Stateful Stream Processing

NOTE: This document is a design draft and does not reflect the state of a stable release, but the work in progress on the current master, and future considerations.

Page contents

- State in streaming programs
- Fault-tolerance in Flink
- State access patterns
  - Local (non-partitioned) state
  - Partitioned state
  - Out-of-core state (Not yet available)
- State requirements
  - State flexibility for applications
  - Job scalability with correctness
  - Scalability to large state sizes
  - Efficient fault tolerance
  - Flexible state backends

State in streaming programs

More complex streaming pipelines generally need to keep some sort of operator state in order to execute the application logic. Examples include, keeping some aggregation or summary of the received elements, or we can imagine more complex states such as keeping a state-machine for detecting patterns for fraudulent financial transactions or holding a model for some machine learning application. While in all mentioned cases we keep some kind of summary of the input history, the concrete requirements vary greatly from one stateful application to another.

This document is intended to serve as a guide for fault-tolerant stateful stream processing in Flink and other streaming systems, by identifying some common usage patterns and requirements for implementing stateful operators in streaming systems.

Fault-tolerance in Flink

While this is not the focus of this document, it is important to introduce the basic mechanism behind fault-tolerance in Flink streaming. The algorithm used by Flink is designed to support exactly-once guarantees for stateful streaming programs (regardless of the actual state representation). The mechanism ensures that even in the presence of failures, the program’s state will eventually reflect every record from the data stream exactly once.

To achieve this functionality Flink periodically draws consistent global snapshots of the execution topology in a highly efficient manner. You can read more about the checkpointing mechanism in the documentation.

State access patterns

To design a properly functioning, scalable state representation we need to understand the different state access and handling patterns that applications need in streaming programs. We can distinguish three main state types (access patterns): local state, partitioned state, out-of-core state.

Local (non-partitioned) state
Local or non-partitioned state is the simplest form of operator state which represent the current state of a specific operator instance in a parallel streaming operator. Local states stored at different operator instances do not interact with each other. For instance if we have a mapper with parallelism of 10 that means each parallel instance holds it's own local state. An important thing to note here is that state updates reflected in a particular local state will depend only on the input of that specific operator instance. As there is no strict control over the input which each operator instance processes (only via stream partitioning), the operator itself needs to deal with this non-deterministic nature of the inputs and this also limits expressivity.

Typical usage of non-partitioned state includes source operators (storing offsets), any global aggregation/summary or analysis operators where the local results will be merged into a global result afterwards.

Scaling out non-partitioned state can in most cases be done by just starting a new operator instance with a blank state, or a user supplied splitting function can be used to split the state of an existing instance. For reducing job parallelism the user should generally provide a function that can merge 2 local states to maintain correct results.

### Local state example 1

```scala
val source: DataStream[MyType] = ...
val everyFifth: DataStream[MyType] = source.filterWithState((in, c: Option[Int]) =>
  c match {
    case Some(c) => (c == 0, if (c == 0) Some(4) else Some(c-1))
    case None => (true, Some(4))
  })
```

### Local state example 2

```scala
val env: StreamExecutionEnvironment = ...
val stream = env.addSource(new SourceFunction[Int]() {
  // store current offset as operator state
  var offset: OperatorState[Int] = _
  var isRunning = false

  override def open(config: Configuration): Unit = {
    // get non-partitioned state from context with default value of 0
    offset = getRuntimeContext().getState("offset", 0, false)
  }

  override def run(ctx: SourceContext[Int]): Int = {
    isRunning = true
    while (isRunning) {
      // output the current offset then increment
      val currOffset = offset.value
      ctx.collect(currOffset)
      offset.update(currOffset + 1)
    }
  }

  override def cancel(): Unit = { isRunning = false }
})
```

Alternatively, the user can also use the Checkpointed interface for implementing local state functionality. This interface gives more fine-grained control over the checkpoint/restore process:
Checkpointed interface example

class CheckpointedSource extends SourceFunction[Int] with Checkpointed[Int] {
    // we keep state as a regular Int now to use it with the Checkpointed interface
    var offset: Int = 0
    var isRunning = false

    override def run(ctx: SourceContext[Int]): Int = {
        isRunning = true
        while (isRunning) {
            ctx.collect(offset)
            offset += 1
        }
    }

    // we simply return the offset as the state (and restore it accordingly)
    override def snapshotState(checkpointID: Long, timeStamp: Long): Int = offset
    override def restoreState(state: Int): Unit = {offset = state}
    override def cancel(): Unit = { isRunning = false }
}

Partitioned state

Partitioned state provides a representation for operator states that are tied to partitions (keys) of the operator input. An independent state is maintained for each partition and operator states for different keys don’t interact. Partitioning is done by some user defined key which is extracted from each input and inputs with the same key share the state. Big advantage of partitioned state is both expressivity and trivial scalability.

In case of partitioned state, the operator instances need not to be aware of the partitioning of the inputs as they can only access the state for the current input and the system guarantees proper partitioning by the selected key. This makes the implementation of per-key operators very simple. Operations using partitioned state can also benefit from the partial ordering guarantees that the flink runtime provides, to implement deterministic behaviour. Furthermore partitioned state can easily be scaled automatically by moving some partitions to new operators and changing the input partitioning.

Typical usage includes any per key stateful operation, such as group aggregates/summaries/window buffers, or analysis over several distinct groups in the stream for instance pattern detection.

Partitioned state is only accessible from operators applied on KeyedDataStreams (the state partitioning will be done by the key in the stream):

Partitioned state example 1

val source: DataStream[Long] = ...
val meanByParity : DataStream[Double] = source.keyBy(_ % 2).mapWithState{
    (in, state: Option[(Int, Long)]) => state match {
        case Some((count, sum)) => ((sum + in).toDouble / (count + 1), Some((count + 1, sum + in)))
        case None => (in, Some(1, in))
    }
}

Or using the Java interfaces and OperatorStates:
Partitioned state example 2

val stream: DataStream[String] = ...
val keyed: KeyedDataStream[String] = stream.keyBy(x => x)
val  = keyed.map(new RichMapFunction[String, (String, Int)]() {

  val countPerID: OperatorState[Int] = _

  override def open(config: Configuration): Unit = {
    // the true flag marks the state as partitioned
    countPerID = getRuntimeContext().getState("count", 0, true)
  }

  override def map(in: String): (String, Int) = {
    // countPerID holds the state for the current input
    val countSoFar= countPerID.value + 1
    countPerID.update(countSoFar)
    (in, countSoFar)
  }
})

Out-of-core state (Not yet available)
We can think of out-of-core state as an extension of partitioned state, where the state is kept outside of the system in an external storage layer (in some key-value store). In contrast with partitioned state, here the states are not stored and checkpointed with other operator states, but are read from and updated externally (we will of course use some sort of caching to increase performance). There are several reasons why one would use out-of-core state in streaming applications:

1. Storing large states that don't fit in memory (with the restriction that a state for one key needs to fit)
2. Making operator states available for external systems

While 1. is important as well and we will touch this issue later, we will focus on 2. here. The problem that arises here is that we want to maintain a consistent view of the streaming state from the outside despite possible node failures during processing. While our fault tolerance mechanism can deal with consistence within the streaming job, we need to extend it to handle out-of-core state consistently.

The main problem here is that we want to simulate a global transaction (a transaction containing all the state updates, a change-log, since the last checkpoint) to the external storage without actually having such a mechanism. This is hard because we need to guarantee that once we have written some update to the external storage we cannot roll-back in case if failure, so we need to make sure that either all states are written or none (we can't even start writing in that case).

API

We could use the OperatorState interface with out-of-core states in a similar way as we would do in case of partitioned states. This assumes that we restrict the usage to KeyedDataStream, which is a fair assumption if we want to have a consistent behaviour.
State requirements

The state interfaces and implementations need to satisfy certain requirements in order to efficiently scale to large data volumes while keeping flexibility for application developers:

1. State flexibility for applications
2. Job scalability with correctness
3. Scalability to large state sizes
4. Efficient fault tolerance
5. Flexible state backends

We will go through each requirement in this section in detail and show how Flink tackles them.

State flexibility for applications

This is probably the only requirement that is hard to capture very precisely. What it means in practice is that the provided state interfaces should cover most of the stateful use cases that can be expected from a distributed system.

To achieve this Flink provides multiple ways to declare and access operator state, giving full flexibility to the applications:

OperatorState interface
The OperatorState interface encapsulates both partitioned and non-partitioned state access in streaming operators. In case of local state access the value returned by `state.value()` is the same for all inputs in an operator system, while in case of partitioned state access the state returned is the state for the key of the current input.

The simple thing about using OperatorStates is that local and partitioned state access happens exactly the same way, and the user only needs to worry about declaring the state access pattern when retrieving the OperatorState instance.

OperatorStates can be used in any RichFunction, and is accessible from the runtime context. The user can retrieve an OperatorState from the RuntimeContext using a string identifier (to allow multiple states) and a default value. In addition, the user can also provide a StateCheckpointer instance for fine grained control over the checkpointing of the state. By default the user states will be serialized using default Java serialization.

Another alternative way to declare stateful operators is by implementing the Checkpointed interface by the user function. The system automatically detects operators that implement this interface and will draw (and restore) state snapshots using the ‘snapshotState’ and ‘restoreState’ methods.

This interface gives ultimate flexibility for the user code to manage state checkpointing and can be used to implement arbitrary state access logic but it comes with the burden of having to take care of partitioning as well in a way that it can potentially handle operator scaling.

This approach might be preferred in cases when the user needs to execute some additional application logic during snapshotting. Of course the two interfaces can be used at the same time (even in the same operator).

Job scalability with correctness

It should be possible to scale out (or in) stateful streaming applications by increasing (or decreasing) the parallelism of the topology while maintaining correct results. This simply means that parallelism should not affect the results of stateful computations.

Arbitrary scaling (as long as we have enough memory) can be achieved when using partitioned or out-of-core states, by repartitioning the DataStream, and redistributing the states according to the new partitioning scheme. To make this efficient partitioned operator states are checkpointed by key, so we can repartition the states (by the handles) without having to retrieve them from the persistent storage. To execute a state repartitioning we stop the streaming job, repartition the states of the last global snapshot, then restart the topology from there.
Maintaining correctness with local state depends on the user application logic. New operator instances can start from the default (empty state). In this case, reducing operator parallelism is an issue as there is no way to merge arbitrary states. To allow scaling in, the user should be able to pass a function that will merge different local states.

### Scalability to large state sizes

User state should be allowed to grow large without out-of-memory exceptions and significant slowdown of the processing/recovery speed. Without any mechanism to handle this several bad things can happen: large user states fill up the heap causing large GC overhead and eventual out-of-memory exceptions, every state snapshot will take considerable longer as the state size grows.

#### Out-of-core state

Out-of-core state (once available) can provide a straightforward way to deal with large state sizes. If the change-log between snapshots is small enough to fit in memory and can be checkpointed efficiently this mechanism can deal with growing state sizes. In this case some tuning may be needed to restrict the size of the local cache, which might slow down processing if the key cardinality is continuously large.

#### Lazy state access

If out-of-core state is not an option due to some reason, we can still use partitioned states to scale to large state sizes. We can make use of the snapshot storage layer to provide lazy access to the states instead of keeping everything in memory (as a middle ground between in-memory and out-of-core).

#### Incremental snapshots

As mentioned earlier, besides memory issues large states can significantly slow down the checkpointing process. It is not viable to serialize the full user state and send it over the network with large state sizes. To tackle this issue we can either reduce checkpoint frequency (which is not a real solution) or provide a mechanism to incrementally take state snapshots. We can do incremental snapshotting in two levels: first we can lazily snapshot only the modified keys in case of partitioned states, second we can make individual state snapshots incremental (for instance window buffers are probably good candidates for this).

#### Efficient fault tolerance

Operator states should allow for efficient snapshots to be used with the fault-tolerance mechanisms and also allow fast recovery of the checkpointed states on failure.

#### Lazy and asynchronous snapshots and recovery

We have already seen how partitioned and out-of-core states can be used to reduce the size of the state that needs to be checkpointed. Another technique we can use with OperatorStates is to do the snapshotting asynchronously without stopping the computation. This is possible as we always know when the user reads or updates the state. The only consideration we need to take here is to make sure the user doesn’t accidentally mutate the state while we are snapshotting, so locking the state object might still be necessary.

#### Flexible state backends

The system should provide different alternatives (and should even support custom implementations) for storing the state and the snapshots. Currently we only support in-memory and filesystem based checkpoint storage out-of-the-box, which can be set in the job configuration or from the StreamExecutionEnvironment.

The user can also implement his own StateHandleProvider to use arbitrary persistent storage to store the checkpoints.