dataArtisans

Apache Flink® Training

DataSet API Advanced

June 15th, 2015

Agenda

- Data types and keys
- More transformations
- Further API concepts

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

- 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 groups, joins & sorts DataSets 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
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

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

- Or field names

```
// group on Double field
d.groupBy("f2").groupReduce(…);
```

PojoType

- Any Java class that
  - Has an empty default constructor
  - Has publicly accessible fields
    (Public or getter/setter)

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

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

PojoType

- Define keys by field name

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

Scala CaseClasses

- Scala case classes are natively supported

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

- Define keys by field name

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

Composite & nested keys

```
DataSet<Tuple3<String, Person, Double>> d = …
```

- Composite keys are supported

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

- Nested fields can be used as types

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

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

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

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

Join & CoGroup Keys

- Key types must match for binary operations!

```
DataSet<Tuple2<Long, String>> d1 = …
DataSet<Tuple2<Long, Long>> d2 = …

// works
d1.join(d2).where(0).equalTo(1).with(…);
// works
d1.join(d2).where(“f0”).equalTo(0).with(…);
// does not work!
d1.join(d2).where(1).equalTo(0).with(…);
```

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;
 }}

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

Getting data in and out

Advanced Sources and Sinks

Supported File Systems

- Flink build-in File Systems:
  - LocalFileSystem (file://)
  - Hadoop Distributed File System (hdfs://)
  - Amazon S3 (s3://)
  - MapR FS (maprfs://)

- Support for all Hadoop File Systems
  - NFS, Tachyon, FTP, har (Hadoop Archive), …

Input/Output Formats

- FileInputFormat
  (recursive directory scans supported)
  - DelimitedInputFormat
    - TextInputFormat (Reads text files linewise)
    - CsvInputFormat (Reads field delimited files)
  - BinaryInputFormat
  - AvroInputFormat (Reads Avro POJOs)
- JDBCInputFormat (Reads result of SQL query)
- HadoopInputFormat
  (Wraps any Hadoop InputFormat)

Hadoop Input/OutputFormats

- Support for all Hadoop I/OFormats
- Read from and write to
  - MongoDB
  - Apache Parquet
  - Apache ORC
  - Apache Kafka (for batch)
  - Compressed file formats (.gz, .zip, ...)
  - and more...

Using InputFormats

```
ExecutionEnvironment env = …

// read text file linewise
env.readTextFile(…);

// read CSV file
env.readCsvFile(…);

// read file with Hadoop FileInputFormat
env.readHadoopFile(…);

// use regular Hadoop InputFormat
env.createHadoopInput(…);

// use regular Flink InputFormat
env.createInput(…);
```

Transformations & Functions

Transformations

- DataSet Basics presented:
  - Map, FlatMap, GroupBy, GroupReduce, Join

- Reduce
- CoGroup
- Combine
- GroupSort
- AllReduce & AllGroupReduce
- Union

- see documentation for more transformations

GroupReduce (Hadoop-style)

- GroupReduceFunction gives iterator over elements of group
  - Elements are streamed (possibly from disk), not materialized in memory
  - Group size can exceed available JVM heap
- Input type and output type may be different

Reduce (FP-style)

- Reduce like in functional programming
- Less generic compared to GroupReduce
  - Function must be commutative and associative
  - Input type == Output type
- System can apply more optimizations
  - Always combinable
  - May use a hash strategy for execution (future)

Reduce (FP-style)

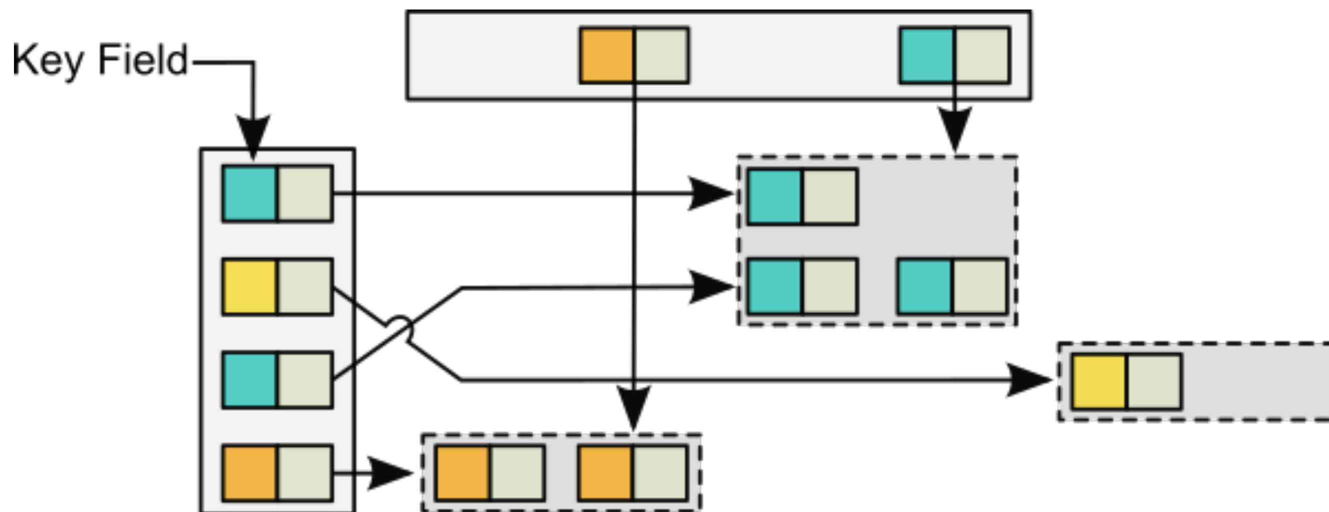
```
DataSet<Tuple2<Long,Long>> sum = data
    .groupBy(0)
    .reduce(new SumReducer());

public static class SumReducer implements
              ReduceFunction<Tuple2<Long,Long>> {

  @Override
  public Tuple2<Long,Long> reduce(
                        Tuple2<Long,Long> v1,
                        Tuple2<Long,Long> v2) {
      v1.f1 += v2.f1;
     return v1;
   }
}
```

CoGroup

- Binary operation (two inputs)
  - Groups both inputs on a key
  - Processes groups with matching keys of both inputs
- Similar to GroupReduce on two inputs



CoGroup

```
DataSet<Tuple2<Long,String>> d1 = …;
DataSet<Long> d2 = …;
DataSet<String> d3 =
   d1.coGroup(d2).where(0).equalTo(1).with(new CoGrouper());

public static class CoGrouper implements
CoGroupFunction<Tuple2<Long,String>,Long,String>{

  @Override
  public void coGroup(Iterable<Tuple2<Long,String> vs1,
                 Iterable<Long> vs2, Collector<String> out) {
    if(!vs2.iterator.hasNext()) {
      for(Tuple2<Long,String> v1 : vs1) {
        out.collect(v1.f1);
      }
    }
} }
```

Combiner

- Local pre-aggregation of data
  - Before data is sent to GroupReduce or CoGroup
  - (functional) Reduce injects combiner automatically
  - Similar to Hadoop Combiner

- Optional for semantics, crucial for performance!
  - Reduces data before it is sent over the network

Combiner WordCount Example

A B B
A C A

Map

(A, 1)
(B, 1)
(B, 1)
(A, 1)
(C, 1)
(A, 1)

Combine

(A, 3)
(C, 1)
(B, 2)

C A C
B A B

Map

(C, 1)
(A, 1)
(C, 1)
(B, 1)
(A, 1)
(B, 1)

Combine

(A, 2)
(C, 2)
(B, 2)

D A A
A B C

Map

(D, 1)
(A, 1)
(A, 1)
(A, 1)
(B, 1)
(C, 1)

Combine

(A, 3)
(C, 1)
(B, 1)
(D, 1)

Reduce

(A, 8)
(C, 4)

Reduce

(B, 5)
(D, 1)

Use a combiner

- Implement `RichGroupReduceFunction<I,O>`
  - Override `combine(Iterable<I> in, Collector<O>);`
  - Same interface as `reduce()` method
  - Annotate your `GroupReduceFunction` with `@Combinable`
  - Combiner will be automatically injected into Flink program

- Implement a `GroupCombineFunction`
  - Same interface as `GroupReduceFunction`
  - `DataSet.combineGroup()`

GroupSort

- Sort groups before they are handed to GroupReduce or CoGroup functions
  - More (resource-)efficient user code
  - Easier user code implementation
  - Comes (almost) for free
  - Aka secondary sort (Hadoop)

```
DataSet<Tuple3<Long,Long,Long> data = …;

data.groupBy(0)
    .sortGroup(1, Order.ASCENDING)
    .groupReduce(new MyReducer());
```

AllReduce / AllGroupReduce

- Reduce / GroupReduce without GroupBy
  - Operates on a single group -> Full DataSet
  - Full DataSet is sent to one machine
  - Will automatically run with parallelism of 1
- Careful with large DataSets!
  - Make sure you have a Combiner

Union

- Union two data set
  - Binary operation, same data type required
  - No duplicate elimination (SQL UNION ALL)
  - Very cheap operation

```
DataSet<Tuple2<String, Long> d1 = …;
DataSet<Tuple2<String, Long> d2 = …;

DataSet<Tuple2<String, Long> d3 =
   d1.union(d2);
```

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()`

RichFunctions & RuntimeContext

- RuntimeContext has useful methods:
  - `getIndexOfThisSubtask ()`
  - `getNumberOfParallelSubtasks()`
  - `getExecutionConfig()`

- Gives access to:
  - Accumulators
  - DistributedCache

Further API Concepts

Broadcast Variables

Example: Tag words with IDs in text corpus

Text
data set

map

map

map

broadcast (small) dictionary to all mappers

Dictionary

Broadcast variables

- register any DataSet as a broadcast variable
- available on all parallel instances

```
// 1. The DataSet to be broadcasted
DataSet<Integer> toBroadcast = env.fromElements(1, 2, 3);

// 2. Broadcast the DataSet
map().withBroadcastSet(toBroadcast, "broadcastSetName");

// 3. inside user defined function
getRuntimeContext().getBroadcastVariable("broadcastSetName");
```

Accumulators

- Lightweight tool to compute stats on data
  - Usuful to verify your assumptions about your data
  - Similar to Counters (Hadoop MapReduce)

- Build in accumulators
  - Int and Long counters
  - Histogramm

- Easily customizable

Accumulators

Example:

Count total number of words in text corpus

Text data set

map
long counter++

map
long counter++

map
long counter++

JobManager
counter =
4 +
18 +
22 =
44

Send local counts from parallel instances to JobManager

Using Accumulators

- Use accumulators to verify your assumptions about the data

```
class Tokenizer extends
            RichFlatMapFunction<String, String>> {

  @Override
  public void flatMap(String val,
                      Collector<String> out) {
     getRuntimeContext()
       .getLongCounter("elementCount").add(1L);
     // do more  stuff.
   }
}
```

Get Accumulator Results

- Accumulators are available via JobExecutionResult
  - returned by env.execute()

```
JobExecutionResult result = env.execute("WordCount");
long ec = result.getAccumulatorResult("elementCount");
```

- Accumulators are displayed
  - by CLI client
  - in the JobManager web frontend