI, or the log files. The getKvState() method returns a CompletableFuture[S] where S is the type of the state (e.g., ValueState[\_] or MapState[\_, \_]). Hence, the client can send out multi‐

ple asynchronous queries and wait for their results. [Example 7-16](#_bookmark1) shows a simple

console dashboard that queries the queryable state of the application shown in the previous section.

*Example 7-16. A simple dashboard application that queries the state of a Flink application*

**object TemperatureDashboard** {

*// assume local setup and TM runs on same machine as client*

**val** proxyHost **=** "127.0.0.1"

**val** proxyPort **=** 9069

*// jobId of running QueryableStateJob*

*// can be looked up in logs of running job or the web UI*

**val** jobId **=** "d2447b1a5e0d952c372064c886d2220a"

*// how many sensors to query*

**val** numSensors **=** 5

*// how often to query the state*

**val** refreshInterval **=** 10000

**def** main(args**: Array**[**String**])**: Unit** = {

*// configure client with host and port of queryable state proxy*

**val** client **= new QueryableStateClient**(proxyHost, proxyPort)

**val** futures **= new Array**[

**CompletableFuture**[**ValueState**[(**String**, **Double**)]]](numSensors) **val** results **= new Array**[**Double**](numSensors)

*// print header line of dashboard table*

**val** header **=**

(**for** (i **<-** 0 until numSensors) **yield** "sensor\_" + (i + 1))

.mkString("\t| ") println(header)

*// loop forever*

**while** (**true**) {

*// send out async queries*

**for** (i **<-** 0 until numSensors) {

futures(i) **=** queryState("sensor\_" + (i + 1), client)

}

*// wait for results*

**for** (i **<-** 0 until numSensors) {

results(i) **=** futures(i).get().value().\_2

}

*// print result*

**val** line **=** results.map(t **=>** f"$t%1.3f").mkString("\t| ") println(line)

*// wait to send out next queries*

**Thread**.sleep(refreshInterval)

}

client.shutdownAndWait()

}

**def** queryState( key**: String**,

client**: QueryableStateClient**)

**: CompletableFuture**[**ValueState**[(**String**, **Double**)]] **=** {

client

.getKvState[**String**, **ValueState**[(**String**, **Double**)], (**String**, **Double**)]( **JobID**.fromHexString(jobId),

"maxTemperature", key, **Types**.**STRING**,

**new ValueStateDescriptor**[(**String**, **Double**)]( "", *// state name not relevant here* **Types**.**TUPLE**[(**String**, **Double**)]))

}

}

In order to run the example, you have to start the streaming application with the queryable state first. Once it is running, look for the JobID in the log file or the web UI; set the JobID in the code of the dashboard and run it as well. The dashboard will

then start querying the state of the running streaming application.

# Summary

Just about every nontrivial streaming application is stateful. The DataStream API provides powerful yet easy-to-use tools to access and maintain operator state. It offers different types of state primitives and supports pluggable state backends. While developers have lots of flexibility to interact with state, Flink’s runtime manages tera‐ bytes of state and ensures exactly-once semantics in case of failure. The combination of time-based computations as discussed in [Chapter 6](#_bookmark0) and scalable state management empowers developers to realize sophisticated streaming applications. Queryable state is an easy-to-use feature and can save you the effort of setting up and maintaining a database or key-value store to expose the results of a streaming application to external applications.