dataArtisans

Apache Flink® Training

DataStream API Advanced

August 26, 2015

What kind of data can Flink handle?

Type System and Keys

Note: Identical to DataSet API

Apache Flink's Type System

- Flink aims to support all data types
  - Ease of programming
  - Seamless integration with existing code

- Programs are analyzed before execution
  - Used data types are identified
  - Serializer & comparator are configured

Apache Flink's Type System

- Data types are either
  - Atomic types (like Java Primitives)
  - Composite types (like Flink Tuples)
- Composite types nest other types
- Not all data types can be used as keys!
  - Flink partitions DataStreams on keys
  - Key types must be comparable

Atomic Types

Flink Type	Java Type	Can be used as key?
BasicType	Java Primitives (Integer, String, …)	Yes
ArrayType	Arrays of Java primitives or objects	No (Yes as of 0.10)
WritableType	Implements Hadoop's Writable interface	Yes, if implements WritableComparable
GenericType	Any other type	Yes, if implements Comparable

Composite Types

- Are composed of fields with other types
  - Fields types can be atomic or composite
- Fields can be addressed as keys
  - Field type must be a key type!
- A composite type can be a key type
  - All field types must be key types!

TupleType

- Java:
  `org.apache.flink.api.java.tuple.Tuple1 to Tuple25`
- Scala:
  use default Scala tuples (1 to 22 fields)

- Tuple fields are typed

```
Tuple3<Integer, String, Double> t3 =
                            new Tuple3(1, “2”, 3.0);

val t3: (Int, String, Double) = (1, ”2”, 3.0)
```

- Tuples give the best performance

TupleType

- Define keys by field position

```
DataStream<Tuple3<Integer, String, Double>> d = …
// group on String field
d.groupBy(1);
```

- Or field names

```
// group on Double field
d.groupBy(“f2”);
```

PojoType

- Any Java class that
  - Has an empty default constructor
  - Has publicly accessible fields
    (public field or default getter & setter)

```
public class Person {
  public int id;
  public String name;
  public Person() {};
  public Person(int id, String name) {…};
}

DataStream<Person> p =
  env.fromElements(new Person(1, ”Bob”));
```

PojoType

- Define keys by field name

```
DataStream<Person> p = …
// group on “name” field
d.groupBy(“name”);
```

Scala CaseClasses

- Scala case classes are natively supported

```
case class Person(id: Int, name: String)
d: DataStream[Person] =
      env.fromElements(Person(1, “Bob”)
```

- Define keys by field name

```
// use field “name” as key
d.groupBy(“name”)
```

Composite & nested keys

```
DataStream<Tuple3<String, Person, Double>> d;
```

- Composite keys are supported

```
// group on both long fields
d.groupBy(0, 1);
```

- Nested fields can be used as types

```
// group on nested “name” field
d.groupBy(“f1.name”);
```

- Full types can be used as key using “*” wildcard

```
// group on complete nested Pojo field
d.groupBy(“f1.*”);
```

  - “*” wildcard can also be used for atomic types

KeySelectors

- Keys can be computed using KeySelectors

```
public class SumKeySelector implements
        KeySelector<Tuple2<Long, Long>, Long> {

  public Long getKey(Tuple2<Long, Long> t) {
    return t.f0 + t.f1;
 }}

DataStream<Tuple2<Long,Long>> d = …
d.groupBy(new SumKeySelector());
```

Windows and aggregates

Windows

- Aggregations on DataStreams are different from aggregations on DataSets
  - e.g., it is not possible to count all elements of a DataStream – they are infinite
- DataStream aggregations make sense on windowed streams
  - i.e., a window of the "latest" elements of a stream
- Windows can be defined on grouped and partitioned streams

Windows (2)

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

// group by second field (age) and keep last 1 minute
// worth of data sliding the window every 10 seconds
passengers
    .groupBy(1)
    .window(Time.of(1, TimeUnit.MINUTES))
    .every(Time.of(10, TimeUnit.SECONDS))
```

Types of windows

- Tumbling time window
  - `.window(Time.of(1, TimeUnit.MINUTES))`
- Sliding time window
  - `.window(Time.of(60, TimeUnit.SECONDS))`
    `.every(Time.of(10, TimeUnit.SECONDS))`
- Count-based sliding window
  - `.window(Count.of(1000))`
    `.every(Count.of(10))`

Aggregations on windowed streams

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

// group by second field (age) and keep last 1 minute
// worth of data sliding the window every 10 seconds
// count passengers
employees
    .groupBy(1)
    .window(Time.of(1, TimeUnit.MINUTES))
    .every(Time.of(10, TimeUnit.SECONDS))
    .mapWindow(new CountSameAge());
```

MapWindow

```
public static class CountSameAge implements
WindowMapFunction<Tuple2<String, Integer>, Tuple2<Integer, Integer>> {

    @Override
    public void mapWindow(Iterable<Tuple2<String, Integer>> persons,
                          Collector<Tuple2<Integer, Integer>> out) {

        Integer ageGroup = 0;
        Integer countsInGroup = 0;

        for (Tuple2<String, Integer> person : persons) {
            ageGroup = person.f1;
            countsInGroup++;
        }

        out.collect(new Tuple2<Integer, Integer>
                                            (ageGroup, countsInGroup));
    }
}
```

Operations on WindowedStreams

- mapWindow
  - Do something over the whole window
- reduceWindow
  - Apply a functional reduce function to the window
- Aggregates: sum, min, max, and others
- flatten
  - Get back a regular DataStream

Working with multiple streams

Connecting streams

```
DataStream<String> strings = …
DataStream<Integer> ints = …

// Create a ConnectedDataStream
strings.connect(ints);
```

- Sometimes several DataStreams need to be correlated with each other and share state

- You can *connect* or *join* two DataStreams

Map on connected streams

```
DataStream<String> strings = …
DataStream<Integer> ints = …

// Create a ConnectedDataStream
strings.connect(ints)
    .map(new CoMapFunction<Integer,String,Boolean> {
        @Override
        public Boolean map1 (Integer value) {
            return true;
        }
        @Override
        public Boolean map2 (String value) {
            return false;
        }
});
```

FlatMap on connected streams

```
DataStream<String> strings = …
DataStream<Integer> ints = …

// Create a ConnectedDataStream
strings.connect(ints)
    .flatMap(new CoFlatMapFunction<Integer,String,String> {
        @Override
        public void flatMap1 (Integer value, Collector<String> out) {
            out.collect(value.toString());
        }
        @Override
        public void flatMap2 (String value, Collector<String> out) {
            for (String word: value.split(" ")) {
                out.collect(word)
            }
        }
});
```

Rich functions and state

RichFunctions

- Function interfaces have only one method
  - Single abstract method (SAM)
  - Support for Java8 Lambda functions

- There is a “Rich” variant for each function.
  - RichFlatMapFunction, …
  - Additional methods
    - `open(Configuration c)`
    - `close()`
    - `getRuntimeContext()`

Note: Identical to DataSet API

RichFunctions & RuntimeContext

- RuntimeContext has useful methods:
  - `getIndexOfThisSubtask ()`
  - `getNumberOfParallelSubtasks()`
  - `getExecutionConfig()`
- Give access to partitioned state

Note: Identical to DataSet API

Stateful computations

- All DataStream transformations can be stateful
  - State is mutable and lives as long as the streaming job is running
  - State is recovered with exactly-once semantics by Flink after a failure
- You can define two kinds of state
  - Local state: each parallel task can register some local variables to take part in Flink's checkpointing
  - Partitioned by key state: an operator on a partitioned by key stream can access and update state corresponding to its key
  - Partitioned state will be available in Flink 0.10

Defining local state

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

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

     private long totalLength = 0;

     @Override
     public Long map (String value) {
          totalLength += value.length();8
          return (Long) value.length();
     }

     @Override
     public Serializable snapshotState(
               long checkpointId,
               long checkpointTimestamp) throws Exception {
          return new Long (totalLength);
     }

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

}
```

Defining partitioned state

```
DataSet<Tuple2<String,String>> aStream;
DataStream<Long> lengths = aStream.groupBy(0).map (new MapWithCounter());

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

    private OperatorState<Long> totalLengthByKey;

    @Override
    public Long map (Tuple2<String,String> value) {
        totalLengthByKey.update(totalLengthByKey.update.value() + 1);
        return (Long) value.f1.length();
    }

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

}
```

Note: Will be available in Flink 0.10

Connecting to Apache Kafka

Kafka and Flink

- “Apache Kafka is a distributed, partitioned, replicated commit log service”
- Kafka uses Apache Zookeeper for coordination
- Kafka maintains feeds of messages in categories called topics
- A Kafka topic can be read by Flink to produce a DataStream, and a DataStream can be written to a Kafka topic
- Flink coordinates with Kafka to provide recovery in the case of failures

Reading data from Kafka

- Enable checkpointing
  E.g., `env.enableCheckpointing(5000);`

- Add a DataStream source from a Kafka topic

```
Properties props = new Properties();
props.setProperty("zookeeper.connect", “localhost:2181”);
props.setProperty("bootstrap.servers", “localhost:9092”);
props.setProperty("group.id", “myGroup”);

// create a data source
DataStream<TaxiRide> rides = env.addSource(
      new FlinkKafkaConsumer082<TaxiRide>(
            “myTopic”,
            new TaxiRideSchema(),
            props)
      );
```

Writing data to Kafka

- Add a Kafka sink to a DataStream by providing
  - The broker address
  - The topic name
  - A serialization schema

```
DataStream<String> aStream = …
aStream.addSink(
    new KafkaSink<String>(
        “localhost:9092”, // default local broker
        “myTopic”,
        new SimpleStringSchema));
```

More API features

Not covered here

- Iterations (feedback edges)
  - Very useful for Machine Learning
- More transformations
  - union, join, ...