FREE eBook

LEARNING apache-flink

Free unaffiliated eBook created from
Stack Overflow contributors.

#apache-flink

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: apache-flink

It is an unofficial and free apache-flink ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official apache-flink.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with apache-flink

Remarks

This section provides an overview of what apache-flink is, and why a developer might want to use it.

It should also mention any large subjects within apache-flink, and link out to the related topics. Since the Documentation for apache-flink is new, you may need to create initial versions of those related topics.

Examples

Overview and requirements

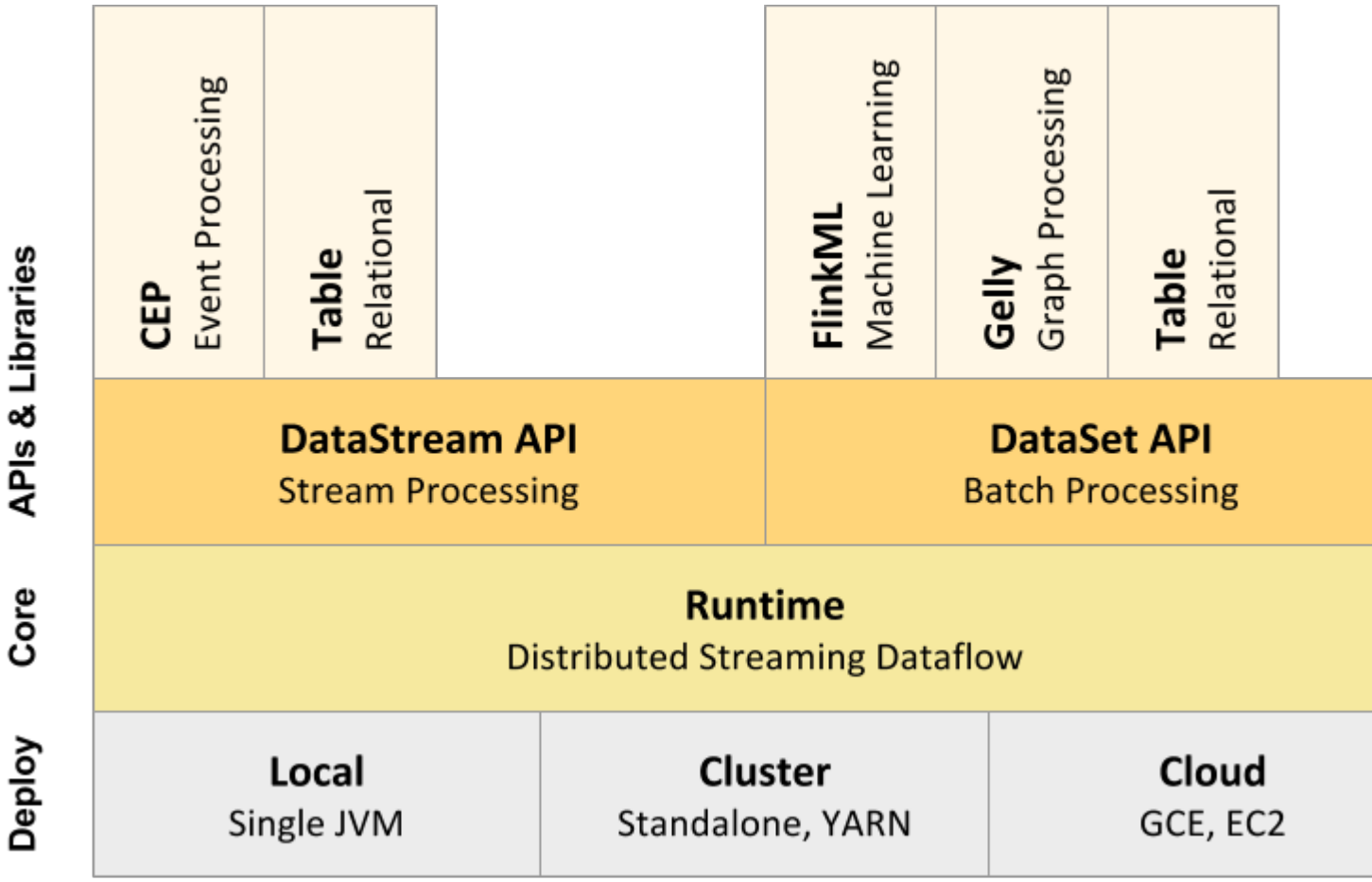
What is Flink

Like Apache Hadoop and Apache Spark, Apache Flink is a community-driven open source framework for distributed Big Data Analytics. Written in Java, Flink has APIs for Scala, Java and Python, allowing for Batch and Real-Time streaming analytics.

Requirements

- a UNIX-like environment, such as Linux, Mac OS X or Cygwin;
- Java 6.X or later;
- [optional] Maven 3.0.4 or later.

Stack



Execution environments

> Apache Flink is a data processing system and **an alternative to Hadoop's MapReduce component**. It comes with its *own runtime* rather than building on top of MapReduce. As such, it can work completely independently of the Hadoop ecosystem.

The `ExecutionEnvironment` is the context in which a program is executed. There are different environments you can use, depending on your needs.

1. *JVM environment*: Flink can run on a single Java Virtual Machine, allowing users to test and debug Flink programs directly from their IDE. When using this environment, all you need is the correct maven dependencies.

2. *Local environment*: to be able to run a program on a running Flink instance (not from within your IDE), you need to install Flink on your machine. See local setup.

3. *Cluster environment*: running Flink in a fully distributed fashion requires a standalone or a yarn cluster. See the cluster setup page or this slideshare for more information. mportant__: the `2.11` in the artifact name is the *scala version*, be sure to match the one you have on your system.

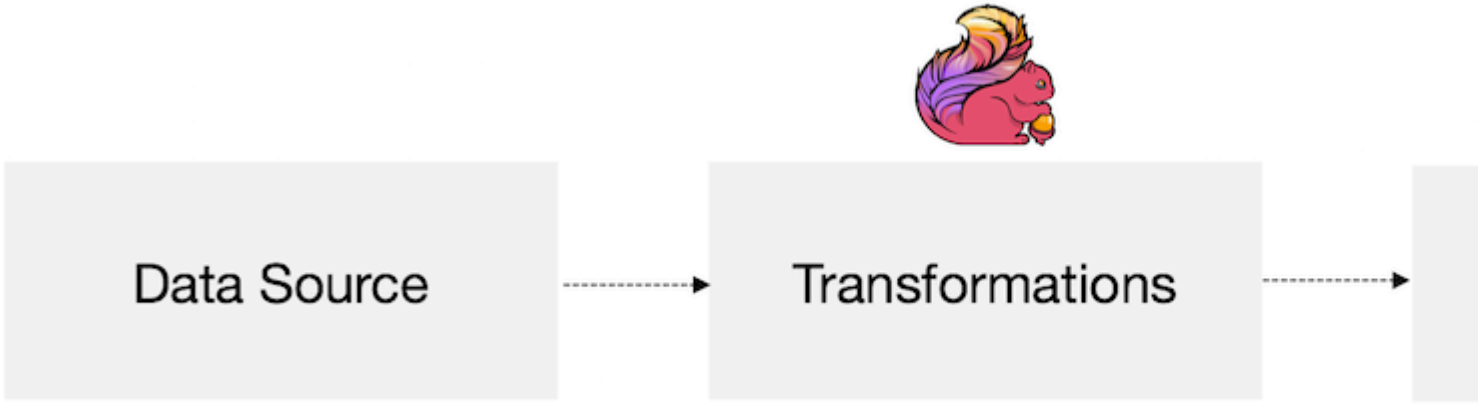
APIs

Flink can be used for either stream or batch processing. They offer three APIs:

- **DataStream API**: stream processing, i.e. transformations (filters, time-windows, aggregations) on unbounded flows of data.
- **DataSet API**: batch processing, i.e. transformations on data sets.
- **Table API**: a SQL-like expression language (like dataframes in Spark) that can be embedded in both batch and streaming applications.

Building blocks

At the most basic level, Flink is made of source(s), transformations(s) and sink(s).



At the most basic level, a Flink program is made up of:

- **Data source**: Incoming data that Flink processes
- **Transformations**: The processing step, when Flink modifies incoming data
- **Data sink**: Where Flink sends data after processing

Sources and sinks can be local/HDFS files, databases, message queues, etc. There are many third-party connectors already available, or you can easily create your own.

Local runtime setup

0. ensure you have java 6 or above and that the `JAVA_HOME` environment variable is set.

1. download the latest flink binary here:

   ```
   wget flink-XXXX.tar.gz
   ```

   If you don't plan to work with Hadoop, pick the hadoop 1 version. Also, note the scala version you download, so you can add the correct maven dependencies in your programs.

2. start flink:

   ```
   tar xzvf flink-XXXX.tar.gz
   ```

```
./flink/bin/start-local.sh
```

Flink is already configured to run locally. To ensure flink is running, you can inspect the logs in `flink/log/` or open the flink jobManager's interface running on `http://localhost:8081`.

3. stop flink:

```
./flink/bin/stop-local.sh
```

Flink Environment setup

To run a flink program from your IDE(we can use either Eclipse or Intellij IDEA(preffered)), you need two dependencies:`flink-java` / `flink-scala` and `flink-clients` (as of february 2016). These JARS can be added using Maven and SBT(if you are using scala).

- **Maven**

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-java</artifactId>
    <version>1.1.4</version>
</dependency>

<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-clients_2.11</artifactId>
    <version>1.1.4</version>
</dependency>
```

- **SBT** name := " "

```
version := "1.0"

scalaVersion := "2.11.8"

libraryDependencies ++= Seq(
  "org.apache.flink" %% "flink-scala" % "1.2.0",
  "org.apache.flink" %% "flink-clients" % "1.2.0"
)
```

important: the `2.11` in the artifact name is the *scala version*, be sure to match the one you have on your system.

WordCount - Table API

This example is the same as *WordCount*, but uses the Table API. See *WordCount* for details about execution and results.

Maven

To use the Table API, add `flink-table` as a maven dependency:

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-table_2.11</artifactId>
    <version>1.1.4</version>
</dependency>
```

The code

```
public class WordCountTable{

    public static void main( String[] args ) throws Exception{

        // set up the execution environment
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
        final BatchTableEnvironment tableEnv = TableEnvironment.getTableEnvironment( env );

        // get input data
        DataSource<String> source = env.fromElements(
                "To be, or not to be,--that is the question:--",
                "Whether 'tis nobler in the mind to suffer",
                "The slings and arrows of outrageous fortune",
                "Or to take arms against a sea of troubles"
        );

        // split the sentences into words
        FlatMapOperator<String, String> dataset = source
                .flatMap( ( String value, Collector<String> out ) -> {
                    for( String token : value.toLowerCase().split( "\\W+" ) ){
                        if( token.length() > 0 ){
                            out.collect( token );
                        }
                    }
                } )
                // with lambdas, we need to tell flink what type to expect
                .returns( String.class );

        // create a table named "words" from the dataset
        tableEnv.registerDataSet( "words", dataset, "word" );

        // word count using an sql query
        Table results = tableEnv.sql( "select word, count(*) from words group by word" );
        tableEnv.toDataSet( results, Row.class ).print();
    }
}
```

Note: For a version using Java < 8, replace the lambda by an anonymous class:

```
FlatMapOperator<String, String> dataset = source.flatMap( new FlatMapFunction<String,
String>(){
        @Override
        public void flatMap( String value, Collector<String> out ) throws Exception{
            for( String token : value.toLowerCase().split( "\\W+" ) ){
                if( token.length() > 0 ){
                    out.collect( token );
                }
```

```
            }
        }
    } );
```

WordCount

Maven

Add the dependencies `flink-java` and `flink-client` (as explained in the *JVM environment setup* example).

The code

```
public class WordCount{

    public static void main( String[] args ) throws Exception{

        // set up the execution environment
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        // input data
        // you can also use env.readTextFile(...) to get words
        DataSet<String> text = env.fromElements(
                "To be, or not to be,--that is the question:--",
                "Whether 'tis nobler in the mind to suffer",
                "The slings and arrows of outrageous fortune",
                "Or to take arms against a sea of troubles,"
        );

        DataSet<Tuple2<String, Integer>> counts =
                // split up the lines in pairs (2-tuples) containing: (word,1)
                text.flatMap( new LineSplitter() )
                        // group by the tuple field "0" and sum up tuple field "1"
                        .groupBy( 0 )
                        .aggregate( Aggregations.SUM, 1 );

        // emit result
        counts.print();
    }
}
```

`LineSplitter.java`:

```
public class LineSplitter implements FlatMapFunction<String, Tuple2<String, Integer>>{

    public void flatMap( String value, Collector<Tuple2<String, Integer>> out ){
        // normalize and split the line into words
        String[] tokens = value.toLowerCase().split( "\\W+" );

        // emit the pairs
        for( String token : tokens ){
            if( token.length() > 0 ){
                out.collect( new Tuple2<String, Integer>( token, 1 ) );
            }
        }
```

```
    }
}
```

If you use Java 8, you can replace `.flatmap(new LineSplitter())` by a lambda expression:

```
DataSet<Tuple2<String, Integer>> counts = text
    // split up the lines in pairs (2-tuples) containing: (word,1)
    .flatMap( ( String value, Collector<Tuple2<String, Integer>> out ) -> {
        // normalize and split the line into words
        String[] tokens = value.toLowerCase().split( "\\W+" );

        // emit the pairs
        for( String token : tokens ){
            if( token.length() > 0 ){
                out.collect( new Tuple2<>( token, 1 ) );
            }
        }
    } )
    // group by the tuple field "0" and sum up tuple field "1"
    .groupBy( 0 )
    .aggregate( Aggregations.SUM, 1 );
```

Execution

From the IDE: simply hit *run* in your IDE. Flink will create an environment inside the JVM.

From the flink command line: to run the program using a standalone local environment, do the following:

1. ensure flink is running (`flink/bin/start-local.sh`);
2. create a jar file (`maven package`);
3. use the `flink` command-line tool (in the `bin` folder of your flink installation) to launch the program:

   ```
   flink run -c your.package.WordCount target/your-jar.jar
   ```

   The `-c` option allows you to specify the class to run. It is not necessary if the jar is executable/defines a main class.

Result

```
(a,1)
(against,1)
(and,1)
(arms,1)
(arrows,1)
(be,2)
(fortune,1)
(in,1)
(is,1)
```

```
(mind,1)
(nobler,1)
(not,1)
(of,2)
(or,2)
(outrageous,1)
(question,1)
(sea,1)
(slings,1)
(suffer,1)
(take,1)
(that,1)
(the,3)
(tis,1)
(to,4)
(troubles,1)
(whether,1)
```

WordCount - Streaming API

This example is the same as *WordCount*, but uses the Table API. See *WordCount* for details about execution and results.

Maven

To use the Streaming API, add `flink-streaming` as a maven dependency:

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-streaming-java_2.11</artifactId>
    <version>1.1.4</version>
</dependency>
```

The code

```
public class WordCountStreaming{

    public static void main( String[] args ) throws Exception{

        // set up the execution environment
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // get input data
        DataStreamSource<String> source = env.fromElements(
                "To be, or not to be,--that is the question:--",
                "Whether 'tis nobler in the mind to suffer",
                "The slings and arrows of outrageous fortune",
                "Or to take arms against a sea of troubles"
        );

        source
                // split up the lines in pairs (2-tuples) containing: (word,1)
                .flatMap( ( String value, Collector<Tuple2<String, Integer>> out ) -> {
                    // emit the pairs
```

```
                    for( String token : value.toLowerCase().split( "\\W+" ) ){
                        if( token.length() > 0 ){
                            out.collect( new Tuple2<>( token, 1 ) );
                        }
                    }
                } )
                // due to type erasure, we need to specify the return type
                .returns( TupleTypeInfo.getBasicTupleTypeInfo( String.class, Integer.class ) )
                // group by the tuple field "0"
                .keyBy( 0 )
                // sum up tuple on field "1"
                .sum( 1 )
                // print the result
                .print();

        // start the job
        env.execute();
    }
}
```

Read Getting started with apache-flink online: https://riptutorial.com/apache-flink/topic/5798/getting-started-with-apache-flink

Chapter 2: Checkpointing

Introduction

(tested on Flink 1.2 and below)

Every function, source or operator in Flink can be stateful. Checkpoints allow Flink to recover state and positions in the streams to give the application the same semantics as a failure-free execution. It is the mecanism behind the guarantees of *fault tolerance* and *exactly-once* processing.

Read this article to understand the internals.

Remarks

Checkpoints are only useful when a failure happens in the cluster, for example when a taskmanager fails. They do not persist after the job itself failed or was canceled.

To be able to resume a stateful job after failure/cancellation, have a look at **savepoints** or **externalized checkpoints (flink 1.2+)**.

Examples

Configuration and setup

Checkpointing configuration is done in two steps. First, you need to choose a *backend*. Then, you can specify the interval and mode of the checkpoints in a per-application basis.

Backends

Available backends

Where the checkpoints are stored depends on the configured backend:

- `MemoryStateBackend`: in-memory state, backup to JobManager's/ZooKeeper's memory. Should be used only for minimal state (default to max. 5 MB, for storing Kafka offsets for example) or testing and local debugging.
- `FsStateBackend`: the state is kept in-memory on the TaskManagers, and state snapshots (i.e. checkpoints) are stored in a file system (HDFS, DS3, local filesystem, ...). This setup is encouraged for large states or long windows and for high availability setups.
- `RocksDBStateBackend`: holds in-flight data in a RocksDB database that is (per default) stored in the TaskManager data directories. Upon checkpointing, the whole RocksDB database is written to a file (like above). Compared to the FsStateBackend, it allows for larger states

(limited only by the disk space vs the size of the taskmanager memory), but the throughput will be lower (data not always in memory, must be loaded from disc).

Note that whatever the backend, metadata (number of checkpoints, localisation, etc.) are always stored in the jobmanager memory and checkpoints **won't persist after the application termination/cancellation**.

Specifying the backend

You specify the backend in your program's `main` method using:

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

Or set the default backend in `flink/conf/flink-conf.yaml`:

```
# Supported backends:
#  - jobmanager (MemoryStateBackend),
#  - filesystem (FsStateBackend),
#  - rocksdb (RocksDBStateBackend),
#  - <class-name-of-factory>
state.backend: filesystem

# Directory for storing checkpoints in a Flink-supported filesystem
# Note: State backend must be accessible from the JobManager and all TaskManagers.
# Use "hdfs://" for HDFS setups, "file://" for UNIX/POSIX-compliant file systems,
# "S3://" for S3 file system.
state.backend.fs.checkpointdir: file:///tmp/flink-backend/checkpoints
```

Enabling checkpoints

Every application need to explicitly enable checkpoints:

```
long checkpointInterval = 5000; // every 5 seconds

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.enableCheckpointing(checkpointInterval);
```

You can optionally specify a *checkpoint mode*. If not, it default to *exactly once*:

```
env.enableCheckpointing(checkpointInterval, CheckpointingMode.AT_LEAST_ONCE);
```

The checkpointing mode defines what consistency guarantees the system gives in the presence of failures. When checkpointing is activated, the data streams are replayed such that lost parts of the processing are repeated. With `EXACTLY_ONCE`, the system draws checkpoints such that a recovery behaves as if the operators/functions see each record "exactly once". With `AT_LEAST_ONCE`, the checkpoints are drawn in a simpler fashion that typically encounters some duplicates upon recovery.

The code

Here is a simple flink application using a stateful mapper with an `Integer` managed state. You can play with the `checkpointEnable`, `checkpointInterval` and `checkpointMode` variables to see their effect:

```
public class CheckpointExample {

    private static Logger LOG = LoggerFactory.getLogger(CheckpointExample.class);
    private static final String KAFKA_BROKER = "localhost:9092";
    private static final String KAFKA_INPUT_TOPIC = "input-topic";
    private static final String KAFKA_GROUP_ID = "flink-stackoverflow-checkpointer";
    private static final String CLASS_NAME = CheckpointExample.class.getSimpleName();


    public static void main(String[] args) throws Exception {

        // play with them
        boolean checkpointEnable = false;
        long checkpointInterval = 1000;
        CheckpointingMode checkpointMode = CheckpointingMode.EXACTLY_ONCE;

        // -------------------------------------------------

        LOG.info(CLASS_NAME + ": starting...");
        final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        // kafka source
        // https://ci.apache.org/projects/flink/flink-docs-release-
1.2/dev/connectors/kafka.html#kafka-consumer
        Properties prop = new Properties();
        prop.put("bootstrap.servers", KAFKA_BROKER);
        prop.put("group.id", KAFKA_GROUP_ID);
        prop.put("auto.offset.reset", "latest");
        prop.put("enable.auto.commit", "false");

        FlinkKafkaConsumer09<String> source = new FlinkKafkaConsumer09<>(
                KAFKA_INPUT_TOPIC, new SimpleStringSchema(), prop);

        // checkpoints
        // internals: https://ci.apache.org/projects/flink/flink-docs-
master/internals/stream_checkpointing.html#checkpointing
        // config: https://ci.apache.org/projects/flink/flink-docs-release-
1.3/dev/stream/checkpointing.html
        if (checkpointEnable) env.enableCheckpointing(checkpointInterval, checkpointMode);

        env
                .addSource(source)
                .keyBy((any) -> 1)
                .flatMap(new StatefulMapper())
                .print();

        env.execute(CLASS_NAME);
    }

        /* ************************************************************
```

```
         * Stateful mapper
         * (cf. https://ci.apache.org/projects/flink/flink-docs-release-
1.3/dev/stream/state.html)
         * ***************************************************************/

    public static class StatefulMapper extends RichFlatMapFunction<String, String> {
        private transient ValueState<Integer> state;

        @Override
        public void flatMap(String record, Collector<String> collector) throws Exception {
            // access the state value
            Integer currentState = state.value();

            // update the counts
            currentState += 1;
            collector.collect(String.format("%s: (%s,%d)",
                    LocalDateTime.now().format(ISO_LOCAL_DATE_TIME), record, currentState));
            // update the state
            state.update(currentState);
        }

        @Override
        public void open(Configuration parameters) throws Exception {
            ValueStateDescriptor<Integer> descriptor =
                    new ValueStateDescriptor<>("CheckpointExample",
TypeInformation.of(Integer.class), 0);
            state = getRuntimeContext().getState(descriptor);
        }
    }
}
```

Running the example and simulating failure

To be able to check the checkpoints, you need to start a `cluster`. The easier way is to use the `start-cluster.sh` script in the `flink/bin` directory:

```
start-cluster.sh
Starting cluster.
[INFO] 1 instance(s) of jobmanager are already running on virusnest.
Starting jobmanager daemon on host virusnest.
Password:
Starting taskmanager daemon on host virusnest.
```

Now, package your app and submit it to flink:

```
mvn clean package
flink run target/flink-checkpoints-test.jar -c CheckpointExample
```

Create some data:

```
kafka-console-producer --broker-list localhost:9092 --topic input-topic
a
b
c
^D
```

The output should be available in `flink/logs/flink-<user>-jobmanager-0-<host>.out`. For example:

```
tail -f flink/logs/flink-Derlin-jobmanager-0-virusnest.out
2017-03-17T08:21:51.249: (a,1)
2017-03-17T08:21:51.545: (b,2)
2017-03-17T08:21:52.363: (c,3)
```

To test the checkpoints, simply kill the taskmanager (this will emulate a failure), produce some data and start a new one:

```
# killing the taskmanager
ps -ef | grep -i taskmanager
kill <taskmanager PID>

# starting a new taskmanager
flink/bin/taskmanager.sh start
```

Note: when starting a new taskmanager, it will use another log file, namely `flink/logs/flink-<user>-jobmanager-1-<host>.out` (notice the integer increment).

What to expect

- *checkpoints disabled*: if you produce data during the failure, they will be definitely lost. But surprisingly enough, the counters will be right !
- *checkpoints enabled*: no data loss anymore (and correct counters).
- *checkpoints with at-least-once mode*: you may see duplicates, especially if you set a checkpoint interval to a high number and kill the taskmanager multiple times

Read Checkpointing online: https://riptutorial.com/apache-flink/topic/9465/checkpointing

Chapter 3: Consume data from Kafka

Examples

KafkaConsumer example

`FlinkKafkaConsumer` let's you consume data from one or more kafka topics.

versions

The consumer to use depends on your kafka distribution.

- `FlinkKafkaConsumer08`: uses the old `SimpleConsumer` API of Kafka. Offsets are handled by Flink and committed to zookeeper.
- `FlinkKafkaConsumer09`: uses the new Consumer API of Kafka, which handles offsets and rebalance automatically.
- `FlinkKafkaProducer010`: this connector supports Kafka messages with timestamps both for producing and consuming (useful for window operations).

usage

The binaries are not part of flink core, so you need to import them:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-kafka-0.${kafka.version}_2.10</artifactId>
  <version>RELEASE</version>
</dependency>
```

The constructor takes three arguments:

- one or more topic to read from
- a deserialization schema telling Flink how to interpret/decode the messages
- kafka consumer configuration properties. Those are the same as a "regular" kafka consumer. The minimum required are:
    - `bootstrap.servers`: a comma separated list of Kafka brokers in the form ip:port. For version 8, use `zookeeper.connect` (list of zookeeper servers) instead
    - `group.id`: the id of the consumer group (see kafka documentation for more details)

In Java:

```
Properties properties = new Properties();
properties.put("group.id", "flink-kafka-example");
properties.put("bootstrap.servers", "localhost:9092");

DataStream<String> inputStream = env.addSource(
        new FlinkKafkaConsumer09<>(
```

```
            kafkaInputTopic, new SimpleStringSchema(), properties));
```

In scala:

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

inputStream = env.addSource(
        new FlinkKafkaConsumer08[String](
            "topic", new SimpleStringSchema(), properties))
```

During development, you can use the kafka properties `enable.auto.commit=false` and `auto.offset.reset=earliest` to reconsume the same data everytime you launch your pogram.

Fault tolerance

As explained in the docs,

> With Flink's checkpointing enabled, the Flink Kafka Consumer will consume records from a topic and periodically checkpoint all its Kafka offsets, together with the state of other operations, in a consistent manner. In case of a job failure, Flink will restore the streaming program to the state of the latest checkpoint and re-consume the records from Kafka, starting from the offsets that where stored in the checkpoint.
>
> The interval of drawing checkpoints therefore defines how much the program may have to go back at most, in case of a failure.

To use fault tolerant Kafka Consumers, you need to enable checkpointing at the execution environment using the `enableCheckpointing` method:

```
final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.enableCheckpointing(5000); // checkpoint every 5 seconds
```

Built-in deserialization schemas

SimpleStringSchema: `SimpleStringSchema` deserializes the message as a string. In case your messages have keys, the latter will be ignored.

```
new FlinkKafkaConsumer09<>(kafkaInputTopic, new SimpleStringSchema(), prop);
```

JSONDeserializationSchema

`JSONDeserializationSchema` deserializes json-formatted messages using *jackson* and returns a stream of `com.fasterxml.jackson.databind.node.ObjectNode` objects. You can then use the `.get("property")` method to access fields. Once again, keys are ignored.

```
new FlinkKafkaConsumer09<>(kafkaInputTopic, new JSONDeserializationSchema(), prop);
```

JSONKeyValueDeserializationSchema

`JSONKeyValueDeserializationSchema` is very similar to the previous one, but deals with messages with json-encoded keys AND values.

```
boolean fetchMetadata = true;
new FlinkKafkaConsumer09<>(kafkaInputTopic, new
JSONKeyValueDeserializationSchema(fetchMetadata), properties);
```

The `ObjectNode` returned contains the following fields:

- `key`: all the fields present in the key
- `value`: all the message fields
- (optional) `metadata`: exposes the `offset`, `partition` and `topic` of the message (pass `true` to the constructor in order to fetch metadata as well).

For example:

```
kafka-console-producer --broker-list localhost:9092 --topic json-topic \
    --property parse.key=true \
    --property key.separator=|
{"keyField1": 1, "keyField2": 2} | {"valueField1": 1, "valueField2" : {"foo": "bar"}}
^C
```

Will be decoded as:

```
{
    "key":{"keyField1":1,"keyField2":2},
    "value":{"valueField1":1,"valueField2":{"foo":"bar"}},
    "metadata":{
        "offset":43,
        "topic":"json-topic",
        "partition":0
    }
}
```

Kafka partitions and Flink parallelism

In kafka, each consumer from the same consumer group gets assigned one or more partitions. Note that it is not possible for two consumers to consume from the same partition. The number of flink consumers depends on the flink parallelism (defaults to 1).

There are three possible cases:

1. **kafka partitions == flink parallelism**: this case is ideal, since each consumer takes care of one partition. If your messages are balanced between partitions, the work will be evenly spread across flink operators;

2. **kafka partitions < flink parallelism**: some flink instances won't receive any messages. To avoid that, you need to call `rebalance` on your input stream *before any operation*, which causes data to be re-partitioned:

```
inputStream = env.addSource(new FlinkKafkaConsumer10("topic", new SimpleStringSchema(),
properties));

inputStream
    .rebalance()
    .map(s -> "message" + s)
    .print();
```

3. **kafka partitions > flink parallelism**: in this case, some instances will handle multiple partitions. Once again, you can use `rebalance` to spread messages evenly accross workers.

Read Consume data from Kafka online: https://riptutorial.com/apache-flink/topic/9003/consume-data-from-kafka

Chapter 4: How to define a custom (de)serialization schema

Introduction

Schemas are used by some connectors (Kafka, RabbitMQ) to turn messages into Java objects and vice-versa.

Examples

Custom Schema Example

To use a custom schema, all you need to do is implement one of the `SerializationSchema` or `DeserializationSchema` interface.

```
public class MyMessageSchema implements DeserializationSchema<MyMessage>,
SerializationSchema<MyMessage> {

    @Override
    public MyMessage deserialize(byte[] bytes) throws IOException {
        return MyMessage.fromString(new String(bytes));
    }

    @Override
    public byte[] serialize(MyMessage myMessage) {
        return myMessage.toString().getBytes();
    }

    @Override
    public TypeInformation<MyMessage> getProducedType() {
        return TypeExtractor.getForClass(MyMessage.class);
    }

    // Method to decide whether the element signals the end of the stream.
    // If true is returned the element won't be emitted.
    @Override
    public boolean isEndOfStream(MyMessage myMessage) {
        return false;
    }
}
```

The `MyMessage` class is defined as follow:

```
public class MyMessage{

    public int id;
    public String payload;
    public Date timestamp;

    public MyMessage(){}
```

```
    public static MyMessage fromString( String s ){
        String[] tokens = s.split( "," );
        if(tokens.length != 3) throw new RuntimeException( "Invalid record: " + s );

        try{
            MyMessage message = new MyMessage();
            message.id = Integer.parseInt(tokens[0]);
            message.payload = tokens[1];
            message.timestamp = new Date( Long.parseLong(tokens[0]));
            return message;
        }catch(NumberFormatException e){
            throw new RuntimeException("Invalid record: " + s);
        }
    }

    public String toString(){
        return String.format("%d,%s,%dl", id, payload, timestamp.getTime());
    }
}
```

Read How to define a custom (de)serialization schema online: https://riptutorial.com/apache-flink/topic/9004/how-to-define-a-custom--de-serialization-schema

Chapter 5: logging

Introduction

This topic shows how to use and configure logging (log4j) in Flink applications.

Examples

Using a logger in your code

Add the `slf4j` dependency to your `pom.xml`:

```
<properties>
    <slf4j.version>1.7.21</slf4j.version>
</properties>

<!-- ... -->

<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>${slf4j.version}</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>${slf4j.version}</version>
</dependency>
```

Create a logger object for use in your class:

```
private Logger LOGGER = LoggerFactory.getLogger(FlinkApp.class);
```

In classes that need to be serialized, such as subclasses of `RichMapFunction`, don't forget to declare `LOGGER` as `transient`:

```
private transient Logger LOG = LoggerFactory.getLogger(MyRichMapper.class);
```

In your code, use `LOGGER` as usual. Use placeholders (`{}`) to format objects and such:

```
LOGGER.info("my app is starting");
LOGGER.warn("an exception occurred processing {}", record, exception);
```

Logging configuration

Local mode

In local mode, for example when running your application from an IDE, you can configure `log4j` as usual, i.e. by making a `log4j.properties` available in the classpath. An easy way in maven is to create `log4j.properties` in the `src/main/resources` folder. Here is an example:

```
log4j.rootLogger=INFO, console

# patterns:
#  d = date
#  c = class
#  F = file
#  p = priority (INFO, WARN, etc)
#  x = NDC (nested diagnostic context) associated with the thread that generated the logging
event
#  m = message


# Log all infos in the console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{dd/MM/yyyy HH:mm:ss.SSS} %5p [%-10c] %m%n

# Log all infos in flink-app.log
log4j.appender.file=org.apache.log4j.FileAppender
log4j.appender.file.file=flink-app.log
log4j.appender.file.append=false
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{dd/MM/yyyy HH:mm:ss.SSS} %5p [%-10c] %m%n

# suppress info messages from flink
log4j.logger.org.apache.flink=WARN
```

Standalone mode

> In standalone mode, the actual configuration used is not the one in your `jar` file. This is because Flink has it own configuration files, which take precedence over your own.

Default files: Flink ships with the following default properties files:

- `log4j-cli.properties`: Used by the Flink command line client (e.g. `flink run`) (not code executed on the cluster)
- `log4j-yarn-session.properties`: Used by the Flink command line client when starting a YARN session (`yarn-session.sh`)
- `log4j.properties`: JobManager/Taskmanager logs (both standalone and YARN)

Note that `${log.file}` default to `flink/log`. It can be overridden in `flink-conf.yaml`, by setting `env.log.dir`,

> `env.log.dir` defines the directory where the Flink logs are saved. It has to be an absolute path.

Log location: the logs are *local*, i.e. they are produced in the machine(s) running the JobManager(s) / Taskmanager(s).

Yarn: when running Flink on Yarn, you have to rely on the logging capabilities of Hadoop YARN. The most useful feature for that is the YARN log aggregation. To enable it, set the `yarn.log-aggregation-enable` property to `true` in the `yarn-site.xml file`. Once that is enabled, you can retrieve all log files of a (failed) YARN session using:

```
yarn logs -applicationId <application ID>
```

Unfortunately, logs are available *only after a session stopped running*, for example after a failure.

Using different configuration(s) for each application

In case you need different settings for your various applications, there is (as of Flink 1.2) no easy way to do that.

If you use the *one-yarn-cluster-per-job* mode of flink (i.e. you launch your scripts with: `flink run -m yarn-cluster` ...), here is a workaround :

1. create a `conf` directory somewhere near your project
2. create symlinks for all files in `flink/conf`:

   ```
   mkdir conf
   cd conf
   ln -s flink/conf/* .
   ```

3. replace the symlink `log4j.properties` (or any other file you want to change) by your own configuration
4. before launching your job, run

   ```
   export FLINK_CONF_DIR=/path/to/my/conf
   ```

Depending on your version of flink, you might need to edit the file `flink/bin/config.sh`. If your run accross this line:

```
FLINK_CONF_DIR=$FLINK_ROOT_DIR_MANGLED/conf
```

change it with:

```
if [ -z "$FLINK_CONF_DIR" ]; then
    FLINK_CONF_DIR=$FLINK_ROOT_DIR_MANGLED/conf;
fi
```

Flink-on-Yarn workaround: get logs in real-time with rsyslog

Yarn does not by default aggregate logs before an application finishes, which can be problematic with streaming jobs that don't even terminate.

A workaround is to use `rsyslog`, which is available on most linux machines.

First, allow incoming udp requests by uncommenting the following lines in `/etc/rsyslog.conf`:

```
$ModLoad imudp
$UDPServerRun 514
```

Edit your `log4j.properties` (see the other examples on this page) to use `SyslogAppender`:

```
log4j.rootLogger=INFO, file

# TODO: change package logtest to your package
log4j.logger.logtest=INFO, SYSLOG

# Log all infos in the given file
log4j.appender.file=org.apache.log4j.FileAppender
log4j.appender.file.file=${log.file}
log4j.appender.file.append=false
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=bbdata: %d{yyyy-MM-dd HH:mm:ss,SSS} %-5p %-60c %x
- %m%n

# suppress the irrelevant (wrong) warnings from the netty channel handler
log4j.logger.org.jboss.netty.channel.DefaultChannelPipeline=ERROR, file

# rsyslog
# configure Syslog facility SYSLOG appender
# TODO: replace host and myTag by your own
log4j.appender.SYSLOG=org.apache.log4j.net.SyslogAppender
log4j.appender.SYSLOG.syslogHost=10.10.10.102
log4j.appender.SYSLOG.port=514
#log4j.appender.SYSLOG.appName=bbdata
log4j.appender.SYSLOG.layout=org.apache.log4j.EnhancedPatternLayout
log4j.appender.SYSLOG.layout.conversionPattern=myTag: [%p] %c:%L - %m %throwable %n
```

The layout is important, because rsyslog treats a newline as a new log entry. Above, newlines (in stacktraces for example) will be skipped. If you really want multiline/tabbed logs to work "normally", edit `rsyslog.conf` and add:

```
$EscapeControlCharactersOnReceive off
```

The use of `myTag:` at the beginning of the `conversionPattern` is useful if you want to redirect all your logs into a specific file. To do that, edit `rsyslog.conf` and add the following rule:

```
if $programname == 'myTag' then /var/log/my-app.log
& stop
```

Read logging online: https://riptutorial.com/apache-flink/topic/9713/logging

Chapter 6: Savepoints and externalized checkpoints

Introduction

Savepoints are *"fat", externally stored checkpoints* that allow us to resume a stateful flink program after a permanent failure, a cancelation or a code update. Before Flink 1.2 and the introduction of *externalized checkpoints*, savepoints needed to be triggered explicitly.

Examples

Savepoints: requirements and preliminary notes

A savepoint stores two things: (a) the positions of all datasources, (b) the states of operators. Savepoints are useful in many circonstances:

- slight application code updates
- Flink update
- changes in parallelism
- ...

As of **version 1.3** (also valid for earlier version):

- checkpoint **must be enabled** for the savepoints to be possible. If you forget to explicitly enable checkpoint using:

  ```
  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
  env.enableCheckpointing(checkpointInterval);
  ```

  you will get:

  ```
  java.lang.IllegalStateException: Checkpointing disabled. You can enable it via the
  execution environment of your job
  ```

- when using window operations, it is crucial to use event-time (vs ingestion or processing time) to yield proper results;

- to be able to upgrade a program and reuse savepoints, **manual uid must be set**. This is because, by default, Flink changes the operator's UID after any change in their code;

- Chained operators are identified by the ID of the first task. It’s not possible to manually assign an ID to an intermediate chained task, e.g. in the chain [a -> b -> c] only a can have its ID assigned manually, but not b or c. To work around this, you can manually define the task chains. If you rely on the automatic ID assignment, a change in the chaining behaviour will also change the IDs (see point above).

More info is available in the FAQ.

Savepoints

Configuration

The configuration is in the file `flink/conf/flink-conf.yaml` (under Mac OSX via homebrew, it is `/usr/local/Cellar/apache-flink/1.1.3/libexec/conf/flink-conf.yaml`).

Flink < 1.2: The configuration is very similar to the checkpoints configuration (topic available). The only difference is that it makes no sense to define an in-memory savepoint backend, since we need the savepoints to persist after Flink's shutdown.

```
# Supported backends: filesystem, <class-name-of-factory>
savepoints.state.backend: filesystem
```

```
# Use "hdfs://" for HDFS setups, "file://" for UNIX/POSIX-compliant file systems,
# (or any local file system under Windows), or "S3://" for S3 file system.
# Note: must be accessible from the JobManager and all TaskManagers !
savepoints.state.backend.fs.checkpointdir: file:///tmp/flink-backend/savepoints
```

Note: If you don't specify a backend, the default backend is *jobmanager*, meaning that your savepoints will disappear once the cluster is shutdown. This is useful for debug only.

Flink 1.2+: as explained in this jira ticket, allowing a savepoint to be saved in the jobmanager's memory makes little sense. Since Flink 1.2, savepoints are necessarily stored into files. The above configuration has been replaced by:

```
# Default savepoint target directory
state.savepoints.dir: hdfs:///flink/savepoints
```

Usage

Getting the job ID

To trigger a savepoint, all you need is the job ID of the application. The job ID is printed in the command line when you launch the job or can be retrieved later using `flink list`:

```
flink list
Retrieving JobManager.
Using address localhost/127.0.0.1:6123 to connect to JobManager.
------------------ Running/Restarting Jobs -------------------
17.03.2017 11:44:03 : 196b8ce6788d0554f524ba747c4ea54f : CheckpointExample (RUNNING)
--------------------------------------------------------------
No scheduled jobs.
```

Triggering a savepoint

To trigger a savepoint, use `flink savepoint <jobID>`:

```
flink savepoint 196b8ce6788d0554f524ba747c4ea54f
Retrieving JobManager.
Using address /127.0.0.1:6123 to connect to JobManager.
Triggering savepoint for job 196b8ce6788d0554f524ba747c4ea54f.
Waiting for response...
Savepoint completed. Path: file:/tmp/flink-backend/savepoints/savepoint-a40111f915fc
You can resume your program from this savepoint with the run command.
```

Note that you can also provide a target directory as a second argument, it will override the default one defined in `flink/bin/flink-conf.yaml`.

In Flink 1.2+, it is also possible to cancel a job AND do a savepoint at the same time, using the `-s` option:

```
flink cancel -s 196b8ce6788d0554f524ba747c4ea54f # use default savepoints dir
flink cancel -s hdfs:///savepoints 196b8ce6788d0554f524ba747c4ea54f # specify target dir
```

Note: it is possible to move a savepoint, but do not rename it !

Resuming from a savepoint

To resume from a specific savepoint, use the `-s [savepoint-dir]` option of the `flink run` command:

```
flink run -s /tmp/flink-backend/savepoints/savepoint-a40111f915fc app.jar
```

Specifying operator UID

To be able to resume from a savepoint after a code change, you must ensure that the new code uses the same UID for operator. To manually assign a UID, call the `.uid(<name>)` fonction right after the operator:

```
env
    .addSource(source)
    .uid(className + "-KafkaSource01")
    .rebalance()
    .keyBy((node) -> node.get("key").asInt())
    .flatMap(new StatefulMapper())
    .uid(className + "-StatefulMapper01")
    .print();
```

Externalized checkpoints (Flink 1.2+)

Before 1.2, the only way to persist state/retain a checkpoint after a job termination/cancellation/persistant failure was through a savepoint, which is triggered manually. Version 1.2 introduced persistent checkpoints.

Persistent checkpoints behave very much like regular periodic checkpoints except the following differences:

1. They persist their meta data into a persistant storage (like savepoints).

2. They are not discarded when the owning job fails permanently. Furthermore, they can be configured to not be discarded when the job is cancelled.

It is thus very similar to savepoints; in fact, savepoints are just externalized checkpoints with a bit more information.

Important note: At the moment, Flink's checkpoint coordinator only retains the last successfully completed checkpoint. This means that whenever a new checkpoint completes then the last completed checkpoint will be discarded. This also applies to externalized checkpoints.

Configuration

Where the metadata about [externalized] checkpoints are stored is configured in `flink-conf.yaml` (and cannot be overriden through code):

```
# path to the externalized checkpoints
state.checkpoints.dir: file:///tmp/flink-backend/ext-checkpoints
```

Note that this directory *only contains the checkpoint metadata* required to restore the checkpoint. The actual checkpoint files are still stored in their configured directory (i.e. `state.bachend.fs.checkpointdir` property).

Usage

You need to explicitly enable external checkpoints in the code using the `getCheckpointConfig()` method of the streaming environment:

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
// enable regular checkpoints
env.enableCheckpointing(5000); // every 5 sec.
// enable externalized checkpoints
env.getCheckpointConfig()

.enableExternalizedCheckpoints(CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);
```

The available `ExternalizedCheckpointCleanup` modes are:

- `RETAIN_ON_CANCELLATION`: the last checkpoint and its metadata are kept on job cancellation; it is your responsibility to clean up afterwards.
- `DELETE_ON_CANCELLATION`: the last checkpoint is deleted upon cancellation, meaning it is only available if the application fails.

To resume from an externalized checkpoint, use the savepoint syntax. For example:

```
flink run -s /tmp/flink-backend/ext-checkpoints/savepoint-02d0cf7e02ea app.jar
```

Read Savepoints and externalized checkpoints online: https://riptutorial.com/apache-

flink/topic/9466/savepoints-and-externalized-checkpoints

Chapter 7: Table API

Examples

Maven dependencies

To use the Table API, add `flink-table` as a maven dependency (in addition to `flink-clients` and `flink-core`):

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-table_2.11</artifactId>
    <version>1.1.4</version>
</dependency>
```

Ensure that the scala version (here 2.11) is compatible with your system.

Simple aggregation from a CSV

Given the CSV file `peoples.csv`:

```
1,Reed,United States,Female
2,Bradley,United States,Female
3,Adams,United States,Male
4,Lane,United States,Male
5,Marshall,United States,Female
6,Garza,United States,Male
7,Gutierrez,United States,Male
8,Fox,Germany,Female
9,Medina,United States,Male
10,Nichols,United States,Male
11,Woods,United States,Male
12,Welch,United States,Female
13,Burke,United States,Female
14,Russell,United States,Female
15,Burton,United States,Male
16,Johnson,United States,Female
17,Flores,United States,Male
18,Boyd,United States,Male
19,Evans,Germany,Male
20,Stephens,United States,Male
```

We want to count people by country and by country+gender:

```
public class TableExample{
    public static void main( String[] args ) throws Exception{
        // create the environments
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
        final BatchTableEnvironment tableEnv = TableEnvironment.getTableEnvironment( env );

        // get the path to the file in resources folder
        String peoplesPath = TableExample.class.getClassLoader().getResource( "peoples.csv"
```

```
).getPath();
        // load the csv into a table
        CsvTableSource tableSource = new CsvTableSource(
                peoplesPath,
                "id,last_name,country,gender".split( "," ),
                new TypeInformation[]{ Types.INT(), Types.STRING(), Types.STRING(),
Types.STRING() } );
        // register the table and scan it
        tableEnv.registerTableSource( "peoples", tableSource );
        Table peoples = tableEnv.scan( "peoples" );

        // aggregation using chain of methods
        Table countriesCount = peoples.groupBy( "country" ).select( "country, id.count" );
        DataSet<Row> result1 = tableEnv.toDataSet( countriesCount, Row.class );
        result1.print();

        // aggregation using SQL syntax
        Table countriesAndGenderCount = tableEnv.sql(
                "select country, gender, count(id) from peoples group by country, gender" );

        DataSet<Row> result2 = tableEnv.toDataSet( countriesAndGenderCount, Row.class );
        result2.print();
    }
}
```

The results are:

```
Germany,2
United States,18

Germany,Male,1
United States,Male,11
Germany,Female,1
United States,Female,7
```

Join tables example

In addition to `peoples.csv` (see *simple aggregation from a CSV*) we have two more CSVs representing products and sales.

`sales.csv` (people_id, product_id):

```
19,5
6,4
10,4
2,4
8,1
19,2
8,4
5,5
13,5
4,4
6,1
3,3
8,3
17,2
6,2
```

```
1,2
3,5
15,5
3,3
6,3
13,2
20,4
20,2
```

`products.csv` (id, name, price):

```
1,Loperamide,47.29
2,pain relief pm,61.01
3,Citalopram,48.13
4,CTx4 Gel 5000,12.65
5,Namenda,27.67
```

We want to get the name and product for each sale of more than 40$:

```
public class SimpleJoinExample{
    public static void main( String[] args ) throws Exception{

        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
        final BatchTableEnvironment tableEnv = TableEnvironment.getTableEnvironment( env );

        String peoplesPath = TableExample.class.getClassLoader().getResource( "peoples.csv"
).getPath();
        String productsPath = TableExample.class.getClassLoader().getResource( "products.csv"
).getPath();
        String salesPath = TableExample.class.getClassLoader().getResource( "sales.csv"
).getPath();

        Table peoples = csvTable(
                tableEnv,
                "peoples",
                peoplesPath,
                "pe_id,last_name,country,gender",
                new TypeInformation[]{ Types.INT(), Types.STRING(), Types.STRING(),
Types.STRING() } );

        Table products = csvTable(
                tableEnv,
                "products",
                productsPath,
                "prod_id,product_name,price",
                new TypeInformation[]{ Types.INT(), Types.STRING(), Types.FLOAT() } );

        Table sales = csvTable(
                tableEnv,
                "sales",
                salesPath,
                "people_id,product_id",
                new TypeInformation[]{ Types.INT(), Types.INT() } );

        // here is the interesting part:
        Table join = peoples
                .join( sales ).where( "pe_id = people_id" )
                .join( products ).where( "product_id = prod_id" )
                .select( "last_name, product_name, price" )
```

```
                .where( "price < 40" );

        DataSet<Row> result = tableEnv.toDataSet( join, Row.class );
        result.print();

    }//end main


    public static Table csvTable( BatchTableEnvironment tableEnv, String name, String path,
String header,
                                  TypeInformation[]
                                          typeInfo ){
        CsvTableSource tableSource = new CsvTableSource( path, header.split( "," ), typeInfo);
        tableEnv.registerTableSource( name, tableSource );
        return tableEnv.scan( name );
    }

}//end class
```

Note that it is important to use different names for each column, otherwise flink will complain about "ambiguous names in join".

Result:

```
Burton,Namenda,27.67
Marshall,Namenda,27.67
Burke,Namenda,27.67
Adams,Namenda,27.67
Evans,Namenda,27.67
Garza,CTx4 Gel 5000,12.65
Fox,CTx4 Gel 5000,12.65
Nichols,CTx4 Gel 5000,12.65
Stephens,CTx4 Gel 5000,12.65
Bradley,CTx4 Gel 5000,12.65
Lane,CTx4 Gel 5000,12.65
```

Using external sinks

A Table can be written to a TableSink, which is a generic interface to support different formats and file systems. A batch Table can only be written to a `BatchTableSink`, while a streaming table requires a `StreamTableSink`.

Currently, flink offers only the `CsvTableSink` interface.

Usage

In the examples above, replace:

```
DataSet<Row> result = tableEnv.toDataSet( table, Row.class );
result.print();
```

with:

```
TableSink sink = new CsvTableSink("/tmp/results", ",");
// write the result Table to the TableSink
table.writeToSink(sink);
// start the job
env.execute();
```

`/tmp/results` is a folder, because flink does parallel operations. Hence, if you have 4 processors, you will likely have 4 files in the results folder.

Also, note that we explicitely call `env.execute()`: this is necessary to start a flink job, but in the previous examples `print()` did it for us.

Read Table API online: https://riptutorial.com/apache-flink/topic/8966/table-api

Credits

S. No	Chapters	Contributors
1	Getting started with apache-flink	Community, Derlin, vdep
2	Checkpointing	Derlin
3	Consume data from Kafka	alpinegizmo, Derlin
4	How to define a custom (de)serialization schema	Derlin
5	logging	Derlin
6	Savepoints and externalized checkpoints	Derlin
7	Table API	Derlin