dataArtisans

Apache Flink® Training

DataStream API Advanced

December 10, 2015

What kind of data can Flink handle?

Type System and Keys

Apache Flink's Type System

- Flink aims to support all data types
  - Ease of programming
  - Seamless integration with existing code
  - DataSet and DataStream API share the same type system!

- 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	Primitive Arrays: Yes Object Arrays: No
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 = …
// key stream by String field
d.keyBy(1);
```

- Or field names

```
// key stream by Double field
d.keyBy("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 = …
// key stream by “name” field
p.keyBy(“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

```
// key stream by field “name”
d.keyBy(“name”)
```

Composite & Nested Keys

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

- Composite keys are supported

```
// key stream by both long fields
d.keyBy(0, 1);
```

- Nested fields can be used as types

```
// key stream by nested “name” field
d.keyBy(“f1.name”);
```

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

```
// key stream by complete nested Pojo field
d.keyBy(“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.keyBy(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 an unbounded DataStream
- DataStream aggregations make sense on windowed streams
  - i.e., a finite set of stream elements
- Only windows on keyed stream can be processed in parallel

Windows (2)

```
// (age, count) of passengers
DataStream<Tuple2<Integer, 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))
    // sum values of second field (count)
    .sum(2);
```

Predefined Keyed 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)
```

Aggregations on Windowed Streams

```
// (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
    .apply(new CountByAge());
```

MapWindow

```
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));
    }
}
```

Operations on Windowed Streams

- `reduce(reduceFunction)`
  - Apply a functional reduce function to the window
- `fold(inialVal, foldFunction)`
  - Apply a functional fold function with a specified initial value to the window
- Aggregation functions
  - `sum()`, `min()`, `max()`, and others

Windows on non-keyed streams

- Windows on non-keyed streams are not parallel!

- TimeWindow (tumbling, 10 seconds)
  ```
  .timeWindowAll(Time.seconds(10))
  ```

- CountWindow (sliding, 20/10)
  ```
  .countWindowAll(20, 10)
  ```

Custom window logic

- The DataStream API allows to define very custom window logic
- Trigger
  - defines when to evaluate a window
  - whether to purge the window or not
- Evictor
  - Allows to remove elements from a window before it is evaluated
- *Careful!* This part of the API requires a good understanding of the windowing mechanism!

Working With Multiple Streams

Connecting Streams

- *Connect* two DataStreams to correlated them with each other

- Apply functions on connected streams to share state

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

ConnectedStreams<String, Integer> coStream =
    strings.connect(ints);
```

Map on Connected Streams

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

ints.connect(strings)
    .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 = …

ints.connect(strings)
    .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);
            }
        }
});
```

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<MyData> rides = env.addSource(
      new FlinkKafkaConsumer082<MyData>(
              “myTopic”,
              new MyDataSchema(),
              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 FlinkKafkaProducer<String>(
        "localhost:9092", // default local broker
        "myTopic",
        new SimpleStringSchema));
```

More API Features

Not Covered Here

- Iterations (feedback edges)
  - Enables certain machine learning algorithms
- More transformations
  - split, union, window join, ...