DataStream API

State & Failure Recovery

Apache Flink® Training

dataArtisans

Flink v1.1.3 – 8.11.2016

Fault Tolerance and Checkpointing

Fault Tolerance

- What happens if a worker thread goes down?
- Flink supports different guarantee levels for failure recovery:
- Exactly once
  - Each event affects the declared state of a program exactly once.
  - **Note**: This does *not* mean that events are processed exactly once!
- At least once
  - Each event affects the declared state of a program at least once
- Deactivated / None / At most once
  - All state is lost in case of a failure

Source & Sink Requirements

- “Exactly once” & “at least once” guarantees require replayable sources
  - Data must be replayed in case of a failure
- “End-to-End exactly once” guarantees require
  - Transactional sinks, or
  - Idempotent writes

Guarantees of Data Sources

Source	Guarantee
Apache Kafka	Exactly once
AWS Kinesis Streams	Exactly once
RabbitMQ	None (v 0.10) / Exactly once (v 1.0)
Twitter Streaming API	None
Collections	Exactly once
Files	Exactly once
Sockets	None

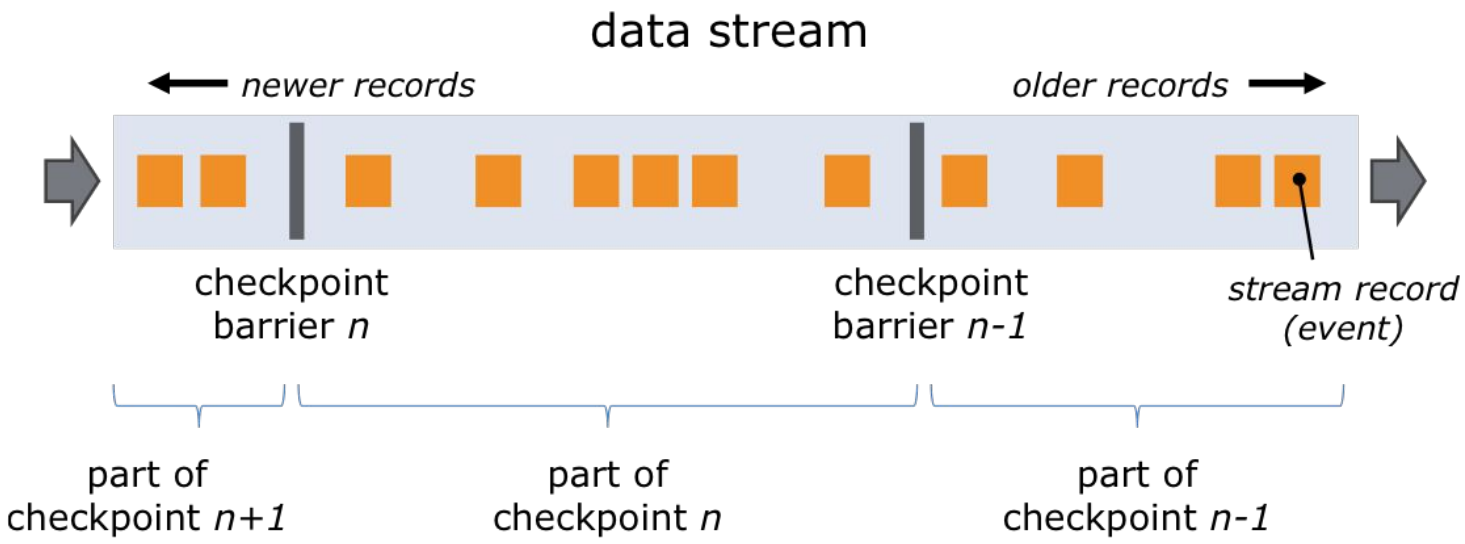
Guarantees of Data Sinks

Sink	Guarantee
HDFS rolling sink	Exactly once
Cassandra	Exactly once (for idempotent updates)
Kafka	At least once
Elasticsearch	At least once
AWS Kinesis Streams	At least once
File sinks	At least once
Socket sinks	At least once
Standard output	At least once
Redis	At least once

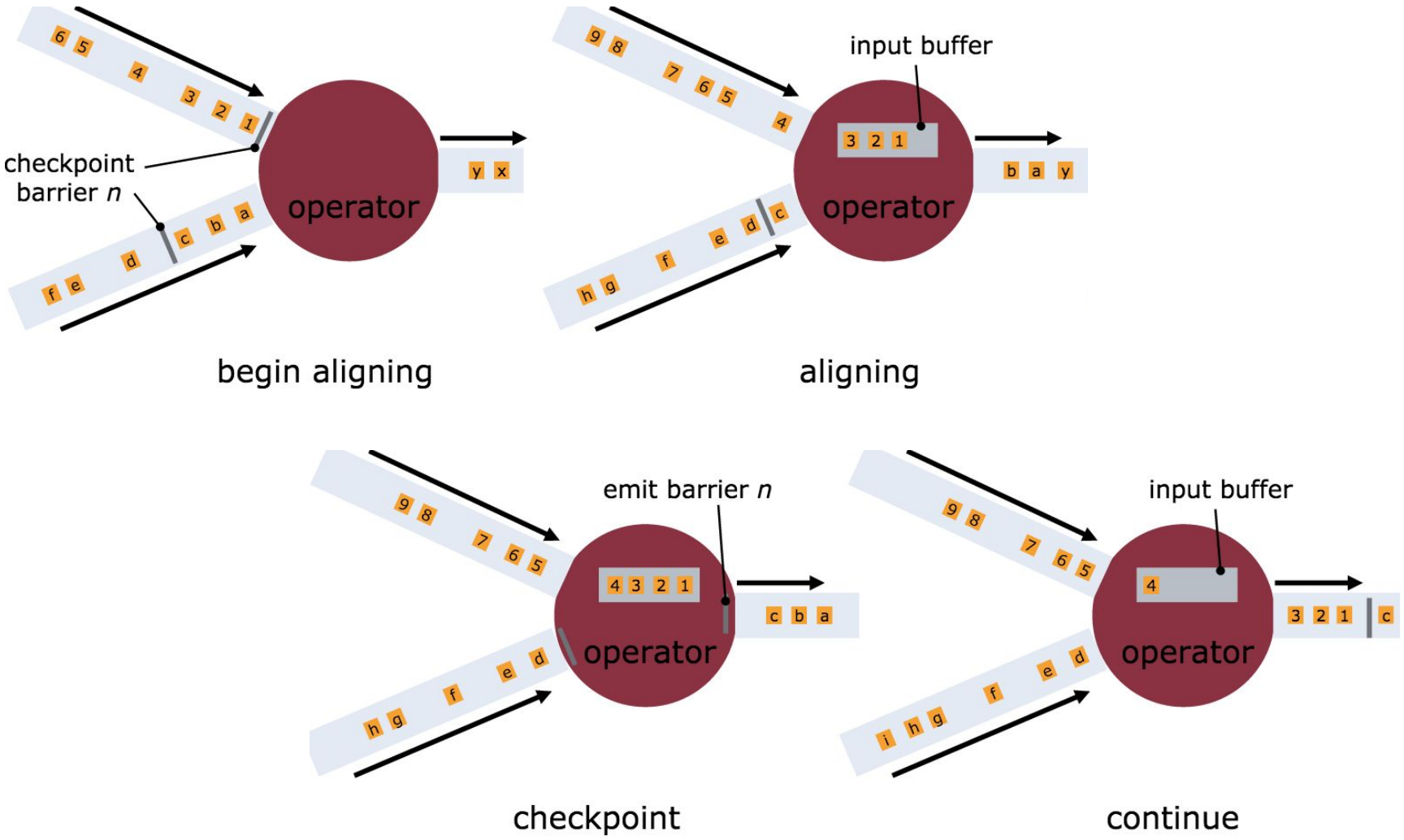
Checkpointing in Flink

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

Checkpoint Barriers



Asynchronous Barrier Snapshotting



Enabling Checkpointing

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

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

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

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

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

Restart Strategies

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

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

- See the docs for details
  https://ci.apache.org/projects/flink/flink-docs-release-1.2/setup/fault_tolerance.html#restart-strategies

Operator State

Stateful Functions

- All DataStream functions can be stateful
  - State is checkpointed and restored in case of a failure (if checkpointing is enabled).
- Flink supports two types of state:
- Local State: Functions can arrange for local variables to be checkpointed.
- Key-Partitioned State: Functions on a keyed stream can access and update state scoped to the current key.
  Note: This mechanism scales better and should be preferred.

Using Local State

```
DataStream<String> aStream;
DataStream<Long> lengths = aStream.map(new MapWithCounter());

public static class MapWithCounter
     implements MapFunction<String, Long>, Checkpointed<Long> {

     private long totalLength = 0;

     @Override
     public Long map (String value) {
          totalLength += value.length();
          return totalLength;
     }

     @Override
     public Long snapshotState(long cpId, long cpTimestamp) throws Exception {
          return totalLength;
     }

     @Override
     public void restoreState(Long state) throws Exception {
          totalLength = state;
     }
}
```

Using Key-Partitioned State

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

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

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

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

State Backends

State in Flink

- There are several sources of state in Flink
  - Windows gather elements and aggregates until they are triggered
  - User-functions use key-partitioned state or implement the Checkpointed interface
  - Sources and Sinks persist state
- Which checkpointing enabled, state is persisted upon checkpoints.
- Internal representation, storage location and method depends on the configured State Backend.

State Backends

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

State Backend Configuration

- Configuration of default state backend in

  ./conf/flink-conf.yaml

- State backend configuration in job

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

- See the docs for details

  https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/state_backends.html

Savepoints

Savepoints

- Savepoints are user-triggered, retained checkpoints.
- A program can be started from a savepoint.
  - Initializes the operator state
- Savepoints are useful for
  - Application updates
  - Updating a Flink version
  - Maintenance & migration
  - A/B testing
  - Rescaling (in the future)

References

- Documentation
  - https://ci.apache.org/projects/flink/flink-docs-release-1.2/internals/stream_checkpointing.html
  - https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/state.html
  - https://ci.apache.org/projects/flink/flink-docs-release-1.2/setup/fault_tolerance.html
  - https://ci.apache.org/projects/flink/flink-docs-release-1.2/setup/savepoints.html
  - https://ci.apache.org/projects/flink/flink-docs-release-1.2/setup/cli.html

- Blog posts
  - http://data-artisans.com/how-apache-flink-enables-new-streaming-applications/