FLIP-13 Side Outputs in Flink

Status:

Current state: *Released*

Discussion thread: http://apache-flink-mailing-list-archive.1008284.n3.nabble.com/Discuss-FLIP-13-Side-Outputs-in-Flink-td14204.html

JIRA: FLINK-4460 - Side Outputs in Flink CLOSED

Released: Flink 1.3

Motivation

Side outputs(a.k.a Multi-outputs) is one of highly requested features in high fidelity stream processing use cases. With this feature, Flink can

- Side output corrupted input data and avoid job fall into “fail -> restart -> fail” cycle
- Side output sparsely received late arriving events while issuing aggressive watermarks in window computation.

Public Interfaces

- Add CollectorWrapper in flink-core util folder, user can wrap Collector and emit side outputs with predefined outputtags

Proposed Changes

We want to introduce outputTag and support operator collect arbitrary types of records with defined output Tags. In this prototype, it demonstrated how things works in raw/hacky form.

API Changes

- **Add abstract class OutputTag (extend from TypeHint)**

  User may declare multiple output tags and use get output stream with one previously defined outputtag.

  ```
  final OutputTag<S> sideOutput1 = new OutputTag<S>(SValue) {};
  ```

- **Add internal interface org.apache.flink.util.RichCollector, expose CollectorWrapper as user facing side output collector wrapper**

  Update userFunctions using RichCollector(Or MultiCollector) the rationale behind is Collector has been used in lot of places other than stream and transformation, adding direct to Collector interface will introduce many empty methods in those classes.

  ```
  public interface RichCollector<T> extends Collector<T>{
     <S> void collect(OutputTag<S> tag, S value);
  }
  ```

  FlatMapFunction as example

  ```
  flatMap(String value, Collector<Tuple2<String, Integer>> out){

        CollectorWrapper wrapper = new CollectorWrapper<>(out);

      //out.collect(new Tuple2<String, Integer>(token, 1));

     wrapper.collect(new Tuple2<String, Integer>(token, 1));

     wrapper.collect(sideOutput1, "sideout");

  }
  ```

- **Add getOutput(OutputTag) to SingleOutputStreamOperator**

User may pass outputtag defined eearlier and get a corresponding outputstream.

There can be more than one outputtag share same type, however, getSideOutput only returns collected record with exams same outputtag type and value.

```
flatMap(..).getSideOutput(sideOutput1)
```

```
StreamGraph add virtualOutputNodes each map to single outputtag from upstream node
```

```
Stream Record and StreamEdge both add outputTag, StreamConfig stores all type of side output serializers
```

Record writer can utilize this information and compare with outputType(impl in prototype) or OutputTag (better implementation) or each Output<OUT> and just write matched stream record to channel

- **TimeStampedCollector, use RichCollector, collect Stream Record with outputTag**

Runtime changes

- **add SideOutputTransformation**

It allows good compatibility without drastic change of current “single typed” output model.

```
public class SideOutputTransformation<T> extends StreamTransformation<T> {

  public SideOutputTransformation(StreamTransformation input, OutputTag<T> tag) {

   super("SideOutput", tag.getTypeInformation(), input.getParallelism());

   this.input = input;

  }

}
```

- OperatorChain Outputs needs to filter stream record with outputtag

Instead of assuming output stream record values are same type, it will need to check output type and stream record type, only output with same outputtag typeinfo.

Compatibility, Deprecation, and Migration Plan

- *What impact (if any) will there be on existing users? We would like to approach with two phase approach. Step 1 would be backward compatible manner. Other than RichCollector change can affects some existing userFunction binary compatibility, code level upgrade effort is very little other than change collector type class name from Collector -> RichCollector.*
- *Moving forward, code refactors would create bigger impact to framework backwards compatbility. We would like to revisit and discuss once we understand benefit and impact balance better.*

Test Plan

Phase 1 will follow same as adding feature to Flink API, adding unit tests and run local env mock tests