**CHAPTER 8**

**Reading from and Writing to External**

**Systems**

Data can be stored in many different systems, such as filesystems, object stores, rela‐ tional database systems, key-value stores, search indexes, event logs, message queues, and so on. Each class of systems has been designed for specific access patterns and excels at serving a certain purpose. Consequently, today’s data infrastructures often consist of many different storage systems. Before adding a new component into the mix, a logical question to ask should be, “How well does it work with the other com‐ ponents in my stack?”

Adding a data processing system, such as Apache Flink, requires careful considera‐ tions because it does not include its own storage layer but relies on external storage systems to ingest and persist data. Hence, it is important for data processors like Flink to provide a well-equipped library of connectors to read data from and write data to external systems as well as an API to implement custom connectors. However, just being able to read or write data to external datastores is not sufficient for a stream processor that wants to provide meaningful consistency guarantees in the case of fail‐ ure.

In this chapter, we discuss how source and sink connectors affect the consistency guarantees of Flink streaming applications and present Flink’s most popular connec‐ tors to read and write data. You will learn how to implement custom source and sink connectors and how to implement functions that send asynchronous read or write requests to external datastores.

**183**

# Application Consistency Guarantees

In [“Checkpoints, Savepoints, and State Recovery”](#_bookmark20), you learned that Flink’s check‐ pointing and recovery mechanism periodically takes consistent checkpoints of an application’s state. In case of a failure, the state of the application is restored from the latest completed checkpoint and processing continues. However, being able to reset the state of an application to a consistent point is not sufficient to achieve satisfying processing guarantees for an application. Instead, the source and sink connectors of an application need to be integrated with Flink’s checkpointing and recovery mecha‐ nism and provide certain properties to be able to give meaningful guarantees.

In order to provide exactly-once state consistency for an application,[1](#_bookmark29) each source connector of the application needs to be able to set its read positions to a previously checkpointed position. When taking a checkpoint, a source operator persists its read‐ ing positions and restores these positions during recovery. Examples for source con‐ nectors that support the checkpointing of reading positions are file-based sources that store the reading offsets in the byte stream of the file or a Kafka source that stores the reading offsets in the topic partitions it consumes. If an application ingests data from a source connector that is not able to store and reset a reading position, it might suffer from data loss in the case of a failure and only provide at-most-once guarantees.

The combination of Flink’s checkpointing and recovery mechanism and resettable source connectors guarantees that an application will not lose any data. However, the application might emit results twice because all results that have been emitted after the last successful checkpoint (the one to which the application falls back in the case of a recovery) will be emitted again. Therefore, resettable sources and Flink’s recovery mechanism are not sufficient to provide end-to-end exactly-once guarantees even though the application state is exactly-once consistent.

An application that aims to provide end-to-end exactly-once guarantees requires spe‐ cial sink connectors. There are two techniques that sink connectors can apply in dif‐ ferent situations to achieve exactly-once guarantees: *idempotent* writes and *transactional* writes.

## Idempotent Writes

An idempotent operation can be performed several times but will only result in a sin‐ gle change. For example, repeatedly inserting the same key-value pair into a hashmap is an idempotent operation because the first insert operation adds the value for the key into the map and all following insertions will not change the map since it already

1. Exactly-once state consistency is a requirement for end-to-end exactly-once consistency but is not the same.

contains the key-value pair. On the other hand, an append operation is not an idem‐ potent operation, because appending an element multiple times results in multiple appends. Idempotent write operations are interesting for streaming applications because they can be performed multiple times without changing the results. Hence, they can to some extent mitigate the effect of replayed results as caused by Flink’s checkpointing mechanism.

It should be noted an application that relies on idempotent sinks to achieve exactly- once results must guarantee that it overrides previously written results while it replays. For example, an application with a sink that upserts into a key-value store must ensure that it deterministically computes the keys that are used to upsert. More‐ over, applications that read from the sink system might observe unexpected results during the time when an application recovers. When the replay starts, previously emitted results might be overridden by earlier results. Hence, an application that con‐ sumes the output of the recovering application might witness a jump back in time, e.g., read a smaller count than before. Also, the overall result of the streaming appli‐ cation will be in an inconsistent state while the replay is in progress because some results will be overridden while others are not. Once the replay completes and the application is past the point at which it previously failed, the result will be consistent again.

## Transactional Writes

The second approach to achieve end-to-end exactly-once consistency is based on transactional writes. The idea here is to only write those results to an external sink system that have been computed before the last successful checkpoint. This behavior guarantees end-to-end exactly-once because in case of a failure, the application is reset to the last checkpoint and no results have been emitted to the sink system after that checkpoint. By only writing data once a checkpoint is completed, the transac‐ tional approach does not suffer from the replay inconsistency of the idempotent writes. However, it adds latency because results only become visible when a check‐ point completes.

Flink provides two building blocks to implement transactional sink connectors—a generic *write-ahead-log (WAL)* sink and a *two-phase-commit (2PC)* sink. The WAL sink writes all result records into application state and emits them to the sink system once it receives the notification that a checkpoint was completed. Since the sink buf‐ fers records in the state backend, the WAL sink can be used with any kind of sink system. However, it cannot provide bulletproof exactly-once guarantees,[2](#_bookmark29) adds to the state size of an application, and the sink system has to deal with a spiky writing pattern.

1. We discuss the consistency guarantees of a WAL sink in more detail in [“GenericWriteAheadSink”](#_bookmark21).

**Application Consistency Guarantees | 185**

In contrast, the 2PC sink requires a sink system that offers transactional support or exposes building blocks to emulate transactions. For each checkpoint, the sink starts a transaction and appends all received records to the transaction, writing them to the sink system without committing them. When it receives the notification that a check‐ point completed, it commits the transaction and materializes the written results. The mechanism relies on the ability of a sink to commit a transaction after recovering from a failure that was opened before a completed checkpoint.

The 2PC protocol piggybacks on Flink’s existing checkpointing mechanism. The checkpoint barriers are notifications to start a new transaction, the notifications of all operators about the success of their individual checkpoint are their commit votes, and the messages of the JobManager that notify about the success of a checkpoint are the instructions to commit the transactions. In contrast to WAL sinks, 2PC sinks can achieve exactly-once output depending on the sink system and the sink’s implementa‐ tion. Moreover, a 2PC sink continuously writes records to the sink system compared to the spiky writing pattern of a WAL sink.

[Table 8-1](#_bookmark8) shows the end-to-end consistency guarantees for different types of source and sink connectors that can be achieved in the *best case*; depending on the imple‐ mentation of the sink, the actual consistency might be worse.

*Table 8-1. End-to-end consistency guarantees for different combinations of sources and sinks*

**Nonresettable source Resettable source**

Any sink At-most-once At-least-once Idempotent sink At-most-once Exactly-once\*

(temporary inconsistencies

during recovery)

WAL sink At-most-once At-least-once

2PC sink At-most-once Exactly-once

# Provided Connectors

Apache Flink provides connectors to read data from and write data to a variety of storage systems. Message queues and event logs, such as Apache Kafka, Kinesis, or RabbitMQ, are common sources to ingest data streams. In batch processing- dominated environments, data streams are also often ingested by monitoring a file‐ system directory and reading files as they appear.

On the sink side, data streams are often produced into message queues to make the events available to subsequent streaming applications, written to filesystems for archiving or making the data available for offline analytics or batch applications, or inserted into key-value stores or relational database systems, like Cassandra, Elastic‐ Search, or MySQL, to make the data searchable and queryable, or to serve dashboard applications.

Unfortunately, there are no standard interfaces for most of these storage systems, except JDBC for relational DBMS. Instead, every system features its own connector library with a proprietary protocol. As a consequence, processing systems like Flink need to maintain several dedicated connectors to be able to read events from and write events to the most commonly used message queues, event logs, filesystems, key- value stores, and database systems.

Flink provides connectors for Apache Kafka, Kinesis, RabbitMQ, Apache Nifi, vari‐ ous filesystems, Cassandra, ElasticSearch, and JDBC. In addition, the Apache Bahir project provides additional Flink connectors for ActiveMQ, Akka, Flume, Netty, and Redis.

In order to use a provided connector in your application, you need to add its depend‐ ency to the build file of your project. We explained how to add connector dependen‐ cies in [“Including External and Flink Dependencies” on page 107](#_bookmark22).

In the following section, we discuss the connectors for Apache Kafka, file-based sour‐ ces and sinks, and Apache Cassandra. These are the most widely used connectors and they also represent important types of source and sink systems. You can find more information about the other connectors in the documentation for [Apache Flink](http://bit.ly/2UtSrGk) or [Apache Bahir](http://bit.ly/2HOGWmE).

## Apache Kafka Source Connector

Apache Kafka is a distributed streaming platform. Its core is a distributed publish- subscribe messaging system that is widely adopted to ingest and distribute event streams. We briefly explain the main concepts of Kafka before we dive into the details of Flink’s Kafka connector.

Kafka organizes event streams as so-called topics. A topic is an event log that guaran‐ tees that events are read in the same order in which they were written. In order to scale writing to and reading from a topic, it can be split into partitions that are dis‐ tributed across a cluster. The ordering guarantee is limited to a partition—Kafka does not provide ordering guarantees when reading from different partitions. The reading position in a Kafka partition is called an offset.

Flink provides source connectors for all common Kafka versions. Through Kafka 0.11, the API of the client library evolved and new features were added. For instance, Kafka 0.10 added support for record timestamps. Since release 1.0, the API has remained stable. Flink provides a universal Kafka connector that works for all Kafka versions since 0.11. Flink also features version-specific connectors for the Kafka ver‐ sions 0.8, 0.9, 0.10, and 0.11. For the remainder of this section, we focus on the uni‐ versal connector and for the version-specific connectors, we refer you to Flink’s documentation.

The dependency for the universal Flink Kafka connector is added to a Maven project as shown in the following:

**<dependency>**

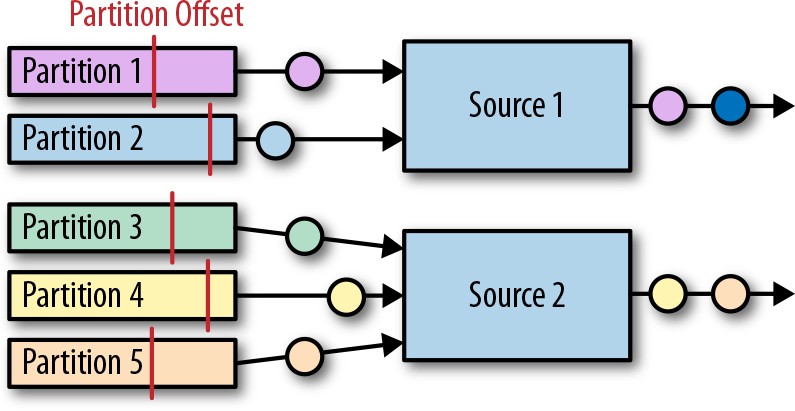
**<groupId>**org.apache.flink**</groupId>**

**<artifactId>**flink-connector-kafka\_2.12**</artifactId>**

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

**</dependency>**

The Flink Kafka connector ingests event streams in parallel. Each parallel source task can read from one or more partitions. A task tracks for each partition its current reading offset and includes it into its checkpoint data. When recovering from a fail‐ ure, the offsets are restored and the source instance continues reading from the checkpointed offset. The Flink Kafka connector does not rely on Kafka’s own offset- tracking mechanism, which is based on so-called consumer groups. [Figure 8-1](#_bookmark9) shows the assignment of partitions to source instances.



*Figure 8-1. Read offsets of Kafka topic partitions*

A Kafka source connector is created as shown in [Example 8-1](#_bookmark23).

*Example 8-1. Creating a Flink Kafka source*

**val** properties **= new Properties**() properties.setProperty("bootstrap.servers", "localhost:9092") properties.setProperty("group.id", "test")

**val** stream**: DataStream**[**String**] **=** env.addSource(

**new FlinkKafkaConsumer**[**String**]( "topic",

**new SimpleStringSchema**(), properties))

The constructor takes three arguments. The first argument defines the topics to read from. This can be a single topic, a list of topics, or a regular expression that matches all topics to read from. When reading from multiple topics, the Kafka connector treats all partitions of all topics the same and multiplexes their events into a single stream.

The second argument is a DeserializationSchema or KeyedDeserialization Schema. Kafka messages are stored as raw byte messages and need to be deserialized into Java or Scala objects. The SimpleStringSchema, which is used in [Example 8-1](#_bookmark23), is a built-in DeserializationSchema that simply deserializes a byte array into a String.

In addition, Flink provides implementations for Apache Avro and text-based JSON encodings. DeserializationSchema and KeyedDeserializationSchema are public interfaces so you can always implement custom deserialization logic.

The third parameter is a Properties object that configures the Kafka client that is internally used to connect to and read from Kafka. A minimum Properties configu‐ ration consists of two entries, "bootstrap.servers" and "group.id". Consult the Kafka documentation for additional configuration properties.

In order to extract event-time timestamps and generate watermarks, you can provide an AssignerWithPeriodicWatermark or an AssignerWithPunctuatedWatermark to a Kafka consumer by calling a FlinkKafkaConsumer.assignTimestampsAndWater mark().[3](#_bookmark29) An assigner is applied to each partition to leverage the per partition order‐ ing guarantees, and the source instance merges the partition watermarks according to

the watermark propagation protocol (see [“Watermark Propagation and Event Time”](#_bookmark10)).

Note that the watermarks of a source instance cannot make pro‐ gress if a partition becomes inactive and does not provide mes‐ sages. As a consequence, a single inactive partition causes the whole application to stall because the application’s watermarks do not make progress.

As of version 0.10.0, Kafka supports message timestamps. When reading from Kafka version 0.10 or later, the consumer will automatically extract the message timestamp as an event-time timestamp if the application runs in event-time mode. In this case,

you still need to generate watermarks and should apply an AssignerWithPeriodicWa termark or an AssignerWithPunctuatedWatermark that forwards the previously assigned Kafka timestamp.

There are a few more notable configuration options. You can configure the starting position from which the partitions of a topic are initially read. Valid options are:

* + The last reading position known by Kafka for the consumer group that was con‐ figured via the group.id parameter. This is the default behavior: FlinkKafkaConsumer.setStartFromGroupOffsets()

1. See [Chapter 6](#_bookmark0) for details about the timestamp assigner interfaces.
   * The earliest offset of each individual partition:

FlinkKafkaConsumer.setStartFromEarliest()

* + The latest offset of each individual partition:

FlinkKafkaConsumer.setStartFromLatest()

* + All records with a timestamp greater than a given timestamp (requires Kafka

0.10.x or later):

FlinkKafkaConsumer.setStartFromTimestamp(long)

* + Specific reading positions for all partitions as provided by a Map object:

FlinkKafkaConsumer.setStartFromSpecificOffsets(Map)

Note that this configuration only affects the first reading positions. In the case of a recovery or when starting from a savepoint, an application will start reading from the offsets stored in the check‐ point or savepoint.

A Flink Kafka consumer can be configured to automatically discover new topics that match the regular expression or new partitions that were added to a topic. These fea‐ tures are disabled by default and can be enabled by adding the parameter

flink.partition-discovery.interval-millis with a nonnegative value to the

Properties object.

## Apache Kafka Sink Connector

Flink provides sink connectors for all Kafka versions since 0.8. Through Kafka 0.11, the API of the client library evolved and new features were added, such as record timestamp support with Kafka 0.10 and transactional writes with Kafka 0.11. Since release 1.0, the API has remained stable. Flink provides a universal Kafka connector that works for all Kafka versions since 0.11. Flink also features version-specific con‐ nectors for Kafka versions 0.8, 0.9, 0.10, and 0.11. For the remainder of this section, we focus on the universal connector and refer you to Flink’s documentation for the version-specific connectors. The dependency for Flink’s universal Kafka connector is added to a Maven project as shown in the following:

**<dependency>**

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

**<artifactId>**flink-connector-kafka\_2.12**</artifactId>**

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

**</dependency>**

A Kafka sink is added to a DataStream application as shown in [Example 8-2](#_bookmark1).

*Example 8-2. Creating a Flink Kafka sink*

**val** stream**: DataStream**[**String**] **=** ...

**val** myProducer **= new FlinkKafkaProducer**[**String**]( "localhost:9092", *// broker list*

"topic", *// target topic*

**new SimpleStringSchema**) *// serialization schema*

stream.addSink(myProducer)

The constructor used in [Example 8-2](#_bookmark1) receives three parameters. The first parameter is a comma-separated string of Kafka broker addresses. The second is the name of the

topic to which the data is written, and the last is a SerializationSchema that con‐ verts the input types of the sink (String in [Example 8-2](#_bookmark1)) into a byte array. SerializationSchema is the counterpart of the DeserializationSchema that we dis‐ cussed in the Kafka source section.

FlinkKafkaProducer provides more constructors with different combinations of arguments as follows:

* + Similar to the Kafka source connector, you can pass a Properties object to pro‐ vide custom options to the internal Kafka client. When using Properties, the list of brokers has to be provided as a "bootstrap.servers" property. Have a look at the Kafka documentation for a comprehensive list of parameters.
  + You can specify a FlinkKafkaPartitioner to control how records are mapped to Kafka partitions. We will discuss this feature in more depth later in this section.
  + Instead of using a SerializationSchema to convert records into byte arrays, you can also specify a KeyedSerializationSchema, which serializes a record into two byte arrays—one for the key and one for the value of a Kafka message. Moreover,

KeyedSerializationSchema also exposes more Kafka-specific functionality, such as overriding the target topic to write to multiple topics.

### At-least-once guarantees for the Kafka sink

The consistency guarantees that Flink’s Kafka sink provides depend on its configura‐ tion. The Kafka sink provides at-least-once guarantees under the following condi‐ tions:

* + Flink’s checkpointing is enabled and all sources of the application are resettable.
  + The sink connector throws an exception if a write does not succeed, causing the application to fail and recover. This is the default behavior. The internal Kafka client can be configured to retry writes before declaring them failed by setting the

retries property to a value larger than zero (the default). You can also configure the sink to log write only failures by calling setLogFailuresOnly(true) on the sink object. Note that this will void any output guarantees of the application.

* + The sink connector waits for Kafka to acknowledge in-flight records before com‐ pleting its checkpoint. This is the default behavior. By calling setFlushOnCheck point(false) on the sink object, you can disable this waiting. However, this will

also disable any output guarantees.

### Exactly-once guarantees for the Kafka sink

Kafka 0.11 introduced support for transactional writes. Due to this feature, Flink’s Kafka sink is also able to provide exactly-once output guarantees given that the sink and Kafka are properly configured. Again, a Flink application must enable check‐

pointing and consume from resettable sources. Moreover, FlinkKafkaProducer pro‐ vides a constructor with a Semantic parameter that controls the consistency guarantees provided by the sink. Possible consistency values are:

* + Semantic.NONE, which provides no guarantees—records might be lost or written multiple times.
  + Semantic.AT\_LEAST\_ONCE, which guarantees that no write is lost but might be duplicated. This is the default setting.
  + Semantic.EXACTLY\_ONCE, which builds on Kafka’s transactions to write each record exactly once.

There are a few things to consider when running a Flink application with a Kafka sink that operates in exactly-once mode, and it helps to roughly understand how Kafka processes transactions. In a nutshell, Kafka’s transactions work by appending all messages to the log of a partition and marking messages of open transactions as uncommitted. Once a transaction is committed, the markers are changed to commit‐ ted. A consumer that reads from a topic can be configured with an isolation level (via

the isolation.level property) to declare whether it can read uncommitted mes‐ sages (read\_uncommitted, the default) or not (read\_committed). If the consumer is configured to read\_committed, it stops consuming from a partition once it encoun‐ ters an uncommitted message and resumes when the message is committed. Hence,

open transactions can block consumers from reading a partition and introduce sig‐ nificant delays. Kafka guards against this by rejecting and closing transactions after a

timeout interval, which is configured with the transaction.timeout.ms property.

In the context of Flink’s Kafka sink, this is important because transactions that time out—due to long recovery cycles, for example—lead to data loss. So, it is crucial to configure the transaction timeout property appropriately. By default, the Flink Kafka

sink sets transaction.timeout.ms to one hour, which means you probably need to adjust the transaction.max.timeout.ms property of your Kafka setup, which is set to 15 minutes by default. Moreover, the visibility of committed messages depends on

the checkpoint interval of a Flink application. Refer to the Flink documentation to learn about a few other corner cases when enabling exactly-once consistency.

#### Check the Configuration of Your Kafka Cluster

The default configuration of a Kafka cluster can still lead to data loss, even after a write is acknowledged. You should carefully revise the configuration of your Kafka setup, paying special attention to the following parameters:

* + - acks
    - log.flush.interval.messages
    - log.flush.interval.ms
    - log.flush.\*

We refer you to the Kafka documentation for details about its con‐ figuration parameters and guidelines for a suitable configuration.

### Custom Partitioning and Writing Message Timestamps

When writing messages to a Kafka topic, a Flink Kafka sink task can choose to which partition of the topic to write. FlinkKafkaPartitioner can be defined in some con‐ structors of the Flink Kafka sink. If not specified, the default partitioner maps each

sink task to a single Kafka partition—all records emitted by the same sink task are written to the same partition and a single partition may contain the records of multi‐ ple sink tasks if there are more tasks than partitions. If the number of partitions is larger than the number of subtasks, the default configuration results in empty parti‐ tions, which can cause problems for Flink applications consuming the topic in event- time mode.

By providing a custom FlinkKafkaPartitioner, you can control how records are routed to topic partitions. For example, you can create a partitioner based on a key

attribute of the records or a round-robin partitioner for even distribution. There is also the option to delegate the partitioning to Kafka based on the message key. This

requires a KeyedSerializationSchema in order to extract the message keys and con‐ figure the FlinkKafkaPartitioner parameter with null to disable the default partitioner.

Finally, Flink’s Kafka sink can be configured to write message timestamps as sup‐ ported since Kafka 0.10. Writing the event-time timestamp of a record to Kafka is

enabled by calling setWriteTimestampToKafka(true) on the sink object.

## Filesystem Source Connector

Filesystems are commonly used to store large amounts of data in a cost-efficient way. In big data architectures they often serve as data source and data sink for batch pro‐ cessing applications. In combination with advanced file formats, such as Apache Par‐ quet or Apache ORC, filesystems can efficiently serve analytical query engines such as Apache Hive, Apache Impala, or Presto. Therefore, filesystems are commonly used to “connect” streaming and batch applications.

Apache Flink features a resettable source connector to ingest data in files as streams. The filesystem source is part of the flink-streaming-java module. Hence, you do not need to add any other dependency to use this feature. Flink supports different

types of filesystems, such as local filesystems (including locally mounted NFS or SAN shares, Hadoop HDFS, Amazon S3, and OpenStack Swift FS). Refer to [“Filesystem](#_bookmark11) [Configuration” on page 237](#_bookmark11) to learn how to configure filesystems in Flink. [Example 8-3](#_bookmark12) shows how to ingest a stream by reading text files line-wise.

*Example 8-3. Creating a filesystem source*

**val** lineReader **= new TextInputFormat**(**null**)

**val** lineStream**: DataStream**[**String**] **=** env.readFile[**String**]( lineReader, *// The FileInputFormat* "hdfs:///path/to/my/data", *// The path to read*

**FileProcessingMode**

.**PROCESS\_CONTINUOUSLY**, *// The processing mode*

30000L) *// The monitoring interval in ms*

The arguments of the StreamExecutionEnvironment.readFile() method are:

* + A FileInputFormat, which is responsible for reading the content of the files. We discuss the details of this interface later in this section. The null parameter of TextInputFormat in [Example 8-3](#_bookmark12) defines the path that is separately set.
  + The path that should be read. If the path refers to a file, the single file is read. If it refers to a directory, FileInputFormat scans the directory for files to read.
  + The mode in which the path should be read. The mode can either be PRO CESS\_ONCE or PROCESS\_CONTINUOUSLY. In PROCESS\_ONCE mode, the read path is scanned once when the job is started and all matching files are read. In PRO CESS\_CONTINUOUSLY, the path is periodically scanned (after an initial scan) and

new and modified files are continuously read.

* + The interval in milliseconds in which the path is periodically scanned. The parameter is ignored in PROCESS\_ONCE mode.

FileInputFormat is a specialized InputFormat to read files from a filesystem.[4](#_bookmark30) A FileInputFormat reads files in two steps. First it scans a filesystem path and creates so-called input splits for all matching files. An input split defines a range on a file,

typically via a start offset and a length. After dividing a large file into multiple splits, the splits can be distributed to multiple reader tasks to read the file in parallel. Depending on the encoding of a file, it can be necessary to only generate a single split

to read the file as a whole. The second step of a FileInputFormat is to receive an

input split, read the file range that is defined by the split, and return all corresponding

records.

A FileInputFormat used in a DataStream application should also implement the CheckpointableInputFormat interface, which defines methods to checkpoint and reset the the current reading position of an InputFormat within a file split. The file‐ system source connector provides only at-least-once guarantees when checkpointing is enabled if the FileInputFormat does not implement the CheckpointableInputFor mat interface because the input format will start reading from the beginning of the split that was processed when the last complete checkpoint was taken.

In version 1.7, Flink provides a few classes that extend FileInputFormat and imple‐ ment CheckpointableInputFormat. TextInputFormat reads text files line-wise (split by newline characters), subclasses of CsvInputFormat read files with comma- separated values, and AvroInputFormat reads files with Avro-encoded records.

In PROCESS\_CONTINUOUSLY mode, the filesystem source connector identifies new files based on their modification timestamp. This means a file is completely reprocessed if

it is modified because its modification timestamp changes. This includes modifica‐ tions due to appending writes. Therefore, a common technique to continuously ingest files is to write them in a temporary directory and atomically move them to the monitored directory once they are finalized. When a file is completely ingested and a checkpoint completed, it can be removed from the directory. Monitoring ingested files by tracking the modification timestamp also has implications if you read from file stores with eventually consistent list operations, such as S3. Since files might not appear in order of their modification timestamps, they may be ignored by the filesys‐ tem source connector.

Note that in PROCESS\_ONCE mode, no checkpoints are taken after the filesystem path is scanned and all splits are created.

If you want to use a filesystem source connector in an event-time application, you should be aware that it can be challenging to generate watermarks since input splits are generated in a single process and round-robin distributed to all parallel readers

1. InputFormat is Flink’s interface to define data sources in the DataSet API.

that process them in order of the modification timestamp of the file. In order to gen‐ erate satisfying watermarks you need to reason about the smallest timestamp of a record that is included in a split later processed by the task.

## Filesystem Sink Connector

Writing a stream into files is a common requirement, for example, to prepare data with low latency for offline ad-hoc analysis. Since most applications can only read files once they are finalized and streaming applications run for long periods of time, streaming sink connectors typically chunk their output into multiple files. Moreover, it is common to organize records into so-called buckets, so that consuming applica‐ tions have more control over which data to read.

Like the filesystem source connector, Flink’s StreamingFileSink connector is con‐ tained in the flink-streaming-java module. Hence, you do not need to add a dependency to your build file to use it.

StreamingFileSink provides end-to-end exactly-once guarantees for an application given that the application is configured with exactly-once checkpoints and all its

sources are reset in the case of a failure. We will discuss the recovery mechanism in more detail later in this section. [Example 8-4](#_bookmark13) shows how to create a StreamingFile Sink with minimal configuration and append it to a stream.

*Example 8-4. Creating a StreamingFileSink in row-encoding mode*

**val** input**: DataStream**[**String**] **=** …

**val** sink**: StreamingFileSink**[**String**] **= StreamingFileSink**

.forRowFormat(

**new Path**("/base/path"),

**new SimpleStringEncoder**[**String**]("UTF-8"))

.build()

input.addSink(sink)

When a StreamingFileSink receives a record, the record is assigned to a bucket. A bucket is a subdirectory of the base path that is configured with the builder of Stream ingFileSink—"/base/path" in [Example 8-4](#_bookmark13).

The bucket is chosen by a BucketAssigner, which is a public interface and returns for every record a BucketId that determines the directory to which the record will be written. The BucketAssigner can be configured on the builder using the withBucketAssigner() method. If no BucketAssigner is explicitly specified, it uses a DateTimeBucketAssigner that assigns records to hourly buckets based on the pro‐

cessing time when they are written.

Each bucket directory contains multiple part files that are concurrently written by multiple parallel instances of the StreamingFileSink. Moreover, each parallel instance chunks its output into multiple part files. The path of a part file has the fol‐

lowing format:

[base-path]/[bucket-path]/part-[task-idx]-[id]

For example, given a base path of "/johndoe/demo" and a part prefix of "part", the path "/johndoe/demo/2018-07-22--17/part-4-8" points to the eight file that was written by the fifth (0-indexed) sink task to bucket "2018-07-22--17"—the 5 p.m. bucket of July 22, 2018.

#### IDs of Committed Files May Not Be Consecutive

Nonconsecutive file IDs, the last number in a committed file’s name, do not indicate data loss. StreamingFileSink simply incre‐ ments the file IDs. When discarding pending files it does not reuse

their IDs.

A RollingPolicy determines when a task creates a new part file. You can configure the RollingPolicy with the withRollingPolicy() method on the builder. By default, StreamingFileSink uses a DefaultRollingPolicy, which is configured to roll part files when they exceed 128 MB or are older than 60 seconds. You can also

configure an inactivity interval after which a part file is rolled.

StreamingFileSink supports two modes of writing records to part files: row encod‐ ing and bulk encoding. In row encoding mode, every record is individually encoded

and appended to a part file. In bulk encoding, records are collected and written in batches. Apache Parquet, which organizes and compresses records in a columnar for‐ mat, is a file format that requires bulk encoding.

[Example 8-4](#_bookmark13) creates a StreamingFileSink with row encoding by providing an Encoder that writes single records to a part file. In [Example 8-4](#_bookmark13), we use a Sim pleStringEncoder, which calls the toString() method of the record and writes the String representation of the record to the file. Encoder is a simple interface with a

single method that can be easily implemented.

A bulk-encoding StreamingFileSink is created as shown in [Example 8-5](#_bookmark27).

*Example 8-5. Creating a StreamingFileSink in bulk-encoding mode*

**val** input**: DataStream**[**String**] **=** …

**val** sink**: StreamingFileSink**[**String**] **= StreamingFileSink**

.forBulkFormat(

**new Path**("/base/path"), **ParquetAvroWriters**.forSpecificRecord(classOf[**AvroPojo**]))

.build()

input.addSink(sink)

A StreamingFileSink in bulk-encoding mode requires a BulkWriter.Factory. In [Example 8-5](#_bookmark27) we use a Parquet writer for Avro files. Note that the Parquet writer is contained in the flink-parquet module, which needs to be added as a dependency.

As usual, BulkWriter.Factory is an interface that can be implemented for custom

file formats, such as Apache Orc.

A StreamingFileSink in bulk-encoding mode cannot choose a Roll ingPolicy. Bulk-encoding formats can only be combined with the OnCheckpointRollingPolicy, which rolls in-progress part files on every checkpoint.

StreamingFileSink provides exactly-once output guarantees. The sink achieves this by a commit protocol that moves files through different stages, in progress, pending,

and finished, and that is based on Flink’s checkpointing mechanism. While a sink writes to a file, the file is in the in-progress state. When the RollingPolicy decides to roll a file, it is closed and moved into the pending state by renaming it. Pending files

are moved into the finished state (again by renaming) when the next checkpoint completes.

#### Pending Files Might Never Be Committed

In some situations, a pending file is never committed. The Stream ingFileSink ensures this does not result in data loss. However, these files are not automatically cleaned up.

Before manually deleting a pending file, you need to check whether it is lingering or about to be committed. Once you find a commit‐ ted file with the same task index and a higher ID, you can safely remove a pending file.

In the case of a failure, a sink task needs to reset its current in-progress file to its writ‐ ing offset at the last successful checkpoint. This is done by closing the current in- progress file and discarding the invalid part at the file’s end, for example, by using the filesystem’s truncate operation.

#### StreamingFileSink Requires Checkpointing Be Enabled

StreamingFileSink will never move files from pending into fin‐ ished state, if an application does not enable checkpointing.

## Apache Cassandra Sink Connector

Apache Cassandra is a popular, scalable, and highly available column store database system. Cassandra models datasets as tables of rows that consist of multiple typed col‐ umns. One or more columns have to be defined as (composite) primary keys. Each row can be uniquely identified by its primary key. Among other APIs, Cassandra fea‐ tures the Cassandra Query Language (CQL), a SQL-like language to read and write records and create, modify, and delete database objects, such as keyspaces and tables.

Flink provides a sink connector to write data streams to Cassandra. Cassandra’s data model is based on primary keys, and all writes to Cassandra happen with upsert semantics. In combination with exactly-once checkpointing, resettable sources, and deterministic application logic, upsert writes yield eventually exactly-once output consistency. The output is only eventually consistent, because results are reset to a previous version during recovery, meaning consumers might read older results than read previously. Also, the versions of values for multiple keys might be out of sync.

In order to prevent temporal inconsistencies during recovery and provide exactly- once output guarantees for applications with nondeterministic application logic, Flink’s Cassandra connector can be configured to leverage a WAL. We will discuss the WAL mode in more detail later in this section. The following code shows the depend‐ ency you need to add to the build file of your application in order to use the Cassan‐ dra sink connector:

**<dependency>**

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

**<artifactId>**flink-connector-cassandra\_2.12**</artifactId>**

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

**</dependency>**

To illustrate the use of the Cassandra sink connector, we use the simple example of a Cassandra table that holds data about sensor readings and consists of two columns,

sensorId and temperature. The CQL statements in [Example 8-6](#_bookmark24) create a keyspace

“example” and a table “sensors” in that keyspace.

*Example 8-6. Defining a Cassandra example table*

CREATE KEYSPACE IF NOT EXISTS example

WITH replication = {'class': 'SimpleStrategy', 'replication\_factor': '1'};

CREATE TABLE IF NOT EXISTS example.sensors ( sensorId VARCHAR,

temperature FLOAT, PRIMARY KEY(sensorId)

);

Flink provides different sink implementations to write data streams of different data types to Cassandra. Flink’s Java tuples and Row type and Scala’s built-in tuples and case classes are handled differently than user-defined POJO types. We discuss both cases separately. [Example 8-7](#_bookmark7) shows how to create a sink that writes a DataStream of tuples,

case classes, or rows into a Cassandra table. In this example, a DataStream[(String,

Float)] is written into the “sensors” table.

*Example 8-7. Creating a Cassandra sink for tuples*

**val** readings**: DataStream**[(**String**, **Float**)] **=** ???

**val** sinkBuilder**: CassandraSinkBuilder**[(**String**, **Float**)] **= CassandraSink**.addSink(readings)

sinkBuilder

.setHost("localhost")

.setQuery(

"INSERT INTO example.sensors(sensorId, temperature) VALUES (?, ?);")

.build()

Cassandra sinks are created and configured using a builder that is obtained by calling the CassandraSink.addSink() method with the DataStream object that should be emitted. The method returns the right builder for the data type of the DataStream. In

[Example 8-7](#_bookmark7), it returns a builder for a Cassandra sink that handles Scala tuples.

The Cassandra sink builders for tuples, case classes, and rows require the specifica‐ tion of a CQL INSERT query.[5](#_bookmark31) The query is configured using the CassandraSink Builder.setQuery() method. During execution, the sink registers the query as a

prepared statement and converts the fields of tuples, case classes, or rows into param‐

eters for the prepared statement. The fields are mapped to the parameters based on their position; the first value is converted to the first parameter and so on.

Since POJO fields do not have a natural order, they need to be treated differently. [Example 8-8](#_bookmark28) shows how to configure a Cassandra sink for a POJO of type

SensorReading.

*Example 8-8. Create a Cassandra sink for POJOs*

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

**CassandraSink**.addSink(readings)

.setHost("localhost")

.build()

1. In contrast to SQL INSERT statements, CQL INSERT statements behave like upsert queries—they override existing rows with the same primary key.

As you can see in [Example 8-8](#_bookmark28), we do not specify an INSERT query. Instead, POJOs are handed to Cassandra’s Object Mapper, which automatically maps POJO fields to fields of a Cassandra table. In order for this to work, the POJO class and its fields need to be annotated with Cassandra annotations and provide setters and getters for all fields as shown in [Example 8-9](#_bookmark14). The default constructor is required by Flink as mentioned in [“Supported Data Types” on page 98](#_bookmark15) when discussing supported data types.

*Example 8-9. POJO class with Cassandra Object Mapper annotations*

@Table(keyspace **=** "example", name **=** "sensors")

**class SensorReadings**(

@Column(name **=** "sensorId") **var** id**: String**, @Column(name **=** "temperature") **var** temp**: Float**) {

**def this**() **=** {

**this**("", 0.0)

}

**def** setId(id**: String**)**: Unit** = **this**.id **=** id

**def** getId**: String** = id

**def** setTemp(temp**: Float**)**: Unit** = **this**.temp **=** temp

**def** getTemp**: Float** = temp

}

In addition to the configuration options in Figures [8-7](#_bookmark7) and [8-8](#_bookmark28), a Cassandra sink builder provides a few more methods to configure the sink connector:

* + setClusterBuilder(ClusterBuilder): The ClusterBuilder builds a Cassandra Cluster that manages the connection to Cassandra. Among other options, it can configure the hostnames and ports of one or more contact points; define load

balancing, retry, and reconnection policies; and provide access credentials.

* + setHost(String, [Int]): This method is a shortcut for a simple Cluster Builder configured with the hostname and port of a single contact point. If no port is configured, Cassandra’s default port 9042 is used.
  + setQuery(String): This specifies the CQL INSERT query to write tuples, case classes, or rows to Cassandra. A query must not be configured to emit POJOs.
  + setMapperOptions(MapperOptions): This provides options for Cassandra’s Object Mapper, such as configurations for consistency, time-to-live (TTL), and

null field handling. The options are ignored if the sink emits tuples, case classes, or rows.

* + enableWriteAheadLog([CheckpointCommitter]): This enables the WAL to pro‐ vide exactly-once output guarantees in the case of nondeterministic application logic. CheckpointCommitter is used to store information about completed check‐

points in an external datastore. If no CheckpointCommitter is configured, the information is written into a specific Cassandra table.

The Cassandra sink connector with WAL is implemented based on Flink’s GenericWriteAheadSink operator. How this operator works, including the role of the CheckpointCommitter, and which consistency guarantees it provides, is described in

more detail in [“Transactional Sink Connectors” on page 209](#_bookmark2).

# Implementing a Custom Source Function

The DataStream API provides two interfaces to implement source connectors along with corresponding RichFunction abstract classes:

* + SourceFunction and RichSourceFunction can be used to define nonparallel source connectors—sources that run with a single task.
  + ParallelSourceFunction and RichParallelSourceFunction can be used to define source connectors that run with multiple parallel task instances.

With the exception of being nonparallel and parallel, both interfaces are identical. Just like the rich variants of processing functions,[6](#_bookmark29) subclasses of RichSourceFunction and RichParallelSourceFunction can override the open() and close() methods and access a RuntimeContext that provides the number of parallel task instances and the index of the current instance, among other things.

SourceFunction and ParallelSourceFunction define two methods:

* + void run(SourceContext<T> ctx)
  + void cancel()

The run() method is doing the actual work of reading or receiving records and ingesting them into a Flink application. Depending on the system from which the data is received, the data might be pushed or pulled. The run() method is called once by Flink and runs in a dedicated source thread, typically reading or receiving data

and emitting records in an endless loop (infinite stream). The task can be explicitly canceled at some point in time or terminated in the case of a finite stream when the input is fully consumed.

The cancel() method is invoked by Flink when the application is canceled and shut down. In order to perform a graceful shutdown, the run() method, which runs in a separate thread, should terminate as soon as the cancel() method is called. [Example 8-10](#_bookmark1) shows a simple source function that counts from 0 to Long.MaxValue.

1. Rich functions were discussed in [Chapter 5](#_bookmark0).

*Example 8-10. SourceFunction that counts to Long.MaxValue*

**class CountSource extends SourceFunction**[**Long**] {

**var** isRunning**: Boolean** = **true**

**override def** run(ctx**: SourceFunction.SourceContext**[**Long**]) **=** {

**var** cnt**: Long** = -1

**while** (isRunning && cnt < **Long**.**MaxValue**) { cnt += 1

ctx.collect(cnt)

}

}

**override def** cancel() **=** isRunning **= false**

}

## Resettable Source Functions

Earlier in this chapter we explained that Flink can only provide satisfying consistency guarantees for applications that use source connectors that can replay their output data. A source function can replay its output if the external system that provides the data exposes an API to retrieve and reset a reading offset. Examples of such systems are filesystems that provide the offset of a file stream and a seek method to move a file stream to a specific position or Apache Kafka, which provides offsets for each parti‐ tion of a topic and can set the reading position of a partition. A counterexample is a source connector that reads data from a network socket, which immediately discards delivered the data.

A source function that supports output replay needs to be integrated with Flink’s checkpointing mechanism and must persist all current reading positions when a checkpoint is taken. When the application is started from a savepoint or recovers from a failure, the reading offsets are retrieved from the latest checkpoint or save‐ point. If the application is started without existing state, the reading offsets must be

set to a default value. A resettable source function needs to implement the Checkpoin tedFunction interface and should store the reading offsets and all related meta infor‐ mation, such as file paths or partition ID, in operator list state or operator union list

state depending on how the offsets should be distributed to parallel task instances in the case of a rescaled application. See [“Scaling Stateful Operators” on page 56](#_bookmark16) for details on the distribution behavior of operator list state and union list state.

In addition, it is important to ensure that the SourceFunction.run() method, which runs in a separate thread, does not advance the reading offset and emit data while a checkpoint is taken; in other words, while the CheckpointedFunction.snapshot

State() method is called. This is done by guarding the code in run() that advances

the reading position and emits records in a block that synchronizes on a lock object,

which is obtained from the SourceContext.getCheckpointLock() method. [Example 8-11](#_bookmark5) makes the CountSource of [Example 8-10](#_bookmark1) resettable.

*Example 8-11. A resettable SourceFunction*

**class ResettableCountSource**

**extends SourceFunction**[**Long**] **with CheckpointedFunction** {

**var** isRunning**: Boolean** = **true var** cnt**: Long** = **\_**

**var** offsetState**: ListState**[**Long**] **= \_**

**override def** run(ctx**: SourceFunction.SourceContext**[**Long**]) **=** {

**while** (isRunning && cnt < **Long**.**MaxValue**) {

*// synchronize data emission and checkpoints*

ctx.getCheckpointLock.synchronized { cnt += 1

ctx.collect(cnt)

}

}

}

**override def** cancel() **=** isRunning **= false**

**override def** snapshotState(snapshotCtx**: FunctionSnapshotContext**)**: Unit** = {

*// remove previous cnt*

offsetState.clear()

*// add current cnt*

offsetState.add(cnt)

}

**override def** initializeState(

initCtx**: FunctionInitializationContext**)**: Unit** = {

**val** desc **= new ListStateDescriptor**[**Long**]("offset", classOf[**Long**]) offsetState **=** initCtx.getOperatorStateStore.getListState(desc)

*// initialize cnt variable*

**val** it **=** offsetState.get()

cnt **= if** (**null** == it || !it.iterator().hasNext) {

-1L

} **else** { it.iterator().next()

}

}

}

## Source Functions, Timestamps, and Watermarks

Another important aspect of source functions are timestamps and watermarks. As pointed out in [“Event-Time Processing”](#_bookmark17) and [“Assigning Timestamps and Generating](#_bookmark3)

[Watermarks”](#_bookmark3), the DataStream API provides two options to assign timestamps and generate watermarks. Timestamps and watermarks can be assigned and generate by a

dedicated TimestampAssigner (see [“Assigning Timestamps and Generating Water‐](#_bookmark3)

[marks”](#_bookmark3) for details) or be assigned and generated by a source function.

A source function assigns timestamps and emits watermarks through its SourceCon text object. SourceContext provides the following methods:

* + def collectWithTimestamp(T record, long timestamp): Unit
  + def emitWatermark(Watermark watermark): Unit

collectWithTimestamp() emits a record with its associated timestamp and

emitWatermark() emits the provided watermark.

Besides removing the need for an additional operator, assigning timestamps and gen‐ erating watermarks in a source function can be beneficial if one parallel instance of a source function consumes records from multiple stream partitions, such as partitions of a Kafka topic. Typically, external systems, such as Kafka, only guarantee message order within a stream partition. Given the case of a source function operator that runs with a parallelism of two and that reads data from a Kafka topic with six parti‐ tions, each parallel instance of the source function will read records from three Kafka topic partitions. Consequently, each instance of the source function multiplexes the records of three stream partitions to emit them. Multiplexing records most likely introduces additional out-of-orderness with respect to the event-time timestamps such that a downstream timestamp assigner might produce more late records than expected.

To avoid such behavior, a source function can generate watermarks for each stream partition independently and always emit the smallest watermark of its partitions as its watermark. This way, it can ensure that the order guarantees on each partition are leveraged and no unnecessary late records are emitted.

Another problem source functions have to deal with are instances that become idle and do not emit anymore data. This can be very problematic, because it may prevent the whole application from advancing its watermarks and hence lead to a stalling application. Since watermarks should be data driven, a watermark generator (either integrated in a source function or in a timestamp assigner) will not emit new water‐ marks if it does not receive input records. If you look at how Flink propagates and updates watermarks (see [“Watermark Propagation and Event Time”](#_bookmark10)), you can see that a single operator that does not advance watermarks can grind all watermarks of an

application to a halt if the application involves a shuffle operation (keyBy(),

rebalance(), etc.).

Flink provides a mechanism to avoid such situations by marking source functions as temporarily idle. While being idle, Flink’s watermark propagation mechanism will

ignore the idle stream partition. The source is automatically set as active as soon as it starts to emit records again. A source function can decide when to mark itself as idle

and does so by calling the method SourceContext.markAsTemporarilyIdle().

# Implementing a Custom Sink Function

In Flink’s DataStream API, any operator or function can send data to an external sys‐ tem or application. A DataStream does not have to eventually flow into a sink opera‐ tor. For instance, you could implement a FlatMapFunction that emits each incoming

record via an HTTP POST call and not via its Collector. Nonetheless, the Data‐ Stream API provides a dedicated SinkFunction interface and a corresponding Rich SinkFunction abstract class.[7](#_bookmark32) The SinkFunction interface provides a single method:

void invoke(IN value, Context ctx)

The Context object of SinkFunction provides access to the current processing time, the current watermark (i.e., the current event time at the sink), and the timestamp of

the record.

[Example 8-12](#_bookmark25) shows a simple SinkFunction that writes sensor readings to a socket. Note that you need to start a process that listens on the socket before starting the pro‐ gram. Otherwise, the program fails with a ConnectException because a connection to

the socket could not be opened. Run the command nc -l localhost 9191 on Linux

to listen on localhost:9191.

*Example 8-12. A simple SinkFunction that writes to a socket*

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

*// write the sensor readings to a socket*

readings.addSink(**new SimpleSocketSink**("localhost", 9191))

*// set parallelism to 1 because only one thread can write to a socket*

.setParallelism(1)

*// -----*

**class SimpleSocketSink**(**val** host**: String**, **val** port**: Int**) **extends RichSinkFunction**[**SensorReading**] {

**var** socket**: Socket** = **\_**

**var** writer**: PrintStream** = **\_**

1. Usually the RichSinkFunction interface is used because sink functions typically need to set up a connection to an external system in the RichFunction.open() method. See [Chapter 5](#_bookmark0) for details on the RichFunction interface.

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

*// open socket and writer*

socket **= new Socket**(**InetAddress**.getByName(host), port) writer **= new PrintStream**(socket.getOutputStream)

}

**override def** invoke( value**: SensorReading**,

ctx**: SinkFunction.Context**[**\_**])**: Unit** = {

*// write sensor reading to socket* writer.println(value.toString) writer.flush()

}

**override def** close()**: Unit** = {

*// close writer and socket* writer.close() socket.close()

}

}

As discussed, the end-to-end consistency guarantees of an application depend on the properties of its sink connectors. In order to achieve end-to-end exactly-once seman‐ tics, an application requires either idempotent or transactional sink connectors. The

SinkFunction in [Example 8-12](#_bookmark25) neither performs idempotent writes nor features

transactional writes. Due to the append-only characteristic of a socket, it is not possi‐

ble to perform idempotent writes. Since a socket does not have built-in transactional support, transactional writes can only be done using Flink’s generic WAL sink. In the following sections, you will learn how to implement idempotent or transactional sink connectors.

## Idempotent Sink Connectors

For many applications, the SinkFunction interface is sufficient to implement an idempotent sink connector. This is possible if the following two properties hold:

1. The result data has a deterministic (composite) key, on which idempotent updates can be performed. For an application that computes the average temper‐ ature per sensor and minute, a deterministic key could be the ID of the sensor and the timestamp for each minute. Deterministic keys are important to ensure all writes are correctly overwritten in case of a recovery.
2. The external system supports updates per key, such as a relational database sys‐ tem or a key-value store.

[Example 8-13](#_bookmark1) illustrates how to implement and use an idempotent SinkFunction that writes to a JDBC database, in this case an embedded Apache Derby database.

*Example 8-13. An idempotent SinkFunction that writes to a JDBC database*

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

*// write the sensor readings to a Derby table*

readings.addSink(**new DerbyUpsertSink**)

*// -----*

**class DerbyUpsertSink extends RichSinkFunction**[**SensorReading**] {

**var** conn**: Connection** = **\_**

**var** insertStmt**: PreparedStatement** = **\_ var** updateStmt**: PreparedStatement** = **\_**

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

*// connect to embedded in-memory Derby*

conn **= DriverManager**.getConnection( "jdbc:derby:memory:flinkExample", **new Properties**())

*// prepare insert and update statements*

insertStmt **=** conn.prepareStatement(

"INSERT INTO Temperatures (sensor, temp) VALUES (?, ?)") updateStmt **=** conn.prepareStatement(

"UPDATE Temperatures SET temp = ? WHERE sensor = ?")

}

**override def** invoke(r**: SensorReading**, context**: Context**[**\_**])**: Unit** = {

*// set parameters for update statement and execute it* updateStmt.setDouble(1, r.temperature) updateStmt.setString(2, r.id)

updateStmt.execute()

*// execute insert statement if update statement did not update any row*

**if** (updateStmt.getUpdateCount == 0) {

*// set parameters for insert statement* insertStmt.setString(1, r.id) insertStmt.setDouble(2, r.temperature)

*// execute insert statement*

insertStmt.execute()

}

}

**override def** close()**: Unit** = { insertStmt.close() updateStmt.close() conn.close()

}

}

Since Apache Derby does not provide a built-in UPSERT statement, the example sink performs UPSERT writes by first trying to update a row and inserting a new row if no

row with the given key exists. The Cassandra sink connector follows the same approach when the WAL is not enabled.

## Transactional Sink Connectors

Whenever an idempotent sink connector is not suitable, either the characteristics of the application’s output, the properties of the required sink system, or due to stricter consistency requirements, transactional sink connectors can be an alternative. As described before, transactional sink connectors need to be integrated with Flink’s checkpointing mechanism because they may only commit data to the external system when a checkpoint completes successfully.

In order to ease the implementation of transactional sinks, Flink’s DataStream API provides two templates that can be extended to implement custom sink operators.

Both templates implement the CheckpointListener interface to receive notifications

from the JobManager about completed checkpoints (see [“Receiving Notifications](#_bookmark1)

[About Completed Checkpoints”](#_bookmark1) for details about the interface):

* + The GenericWriteAheadSink template collects all outgoing records per check‐ point and stores them in the operator state of the sink task. The state is check‐

pointed and recovered in the case of a failure. When a task receives a checkpoint completion notification, it writes the records of the completed checkpoints to the external system. The Cassandra sink connector with WAL-enabled implements this interface.

* + The TwoPhaseCommitSinkFunction template leverages transactional features of the external sink system. For every checkpoint, it starts a new transaction and

writes all following records to the sink system in the context of the current trans‐ action. The sink commits a transaction when it receives the completion notifica‐ tion of the corresponding checkpoint.

In the following, we describe both interfaces and their consistency guarantees.

### GenericWriteAheadSink

GenericWriteAheadSink eases the implementation of sink operators with improved consistency properties. The operator is integrated with Flink’s checkpointing mecha‐

nism and aims to write each record exactly once to an external system. However, you should be aware that failure scenarios exist in which a write-ahead log sink emits

records more than once. Hence, a GenericWriteAheadSink does not provide bullet‐

proof exactly-once guarantees but only at-least-once guarantees. We will discuss these

scenarios in more detail later in this section.

GenericWriteAheadSink works by appending all received records to a write-ahead log that is segmented by checkpoints. Every time the sink operator receives a check‐

point barrier, it starts a new section and all the following records are appended to the new section. The WAL is stored and checkpointed as operator state. Since the log will be recovered, no records will be lost in the case of a failure.

When GenericWriteAheadSink receives a notification about a completed checkpoint, it emits all records that are stored in the WAL in the segment corresponding to the

successful checkpoint. Depending on the concrete implementation of the sink opera‐ tor, the records can be written to any kind of storage or message system. When all records have been successfully emitted, the corresponding checkpoint must be inter‐ nally committed.

A checkpoint is committed in two steps. First, the sink persistently stores the infor‐ mation that the checkpoint was committed and secondly it removes the records from the WAL. It is not possible to store the commit information in Flink’s application state because it is not persistent and would be reset in case of a failure. Instead,

GenericWriteAheadSink relies on a pluggable component called CheckpointCommit ter to store and look up information about committed checkpoints in an external persistent storage. For example, the Cassandra sink connector by default uses a Check pointCommitter that writes to Cassandra.

Thanks to the built-in logic of GenericWriteAheadSink, it is not difficult to imple‐ ment a sink that leverages a WAL. Operators that extend GenericWriteAheadSink need to provide three constructor parameters:

* + A CheckpointCommitter as discussed before
  + A TypeSerializer to serialize the input records
  + A job ID that is passed to the CheckpointCommitter to identify commit informa‐ tion across application restarts

Moreover, the write-ahead operator needs to implement a single method:

boolean sendValues(**Iterable**<**IN**> values, long chkpntId, long timestamp) GenericWriteAheadSink calls the sendValues() method to write the records of a completed checkpoint to the external storage system. The method receives an

Iterable over all records of a checkpoint, the ID of the checkpoint, and the time‐ stamp of when the checkpoint was taken. The method must return true if all writes succeeded and false if a write failed.

[Example 8-14](#_bookmark6) shows the implementation of a write-ahead sink that writes to the stan‐ dard output. It uses FileCheckpointCommitter, which we do not discuss here. You can look up its implementation in the repository that contains the examples of the

book.

Note that GenericWriteAheadSink does not implement the Sink Function interface. So, sinks that extend GenericWriteAheadSink cannot be added using DataStream.addSink() but are attached using the DataStream.transform() method.

*Example 8-14. A WAL sink that writes to the standard output*

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

*// write the sensor readings to the standard out via a write-ahead log*

readings.transform(

"WriteAheadSink", **new SocketWriteAheadSink**)

*// -----*

**class StdOutWriteAheadSink extends GenericWriteAheadSink**[**SensorReading**](

*// CheckpointCommitter that commits checkpoints to the local filesystem*

**new FileCheckpointCommitter**(**System**.getProperty("java.io.tmpdir")),

*// Serializer for records*

createTypeInformation[**SensorReading**]

.createSerializer(**new ExecutionConfig**),

*// Random JobID used by the CheckpointCommitter*

**UUID**.randomUUID.toString) {

**override def** sendValues(

readings**: Iterable**[**SensorReading**], checkpointId**: Long**,

timestamp**: Long**)**: Boolean** = {

**for** (r **<-** readings.asScala) {

*// write record to standard out*

println(r)

}

**true**

}

}

The examples repository contains an application that fails and recovers in regular intervals to demonstrate the behavior of StdOutWriteAheadSink and a regular Data Stream.print() sink in case of failures.

As mentioned earlier, GenericWriteAheadSink cannot provide bulletproof exactly- once guarantees. There are two failure cases that can result in records being emitted

more than once:

1. The program fails while a task is running the sendValues() method. If the exter‐ nal sink system cannot atomically write multiple records—either all or none—

some records might have been written and others not. Since the checkpoint was not committed yet, the sink will write all records again during recovery.

1. All records are correctly written and the sendValues() method returns true; however, the program fails before CheckpointCommitter is called or Checkpoint Committer fails to commit the checkpoint. During recovery, all records of not- yet-committed checkpoints will be written again.

Note that these failure scenarios do not affect the exactly-once guarantees of the Cassandra sink connector because it performs UPSERT writes. The Cassandra sink connector benefits from the WAL because it guards from nondeterministic keys and prevents inconsistent writes to Cassandra.

### TwoPhaseCommitSinkFunction

Flink provides the TwoPhaseCommitSinkFunction interface to ease the implementa‐ tion of sink functions that provide end-to-end exactly-once guarantees. However,

whether a 2PC sink function provides such guarantees or not depends on the imple‐ mentation details. We start the discussion of this interface with a question: “Isn’t the 2PC protocol too expensive?”

In general, 2PCs are an expensive approach to ensuring consistency in a distributed system. However, in the context of Flink, the protocol is only run once for every

checkpoint. Moreover, the protocol of TwoPhaseCommitSinkFunction piggybacks on Flink’s regular checkpointing mechanism and thus adds little overhead. The TwoPhase CommitSinkFunction works quite similar to the WAL sink, but it does not collect records in Flink’s application state; rather, it writes them in an open transaction to an

external sink system.

The TwoPhaseCommitSinkFunction implements the following protocol. Before a sink task emits its first record, it starts a transaction on the external sink system. All subse‐

quently received records are written in the context of the transaction. The voting phase of the 2PC protocol starts when the JobManager initiates a checkpoint and injects barriers in the sources of the application. When an operator receives the bar‐ rier, it checkpoints it state and sends an acknowledgment message to the JobManager once it is done. When a sink task receives the barrier, it persists its state, prepares the current transaction for committing, and acknowledges the checkpoint at the JobMan‐ ager. The acknowledgment messages to the JobManager are analogous to the commit vote of the textbook 2PC protocol. The sink task must not yet commit the transac‐ tion, because it is not guaranteed that all tasks of the job will complete their check‐ points. The sink task also starts a new transaction for all records that arrive before the next checkpoint barrier.

When the JobManager receives successful checkpoint notifications from all task instances, it sends the checkpoint completion notification to all interested tasks. This

notification corresponds to the 2PC protocol’s commit command. When a sink task receives the notification, it commits all open transactions of previous checkpoints.[8](#_bookmark29) Once a sink task acknowledges its checkpoint, it must be able to commit the corre‐ sponding transaction, even in the case of a failure. If the transaction cannot be com‐ mitted, the sink loses data. An iteration of the 2PC protocol succeeds when all sink tasks committed their transactions.

Let’s summarize the requirements for the external sink system:

* + The external sink system must provide transactional support or the sink must be able to emulate transactions on the external system. Hence, the sink should be able to write to the sink system, but the written data must not be made visible to the outside before it is committed.
  + A transaction must be open and accept writes for the duration of a checkpoint interval.
  + A transaction must wait to be committed until a checkpoint completion notifica‐ tion is received. In the case of a recovery cycle, this may take some time. If the sink system closes a transaction (e.g., with a timeout), the not committed data will be lost.
  + The sink must be able to recover a transaction after a process failed. Some sink systems provide a transaction ID that can be used to commit or abort an open transaction.
  + Committing a transaction must be an idempotent operation—the sink or exter‐ nal system should be able to notice that a transaction was already committed or a repeated commit must have no effect.

The protocol and the requirements of the sink system might be easier to understand by looking at a concrete example. [Example 8-15](#_bookmark26) shows a TwoPhaseCommitSinkFunc tion that writes with exactly-once guarantees to a filesystem. Essentially, this is a sim‐ plified version of the BucketingFileSink discussed earlier.

*Example 8-15. A transactional sink that writes to files*

**class TransactionalFileSink**(**val** targetPath**: String**, **val** tempPath**: String**) **extends TwoPhaseCommitSinkFunction**[(**String**, **Double**), **String**, **Void**]( createTypeInformation[**String**].createSerializer(**new ExecutionConfig**), createTypeInformation[**Void**].createSerializer(**new ExecutionConfig**)) {

**var** transactionWriter**: BufferedWriter** = **\_**

1. A task might need to commit multiple transactions if an acknowledgment message is lost.

*/\*\* Creates a temporary file for a transaction into which the records are*

*\* written.*

*\*/*

**override def** beginTransaction()**: String** = {

*// path of transaction file is built from current time and task index*

**val** timeNow **= LocalDateTime**.now(**ZoneId**.of("UTC"))

.format(**DateTimeFormatter**.**ISO\_LOCAL\_DATE\_TIME**)

**val** taskIdx **= this**.getRuntimeContext.getIndexOfThisSubtask **val** transactionFile **=** s"$timeNow-$taskIdx"

*// create transaction file and writer*

**val** tFilePath **= Paths**.get(s"$tempPath/$transactionFile") **Files**.createFile(tFilePath)

**this**.transactionWriter **= Files**.newBufferedWriter(tFilePath) println(s"Creating Transaction File: $tFilePath")

*// name of transaction file is returned to later identify the transaction*

transactionFile

}

*/\*\* Write record into the current transaction file. \*/*

**override def** invoke( transaction**: String**, value**:** (**String**, **Double**),

context**: Context**[**\_**])**: Unit** = { transactionWriter.write(value.toString) transactionWriter.write('\n')

}

*/\*\* Flush and close the current transaction file. \*/* **override def** preCommit(transaction**: String**)**: Unit** = { transactionWriter.flush()

transactionWriter.close()

}

*/\*\* Commit a transaction by moving the precommitted transaction file*

*\* to the target directory.*

*\*/*

**override def** commit(transaction**: String**)**: Unit** = {

**val** tFilePath **= Paths**.get(s"$tempPath/$transaction")

*// check if the file exists to ensure that the commit is idempotent*

**if** (**Files**.exists(tFilePath)) {

**val** cFilePath **= Paths**.get(s"$targetPath/$transaction") **Files**.move(tFilePath, cFilePath)

}

}

*/\*\* Aborts a transaction by deleting the transaction file. \*/*

**override def** abort(transaction**: String**)**: Unit** = {

**val** tFilePath **= Paths**.get(s"$tempPath/$transaction") **if** (**Files**.exists(tFilePath)) { **Files**.delete(tFilePath)

}

}

}

TwoPhaseCommitSinkFunction[IN, TXN, CONTEXT] has three type parameters:

* + IN specifies the type of the input records. In [Example 8-15](#_bookmark26), this is a Tuple2 with a

String and a Double field.

* + TXN defines a transaction identifier that can be used to identify and recover a transaction after a failure. In [Example 8-15](#_bookmark26), this is a string holding the name of

the transaction file.

* + CONTEXT defines an optional custom context. TransactionalFileSink

in [Example 8-15](#_bookmark26) does not need the context and hence sets the type to Void.

The constructor of TwoPhaseCommitSinkFunction requires two TypeSerializer— one for the TXN type and the other for the CONTEXT type.

Finally, TwoPhaseCommitSinkFunction defines five functions that need to be imple‐ mented:

* + beginTransaction(): TXN starts a new transaction and returns the transaction identifier. TransactionalFileSink in [Example 8-15](#_bookmark26) creates a new transaction file and returns its name as the identifier.
  + invoke(txn: TXN, value: IN, context: Context[\_]): Unit writes a value to the current transaction. The sink in [Example 8-15](#_bookmark26) appends the value as a String to the transaction file.
  + preCommit(txn: TXN): Unit precommits a transaction. A precommitted trans‐ action may not receive further writes. Our implementation in [Example 8-15](#_bookmark26)

flushes and closes the transaction file.

* + commit(txn: TXN): Unit commits a transaction. This operation must be idem‐ potent—records must not be written twice to the output system if this method is

called twice. In [Example 8-15](#_bookmark26), we check if the transaction file still exists and move it to the target directory if that is the case.

* + abort(txn: TXN): Unit aborts a transaction. This method may also be called twice for a transaction. Our TransactionalFileSink in [Example 8-15](#_bookmark26) checks if the transaction file still exists and deletes it if that is the case.

As you can see, the implementation of the interface is not too involved. However, the complexity and consistency guarantees of an implementation depend on, among other things, the features and capabilities of the sink system. For instance, Flink’s

Kafka producer implements the TwoPhaseCommitSinkFunction interface. As men‐

tioned before, the connector might lose data if a transaction is rolled back due to a

timeout.[9](#_bookmark29) Hence, it does not offer definitive exactly-once guarantees even though it implements the TwoPhaseCommitSinkFunction interface.

# Asynchronously Accessing External Systems

Besides ingesting or emitting data streams, enriching a data stream by looking up information in a remote database is another common use case that requires interact‐ ing with an external storage system. An example is the well-known Yahoo! stream processing benchmark, which is based on a stream of advertisement clicks that need to be enriched with details about their corresponding campaign that are stored in a key-value store.

The straightforward approach for such use cases is to implement a MapFunction that queries the datastore for every processed record, waits for the query to return a result,

enriches the record, and emits the result. While this approach is easy to implement, it suffers from a major issue: each request to the external datastore adds significant

latency (a request/response involves two network messages) and the MapFunction

spends most of its time waiting for query results.

Apache Flink provides the AsyncFunction to mitigate the latency of remote I/O calls. AsyncFunction concurrently sends multiple queries and processes their results asyn‐ chronously. It can be configured to preserve the order of records (requests might

return in a different order than the order in which they were sent out) or return the results in the order of the query results to further reduce the latency. The function is also properly integrated with Flink’s checkpointing mechanism—input records that are currently waiting for a response are checkpointed and queries are repeated in the

case of a recovery. Moreover, AsyncFunction properly works with event-time pro‐

cessing because it ensures watermarks are not overtaken by records even if out-of-

order results are enabled.

In order to take advantage of AsyncFunction, the external system should provide a client that supports asynchronous calls, which is the case for many systems. If a sys‐

tem only provides a synchronous client, you can spawn threads to send requests and handle them. The interface of AsyncFunction is shown in the following:

**trait AsyncFunction**[**IN**, **OUT**] **extends Function** {

**def** asyncInvoke(input**: IN**, resultFuture**: ResultFuture**[**OUT**])**: Unit**

}

The type parameters of the function define its input and output types. The asyncInvoke() method is called for each input record with two parameters. The first parameter is the input record and the second parameter is a callback object to return

1. See details in [“Apache Kafka Sink Connector”](#_bookmark18).

the result of the function or an exception. In [Example 8-16](#_bookmark4), we show how to apply

AsyncFunction on a DataStream.

*Example 8-16. Applying AsyncFunction on a DataStream*

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

**val** sensorLocations**: DataStream**[(**String**, **String**)] **= AsyncDataStream**

.orderedWait( readings,

**new DerbyAsyncFunction**,

5, **TimeUnit**.**SECONDS**, *// timeout requests after 5 seconds*

100) *// at most 100 concurrent requests*

The asynchronous operator that applies an AsyncFunction is configured with the AsyncDataStream object,[10](#_bookmark30) which provides two static methods: orderedWait() and unorderedWait(). Both methods are overloaded for different combinations of parameters. orderedWait() applies an asynchronous operator that emits results in the order of the input records, while the operator of unorderWait() only ensures

watermarks and checkpoint barriers remain aligned. Additional parameters specify

when to time out the asynchronous call for a record and how many concurrent requests to start. [Example 8-17](#_bookmark19) shows DerbyAsyncFunction, which queries an embedded Derby database via its JDBC interface.

*Example 8-17. AsyncFunction that queries a JDBC database*

**class DerbyAsyncFunction**

**extends AsyncFunction**[**SensorReading**, (**String**, **String**)] {

*// caching execution context used to handle the query threads* **private lazy val** cachingPoolExecCtx **= ExecutionContext**.fromExecutor(**Executors**.newCachedThreadPool())

*// direct execution context to forward result future to callback object*

**private lazy val** directExecCtx **= ExecutionContext**.fromExecutor(

org.apache.flink.runtime.concurrent.**Executors**.directExecutor())

*/\*\**

* *Executes JDBC query in a thread and handles the resulting Future*
* *with an asynchronous callback.*

*\*/*

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

resultFuture**: ResultFuture**[(**String**, **String**)])**: Unit** = {

1. The Java API provides an AsyncDataStream class with the respective static methods.

**Asynchronously Accessing External Systems | 217**

**val** sensor **=** reading.id

*// get room from Derby table as Future*

**val** room**: Future**[**String**] **= Future** {

*// Creating a new connection and statement for each record.*

*// Note: This is NOT best practice!*

*// Connections and prepared statements should be cached.*

**val** conn **= DriverManager**

.getConnection( "jdbc:derby:memory:flinkExample", **new Properties**())

**val** query **=** conn.createStatement()

*// submit query and wait for result; this is a synchronous call*

**val** result **=** query.executeQuery(

s"SELECT room FROM SensorLocations WHERE sensor = '$sensor'")

*// get room if there is one*

**val** room **= if** (result.next()) { result.getString(1)

} **else** { "UNKNOWN ROOM"

}

*// close resultset, statement, and connection*

result.close() query.close() conn.close()

*// return room*

room

}(cachingPoolExecCtx)

*// apply result handling callback on the room future*

room.onComplete {

**case Success**(r) **=>** resultFuture.complete(**Seq**((sensor, r)))

**case Failure**(e) **=>** resultFuture.completeExceptionally(e)

}(directExecCtx)

}

}

The asyncInvoke() method of DerbyAsyncFunction in [Example 8-17](#_bookmark19) wraps the blocking JDBC query in a Future, which is executed via CachedThreadPool. To keep the example concise, we create a new JDBC connection for each record, which is, of

course, quite inefficient and should be avoided. Future[String] holds the result of the JDBC query.

Finally, we apply an onComplete() callback on Future and pass the result (or a possi‐ ble exception) to the ResultFuture handler. In contrast to the JDBC query Future, the onComplete() callback is processed by DirectExecutor because passing the result

to ResultFuture is a lightweight operation that does not require a dedicated thread. Note that all operations are done in a nonblocking fashion.

It is important to point out that an AsyncFunction instance is sequentially called for each of its input records—a function instance is not called in a multithreaded fashion. Therefore, the asyncInvoke() method should quickly return by starting an asynchro‐ nous request and handling the result with a callback that forwards the result to ResultFuture. Common antipatterns that must be avoided include:

* + Sending a request that blocks the asyncInvoke() method
  + Sending an asynchronous request but waiting inside the asyncInvoke() method for the request to complete

# Summary

In this chapter you learned how Flink DataStream applications can read data from and write data to external systems and the requirements for an application to achieve different end-to-end consistency guarantees. We presented Flink’s most commonly used built-in source and sink connectors, which also serve as representatives for dif‐ ferent types of storage systems, such as message queues, filesystems, and key-value stores.

Subsequently, we showed you how to implement custom source and sink connectors, including WAL and 2PC sink connectors, providing detailed examples. Finally, you

learned about Flink’s AsyncFunction, which can significantly improve the perfor‐

mance of interacting with external systems by performing and handling requests

asynchronously.

**Summary | 219**