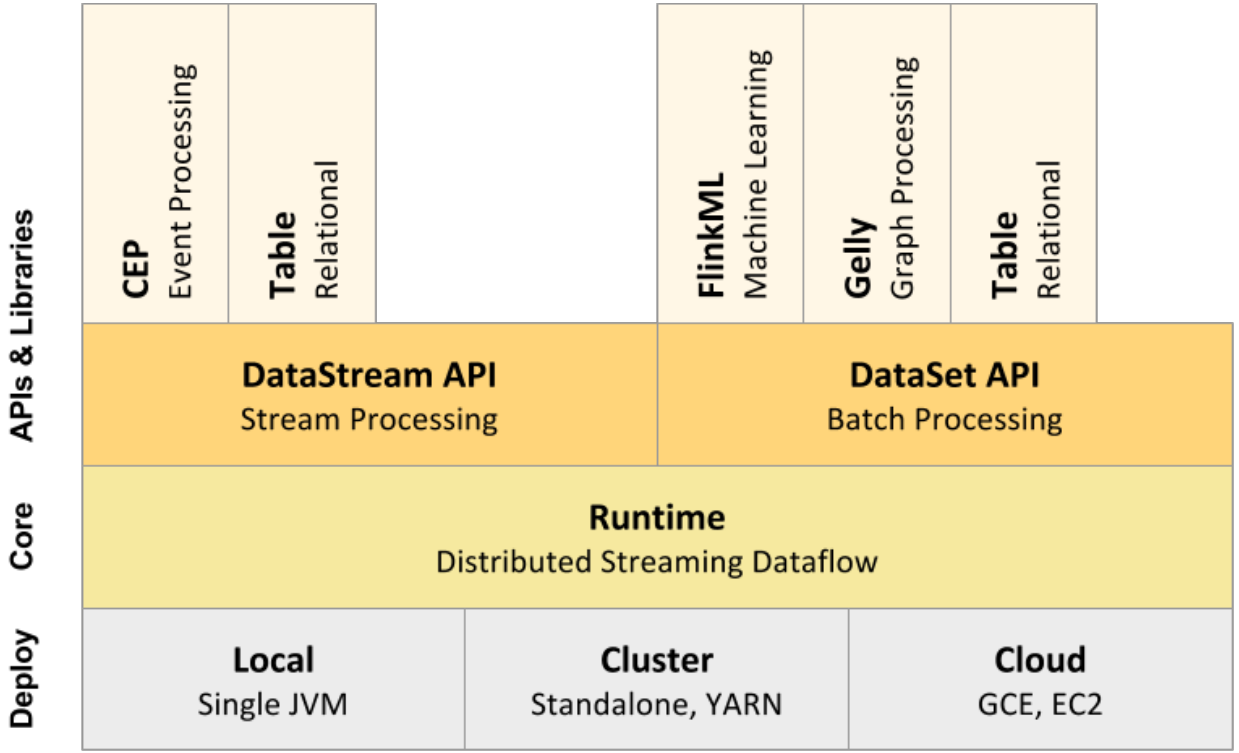
Apache Flink APIs
DataSet / DataStream

Timo Walther

Flink Committer, PMC Member
twalthr@apache.org

Overview / Flink Stack



DataSet API

Example: WordCount

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
        // split up the lines in tuples containing: (word,1)
        inputText.flatMap(new Tokenizer())
        // group by the tuple field "0"
        .groupBy(0)
        //sum up tuple field "1"
        .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

Execution Environment

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
        // split up the lines in tuples containing: (word,1)
        inputText.flatMap(new Tokenizer())
        // group by the tuple field "0"
        .groupBy(0)
        //sum up tuple field "1"
        .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

Data Sources

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
        // split up the lines in tuples containing: (word,1)
        inputText.flatMap(new Tokenizer())
        // group by the tuple field "0"
        .groupBy(0)
        //sum up tuple field "1"
        .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

Data types

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
        // split up the lines in tuples containing: (word,1)
        inputText.flatMap(new Tokenizer())
        // group by the tuple field "0"
        .groupBy(0)
        //sum up tuple field "1"
        .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

Transformations

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
        // split up the lines in tuples containing: (word,1)
        inputText.flatMap(new Tokenizer())
        // group by the tuple field "0"
        .groupBy(0)
        //sum up tuple field "1"
        .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

User functions

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
        // split up the lines in tuples containing: (word,1)
        inputText.flatMap(new Tokenizer())
        // group by the tuple field "0"
        .groupBy(0)
        //sum up tuple field "1"
        .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

DataSinks

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
        // split up the lines in tuples containing: (word,1)
        inputText.flatMap(new Tokenizer())
        // group by the tuple field "0"
        .groupBy(0)
        //sum up tuple field "1"
        .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

Execute!

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
        // split up the lines in tuples containing: (word,1)
        inputText.flatMap(new Tokenizer())
        // group by the tuple field "0"
        .groupBy(0)
        //sum up tuple field "1"
        .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

User function example: Map

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

    @Override
    public void flatMap(String value, Collector<Tuple2<String, Integer>> out) {
        // normalize and split the line
        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));
            }
        }
    }
}
```

User function example: Map

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

    @Override
    public void flatMap(String value, Collector<Tuple2<String, Integer>> out) {
        // normalize and split the line
        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));
            }
        }
    }
}
```

User function example: Reduce

```
public static class SumWords
    implements GroupReduceFunction<Tuple2<String, Integer>, Tuple2<String, Integer>> {

    @Override
    public void reduce(Iterable<Tuple2<String, Integer>> values,
                       Collector<Tuple2<String, Integer>> out) {
        int count = 0;
        String word = null;
        for (Tuple2<String, Integer> tuple : values) {
            word = tuple.f0;
            count++;
        }
        out.collect(new Tuple2<String, Integer>(word, count));
    }

}
```

Important Operators: .map()

```
// Takes one element and produces one element.
DataSet<Integer> tokenized = text.map(new MapFunction<String, Integer>() {
            @Override
            public Integer map(String value) {
                return Integer.parseInt(value);
            }
});

DataSet<ExperimentResult> converted = text.map(
              new MapFunction<String, ExperimentResult>() { ... });

static class ExperimentResult {
    public String name;
    public Tuple2<String, Long>[] parameters;
    public int result;
}
```

.flatMap(), .filter(),

```
// Takes one element and produces zero, one, or more elements.
data.flatMap(new FlatMapFunction<String, String>() {
            public void flatMap(String value, Collector<String> out) {
                for (String s : value.split(" ")) {
                  out.collect(s);
                }
            }
});

// Retains those elements for which the function returns true.
data.filter(new FilterFunction<Integer>() {
          public boolean filter(Integer value) { return value > 1000; }
});
```

.join()

```
DataSet<ExperimentResult> input1= ...
DataSet<ExperimentResult> input2= ...

// Joins two data sets by creating all pairs of elements that are equal on
// their keys.
DataSet<Tuple2<ExperimentResult, ExperimentResult>> =
            input1.join(input2)
               .where("name")       // key of the first input
               .equalTo("name");    // key of the second input

DataSet<ExperimentResult> =
            input1.join(input2)
               .where("name")       // key of the first input
               .equalTo("name")     // key of the second input
               .with(...);
```

.reduce(), .reduceGroup()

```
// Combines a group of elements into a single element.
number.reduce(new ReduceFunction<Integer> {
              public Integer reduce(Integer a, Integer b) { return a + b; }
});

// Combines a group of elements into one or more elements.
data
    .groupBy("name")
    .reduceGroup(new GroupReduceFunction<Integer, Integer> {
      public void reduce(Iterable<Integer> values, Collector<Integer> out) {
        int sum = 0;
        for (Integer i : values) sum++;
        out.collect(sum);
      }
});
```

DataStream API

Example: Window WordCount

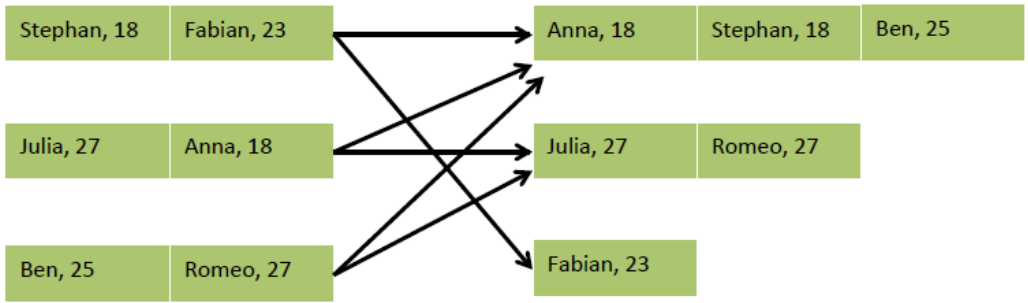
```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
    // configure event time
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

    DataStream<Tuple2<String, Integer>> counts = env
        // read stream of words from socket
        .socketTextStream("localhost", 9999)
        // split up the lines in tuples containing: (word,1)
        .flatMap(new Splitter())
        // key stream by the tuple field "0”
        .keyBy(0)
        // compute counts every 5 minutes
        .timeWindow(Time.minutes(5))
        //sum up tuple field "1"
        .sum(1);

    // print result in command line and execute program
    counts.print();
    env.execute("Socket WordCount Example");
}
```

DataSources

```
// read text socket from port
DataStream<String> socketLines = env.socketTextStream("localhost", 9999);

// read a text file ingesting new elements every 100 milliseconds
DataStream<String> localLines =
    env.readFileStream("/path/to/file", 100, WatchType.PROCESS_ONLY_APPENDED);

// read data stream from custom source function
DataStream<<Tuple2<Long, String> stream = env.addSource(new MySourceFunction());

// read data from many stream serving systems such as Apache Kafka
Properties properties = new Properties();
properties.setProperty("bootstrap.servers", "localhost:9092");
properties.setProperty("zookeeper.connect", "localhost:2181");
properties.setProperty("group.id", "test");
DataStream<String> stream = env
     .addSource(new FlinkKafkaConsumer08<>("topic", new SimpleStringSchema(), properties))
```

.keyBy()

```
// Organizes a DataStream by a key / partitions the data.
// All elements with the same key are processed by the same operator

// (name, age) of employees
DataStream<Tuple2<String, Integer>> passengers = …

// group by second field (age)
DataStream<Integer, Integer> grouped = passengers.keyBy(1)
```



Windows

- Aggregations on DataStreams are different from aggregations on DataSets e.g., it is not possible to count all elements of an unbounded DataStream
- DataStream aggregations make sense on windowed streams
  → Discretize streams
- Only windows on keyed stream can be processed in parallel

Windows

- **Tumbling time window**
  `.timeWindow(Time.minutes(1))`
- **Sliding time window**
  `.timeWindow(Time.minutes(1), Time.seconds(30))`
- **Tumbling count window**
  `.countWindow(100)`
- **Sliding count window**
  `.countWindow(100, 10)`

Windows

```
// (name, age) of passengers
DataStream<Tuple2<String, Integer>> passengers = …

passengers
    // group by first field (age)
    .keyBy(0)
    // window of 1 minute length triggered every 10 seconds
    .timeWindow(Time.minutes(1), Time.seconds(10))
    // apply a custom window function on window data
    // or reduce(), fold(), sum(), min(), max(), etc.
    .apply(new CountByAge());
```

Windows

```
public static class CountByAge implements WindowFunction<
    Tuple2<String, Integer>, // input type
    Tuple3<Integer, Long, Integer>, // output type
    Tuple, // key type
    TimeWindow> // window type {

    @Override
    public void apply(
        Tuple key,
        TimeWindow window,
        Iterable<Tuple2<String, Integer>> persons,
        Collector<Tuple3<Integer, Long, Integer>> out) {

        int age = ((Tuple1<Integer>)key).f0;
        int cnt = 0;

        for (Tuple2<String, Integer> p : persons) { cnt++; }
        // return (age, window-end-time, count)
        out.collect(new Tuple3<>(age, window.getEnd(), cnt));
    }
}
```

Stateful Functions

- All DataStream functions can be stateful.
  (State can be checkpointed and recovered in case of a failure.)

- Types of states:

  - ***Local State***
    Functions can register local variables to be checkpointed.

  - ***Key-Partitioned State***
    Functions on a keyed stream can access and update state scoped to the current key.

Key-Partitioned State

```
KeyedStream<Tuple2<String, String>, Tuple> keyedStream = aStream.keyBy(0);
DataStream<Long> lengths = keyedStream.map(new MapWithCounter());

public static class MapWithCounter extends RichMapFunction<Tuple2<String, String>, Long> {

    private ValueState<Long> totalLengthByKey;

    @Override
    public void open(Configuration conf) {
        totalLengthByKey = getRuntimeContext().getState("totalLengthByKey", Long.class, 0L);
    }

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

flink.apache.org
@ApacheFlink

Or follow me on Twitter: @twalthr