Apache Flink Streaming
DATA-DRIVEN DISTRIBUTED DATA STREAM PROCESSING

Seif Haridi, KTH/SICS
Paris Carbone, KTH
Gyula Fóra, SICS
1 year of Flink - code

April 2014

Stratosphere accepted as Apache Incubator Project
16 Apr 2014
We are happy to announce that Stratosphere has been accepted as a project for the Apache Incubator. The proposal has been accepted by the Incubator PMC members earlier this week. The Apache Incubator is the first step in the process of giving a project to the Apache Software Foundation. While under incubation, the project will move to the Apache infrastructure and adopt the community-driven development principles of the Apache Foundation. Projects can graduate from incubation to become top-level projects if they show activity, a healthy community dynamic, and releases.

April 2015

DataSet API (Java/Scala)
Flink core
Local Remote Yarn

DataSet (Java/Scala)
Flink core
Local Remote Yarn Tez Embedded

DataStream (Java/Scala)
Community growth

#unique contributors by git commits

Year:
- 2010: ago.-10
- 2011: feb.-11, sep.-11
- 2012: abr.-12, oct.-12
- 2013: may.-13, nov.-13
- 2014: jun.-14, dic.-14, jul.-15
Introduction

- The Flink Vision
- Flink Stack Overview
- Programming Model
- Execution Model
What is Apache Flink

Distributed Data Flow Processing System

- Focused on large-scale data analytics
- Unified real-time stream and batch processing
- Easy and powerful APIs in Java / Scala (+ Python)
- Robust and fast execution backend
“Provide a use-case complete solution that can unify batch and streaming data processing tasks under the same codebase and execution runtime”
WHAT ARE WE BUILDING

An engine that puts equal emphasis to streaming and batch

Real-time data streams
- Event logs
  - Kafka, RabbitMQ, ...
- Historic data
  - HDFS, JDBC, ...

Flink
THE FLINK STACK

- Python
- Gelly
- Table
- Flink ML
- Dataflow
- SAMOA

Hadoop M/R

- DataSet (Java/Scala)
- DataStream (Java/Scala)

Flink Optimizer

Flink Runtime

- Local
- Remote
- Yarn
- Tez
- Embedded
Stream processing

- **Data stream**: Infinite sequence of data arriving in a continuous fashion.
- **Stream processing**: Analyzing and acting on real-time streaming data, using continuous queries
3 Parts of a Streaming Infrastructure

- Server Logs
- Sensors
- Transaction logs

Gathering → Kafka → Broker → Analysis
### Streaming landscape

<table>
<thead>
<tr>
<th></th>
<th><strong>Apache Storm</strong></th>
<th><strong>Spark Streaming</strong></th>
<th><strong>Apache Samza</strong></th>
<th><strong>Apache Flink</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>![Storm Logo]</td>
<td>• True streaming over distributed dataflow</td>
<td>• Stream processing emulated on top of batch system (non-native)</td>
<td>• True streaming built on top of Apache Kafka, state is first class citizen</td>
<td>• True streaming over stateful distributed dataflow</td>
</tr>
<tr>
<td></td>
<td>• Low level API (Bolts, Spouts) + Trident</td>
<td>• Functional API (DStreams), restricted by batch runtime</td>
<td>• Slightly different stream notion, low level API</td>
<td>• Rich functional API exploiting streaming runtime; e.g. rich windowing semantics</td>
</tr>
</tbody>
</table>

**Apache Storm**
- True streaming over distributed dataflow
- Low level API (Bolts, Spouts) + Trident

**Spark Streaming**
- Stream processing emulated on top of batch system (non-native)
- Functional API (DStreams), restricted by batch runtime

**Apache Samza**
- True streaming built on top of Apache Kafka, state is first class citizen
- Slightly different stream notion, low level API

**Apache Flink**
- True streaming over stateful distributed dataflow
- Rich functional API exploiting streaming runtime; e.g. rich windowing semantics
Flink Streaming
What is Flink Streaming

- Native, low-latency stream processor
- Expressive functional API
- Flexible operator state, stream windows
- Exactly-once processing semantics
PROGRAMMING MODEL

- **Data Stream**
  - An abstract data type representing an unbounded, partitioned immutable sequence of events

- **Stream Operators**
  - Stream transformations that generate new output Data Streams from input ones
EXECUTION MODELS

1) BAtched/Stateless  (scheduled in Batches)

STATELESS SHORT-LIVED TASKS

(Hadoop, Spark)

DISTRIBUTED STREAMING OVER BATCHES

(Spark Streaming)

2) DataFlow/STATEFUL  (continuous/scheduled once)

long-lived task execution

state is kept inside tasks

(Storm, Samza, Naiad, Flink)
Native vs non-native streaming

Non-native streaming

```java
while (true) {
    // get next few records
    // issue batch computation
}
```

Native streaming

```java
while (true) {
    // process next record
}
```
WHY DATAFLOW

1) BATCHED/STATELESS (SCHEDULED IN BATCHES)
   - Trivial Fault Tolerance (lost batches can be recomputed)
   - High Throughput
   - High Latency (batching latency)
   - Limited Expressivity (stateless nature of tasks)

2) DATAFLOW/STATEFUL (CONTINUOUS/SCHEDULED ONCE)
   - Low Latency
   - True Streaming
   - Non trivial Fault Tolerance
     - (tasks should recover from consistent state)
API
OVERVIEW

- Stream Sources, Sinks
- Transformations
- Windowing Semantics
Overview of the API

- **Data stream sources**
  - File system
  - Message queue connectors
  - Arbitrary source functionality

- **Stream transformations**
  - Basic transformations: *Map, Reduce, Filter, Aggregations*…
  - Binary stream transformations: *CoMap, CoReduce*…
  - Windowing semantics: *Policy based flexible windowing (Time, Count, Delta)*…
  - Temporal binary stream operators: *Joins, Crosses*…
  - Native support for iterations

- **Data stream outputs**

- For the details please refer to the programming guide:
Basic Transformations

- map, filter, reduce, aggregations (eg. max, sum)
- reduce is incremental
  Stream(1, 2, 3, 4, ...).sum \Rightarrow Stream(1, 3, 6, 10,...)

Binary Transformations

- merge (union) , coMap, coReduce (two streams)
- join, cross (defined per window)
Binary stream transformations

- Apply shared transformations on streams of different types.
- Shared state between transformations
- CoMap, CoFlatMap, CoReduce…

```java
public interface CoMapFunction<IN1, IN2, OUT> {
    public OUT map1(IN1 value);
    public OUT map2(IN2 value);
}
```
case class Word(word: String, count: Long)

val input = env.socketTextStream(host, port);
val words = input.flatMap {ln => ln.split("\\W+")}.
map(w => Word(w,1))
val counts = words.groupBy("word").sum("count")
.print()

- In grouped streams, for each incoming tuple the selected field is transformed to the aggregated value
WINDOWING SEMANTICS

• Trigger and Eviction policies
  • `window(<eviction>, <trigger>)`
  • `window(<eviction>).every(<trigger>)`

• Built-in policies:
  – Time: `Time.of(length, TimeUnit/Custom timestamp)`
  – Count: `Count.of(windowSize)`
  – Delta: `Delta.of(treshold, Distance function, Start value)`

• Window transformations:
  – Reduce
  – mapWindow

• Custom trigger and eviction policies can also be trivially implemented
case class Word(word: String, count: Long)

val input = env.socketTextStream(host, port);
val words = input flatMap {
    line => line.split("\W+").map(Word(_,1))
        .window(Count.of(20)).every(Count.of(10))
}.
val counts = words.groupBy("word").sum("count")
Flexible windows

```scala
case class Count(symbol: String, count: Int)
val defaultPrice = StockPrice("", 1000)

// Use delta policy to create price change warnings
val priceWarnings = stockStream.groupBy("symbol")
  .window(Delta.of(0.05, priceChange, defaultPrice))
  .mapWindow(sendWarning _)

// Count the number of warnings every half a minute
val warningsPerStock = priceWarnings.map(Count(_, 1))
  .groupBy("symbol")
  .window(Time.of(30, SECONDS))
  .sum("count")
```

Performance

- Performance optimizations
  - Effective serialization due to strongly typed topologies
  - Operator chaining (thread sharing/no serialization)
  - Different automatic query optimizations

- Competitive performance
  - ~ 1.5m events / sec / core
  - As a comparison Storm promises ~ 1m tuples / sec / node
OPTIMISATIONS

- **Window Pre-aggregates**
  - **Implemented**: sliding (panes), tumbling/jumping window pre-aggregates
  - **Pending**: Operator Sharing, Optimistic pre-aggregations

- **Operator Chaining**
  - Collapsing multiple operators into a single execution thread

- **Operator Reordering**
Fault Tolerance
Overview

- Fault tolerance in other systems
  - Message tracking/acks (Storm)
  - RDD re-computation (Spark)

- Fault tolerance in Apache Flink
  - Based on consistent global snapshots
  - Algorithm inspired by Chandy-Lamport
  - Low runtime overhead, stateful exactly-once semantics
PROCESSING GUARANTEES

- Explicit state representation
- Periodic minimal state snapshotting
- Partial execution graph recovery
- Towards exactly-once processing semantics
Pushes checkpoint barriers through the data flow

After barrier = Not in snapshot
Before barrier = part of the snapshot
(backup till next snapshot)

Asynchronous Barrier Snapshotting for globally consistent checkpoints
State management

- State declared in the operators is managed and checkpointed by Flink
- Pluggable backends for storing persistent snapshots
  - Currently: JobManager, FileSystem (HDFS, Tachyon)
- State partitioning and flexible scaling in the future
A USE CASE

- Get stock price updates from multiple sources
- Generate online statistics on the stock data
- Detect stock price fluctuations
- Detect twitter trends on stock mentions
- Correlate trends and fluctuations
USE CASE STEPS

- Stock DataStream creation
- Rolling window analytics
- Detecting stock price fluctuations
- Detecting trends from twitter streams
- Correlating stock fluctuations with trends

- Detailed explanation and source code on our blog
case class StockPrice(symbol: String, price: Double)
val env = StreamExecutionEnvironment.getExecutionEnvironment

(1) val socketStockStream = env.socketTextStream("localhost", 9999)
    .map(x => { val split = x.split(",")
                  StockPrice(split(0), split(1).toDouble) })

(2) val SPX_Stream = env.addSource(generateStock("SPX")(10) _)
val FTSE_Stream = env.addSource(generateStock("FTSE")(20) _)

(3) val stockStream = socketStockStream.merge(SPX_Stream, FTSE_STREAM)
ROLLING ANALYTICS

```
val windowedStream = stockStream
  .window(Time.of(10, SECONDS)).every(Time.of(5, SECONDS))

(2) val lowest = windowedStream.minBy("price")
(3) val maxByStock = windowedStream.groupBy("symbol").maxBy("price")
(4) val rollingMean = windowedStream.groupBy("symbol").mapWindow(mean _)
```
case class Count(symbol : String, count : Int)

val priceWarnings = stockStream.groupBy("symbol")
  .window(Delta.of(0.05, priceChange, defaultPrice))
  .mapWindow(sendWarning _)

val warningsPerStock = priceWarnings.map(Count(_, 1)).groupBy("symbol")
  .window(Time.of(30, SECONDS))
  .sum("count")
CREATING TREND STREAMS

"hdp is on the rise!"
"I wish I bought more
YHOO and HDP stocks"

1. val tweetStream = env.addSource(generateTweets (_))
2. val mentionedSymbols = tweetStream.flatMap(tweet => tweet.split(" "))
   .map(_.toUpperCase())
   .filter(symbols.contains(_))
3. val tweetsPerStock = mentionedSymbols.map(Count(_, 1)).groupBy("symbol")
4. .window(Time.of(30, SECONDS))
   .sum("count")

Count(HDP, 2)
Count(YHOO, 1)
JOINING STREAMS

```
val tweetsAndWarning = warningsPerStock.join(tweetsPerStock)
  .onWindow(30, SECONDS)
  .where("symbol")
  .equalTo("symbol")
  { (c1, c2) => (c1.count, c2.count) }

val rollingCorrelation = tweetsAndWarning
  .window(Time.of(30, SECONDS))
  .mapWindow(computeCorrelation _)
```
Background slides
ONGOING WORK

- Machine Learning Pipelines
- Streaming Graphs
Streaming roadmap for 2015

- Improved state management
  - New backends for state snapshotting
  - Support for state partitioning and incremental snapshots
  - Master Failover

- Improved job monitoring

- Integration with other Apache projects
  - SAMOA (PR ready), Zeppelin (PR ready), Ignite

- Streaming machine learning and other new libraries
Combining scikit-learn and MOA for a first-ever distributed, multi-paradigm ML pipelines library
STREAMING GRAPHS

- Streaming newly generated graph data
- Keeping only the *fresh* state in memory
- Continuously computing graph approximations
INTEGRATIONS

- Apache Samoa (incubating)
- Flink Deployments with Karamel
- Table API
- Google DataFlow API (done)
- Apache Storm Compatibility Layer
PROJECT WEBSITE: https://flink.apache.org/

PROJECT REPO: https://github.com/apache/flink


USER MAILLIST: user@flink.apache.org