Towards Declarative Stream Processing using Apache Flink

Tilmann Rabl
Berlin Big Data Center

www.dima.tu-berlin.de | bbdc.berlin | rabl@tu-berlin.de
Agenda
Agenda

Apache Flink Primer
• Architecture
• Execution Engine
• Some key features
• Some demo (!)

Stream Processing with Apache Flink
• Flexible Windows/Stream Discretization
• Exactly-once Processing & Fault Tolerance

Apache Flink Use Cases
• How companies use Flink (if we have time)

With slides from Data Artisans, Volker Markl, Asterios Katsifodimos, Juan Soto
Flink Primer
What is Apache Flink?

Apache Flink is an open source platform for scalable batch and stream data processing.

- The core of Flink is a distributed streaming dataflow engine.
  - Executing dataflows in parallel on clusters
  - Providing a reliable foundation for various workloads
- `DataSet` and `DataStream` programming abstractions are the foundation for user programs and higher layers

http://flink.apache.org
What can I do with it?

An engine that can **natively** support all these workloads.

- **Batch processing**
- **Stream processing**
- **Machine Learning at scale**
- **Graph Analysis**
Flink Timeline

2008
Initial vision for a big data analytics platform

2009
DFG Proposal for Stratosphere I

Consortium
StratoSphere Above the Clouds

2010
Grant Award
Start of Stratosphere I

DFG Proposal for Stratosphere II

2012
Grant Award
Start of Stratosphere II

Spinning Fast Iterative Dataflows paper published

2014
The VLDB Journal
Stratosphere System paper published

APACHE Flink
Incubator Project

2015
Flink Top Level Project

APACHE Flink
Top Level Project

dataArtisans
Founded

2015
1st Flink Forward Conference

FlinkForward
2nd Flink Forward Conference
September 2016
flink-forward.org

2016
Flink Community Groups
Across Europe

Berlin 758
Paris 500
Madrid 384
Stockholm 313
Brussels 279
London 190

21 Meetups Worldwide
186 Contributors
6326 Members
20 Cities
12 Countries
Stratosphere: General Purpose Programming + Database Execution

Draws on Database Technology:
- Relational Algebra
- Declarativity
- Query Optimization
- Robust Out-of-core

Adds:
- Iterations
- Advanced Dataflows
- General APIs
- Native Streaming

Draws on MapReduce Technology:
- Scalability
- User-defined Functions
- Complex Data Types
- Schema on Read
Flink in the Analytics Ecosystem

Applications & Languages
- Hive
- Cascading
- Giraph
- Mahout
- Pig
- Crunch

Data processing engines
- MapReduce
- Flink
- Spark
- Storm
- Tez

App and resource management
- Yarn
- Mesos

Storage, streams
- HDFS
- HBase
- Kafka
- ...
Where in my cluster does Flink fit?

- Gather and backup streams
- Offer streams for consumption
- Provide stream recovery
- Analyze and correlate streams
- Create derived streams and state
- Provide these to upstream systems
Flink Execution Model

- Flink program = DAG* of operators and intermediate streams
- Operator = computation + state
- Intermediate streams = logical stream of records
Architecture

• Hybrid MapReduce and MPP database runtime

• Pipelined/Streaming engine
  – Complete DAG deployed
Technology inside Flink

```
case class Path (from: Long, to: Long)
val tc = edges.iterate(10) {
  paths: DataSet[Path] =>
  val next = paths
  .join(edges)
  .where("to")
  .equalTo("from")
  (path, edge) =>
  Path(path.from, edge.to)
  .union(paths)
  .distinct()
  next
}
```

Program
Rich set of operators

Map, Reduce, Join, CoGroup, Union, Iterate, Delta Iterate, Filter, FlatMap, GroupReduce, Project, Aggregate, Distinct, Vertex-Update, Accumulators, ...
Effect of optimization

- Hash vs. Sort
- Partition vs. Broadcast
- Caching
- Reusing partition/sort

Execution Plan A
Run on a sample on the laptop

Execution Plan B
Run on large files on the cluster

Execution Plan C
Run a month later after the data evolved
Scale Out

Source → Map → Reduce → Iterate → Join → Reduce → Sink

Source → Map

Source → Map → Reduce → Iterate → Join → Reduce → Sink

Source → Map
Flink Optimizer
Transitive Closure

• What you write is **not** what is executed
• No need to hardcode execution strategies

Flink Optimizer decides:
- Pipelines and dam/barrier placement
- Sort- vs hash-based execution
- Data exchange (partition vs. broadcast)
- Data partitioning steps
- In-memory caching

---

![Diagram](image-url)
Built-in vs. driver-based looping

Loop outside the system, in driver program

Iterative program looks like many independent jobs

Dataflows with feedback edges

System is iteration-aware, can optimize the job
Managed Memory

- Language APIs automatically converts objects to tuples
  - Tuples mapped to pages/buffers of bytes
  - Operators can work on pages/buffers
- Full control over memory, out-of-core enabled
- Operators (e.g., Hybrid Hash Join) address individual fields (not deserialize object): robust

```
public class Person {
  int id;
  String name;
}
```

<table>
<thead>
<tr>
<th>Tuple3&lt;Integer, Double, Person&gt;</th>
<th>F0: int</th>
<th>F1: double</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntSerialized</td>
<td></td>
<td>IntSerialized</td>
<td></td>
</tr>
<tr>
<td>DoubleSerialized</td>
<td></td>
<td></td>
<td>String</td>
</tr>
<tr>
<td>StringSerialized</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory runs out
Sneak peak: Two of Flink’s APIs

```scala
case class Word (word: String, frequency: Int)

val lines: DataSet[String] = env.readTextFile(...)  
lines.flatMap {line => line.split(" ")  
  .map(word => Word(word,1))  
  .groupBy("word").sum("frequency")  
  .print()

val lines: DataStream[String] = env.fromSocketStream(...)  
lines.flatMap {line => line.split(" ")  
  .map(word => Word(word,1))  
  .keyBy("word")  
  .window(Time.of(5,SECONDS)).every(Time.of(1,SECONDS))  
  .sum("frequency")  
  .print()
```

**DataSet API (batch):**

**DataStream API (streaming):**
DEMO – BATCH

„Inspired“ by
http://dataartisans.github.io/flink-training/index.html
Stream Processing with Flink
Ingredients of a Streaming System

• Streaming Execution Engine
• Windowing (a.k.a Discretization)
• Fault Tolerance
• High Level Programming API (or language)
Batch is a Special Case of Streaming

Lower-overhead fault-tolerance via replaying intermediate results

Blocking operators (e.g., hybrid hash join) are embedded in streaming topology
Mini-Batching vs Native Streaming

Discretized Streams (D-Streams)

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

Native streaming

while (true) {
    // process next record
}
Problems of Mini-Batch

• Latency
  – Each mini-batch schedules a new job, loads user libraries, establishes DB connections, etc

• Programming model
  – Does not separate business logic from recovery – changing the mini-batch size changes query results

• Power
  – Keeping and updating state across mini-batches only possible by immutable computations
Stream Discretization

- Data is unbounded
  - Interested in a (recent) part of it e.g. last 10 days
- Most common windows around: time, and count
  - Mostly in sliding, fixed, and tumbling forms
- Need for data-driven window definitions
  - e.g., user sessions (periods of user activity followed by inactivity), price changes, etc.

The world beyond batch: Streaming 101, Tyler Akidau
Great read!
Flink’s Discretization

• Allows very flexible windowing

• Borrows ideas (and extends) from IBM’s SPL
  – SLIDE = Trigger = When to emit a window
  – RANGE = Eviction = What the window contains

• Allows for lots of optimization
  – Not part of this talk
Flink’s Windowing

- Windows can be any combination of (multiple) triggers & evictions
  - Arbitrary tumbling, sliding, session, etc. windows can be constructed.

- Common triggers/evictions part of the API
  - Time (processing vs. event time), Count

- Even more flexibility: define your own UDF trigger/eviction

- Examples:
  ```java
dataStream.windowAll(TumblingEventTimeWindows.of(Time.seconds(5)));
dataStream.keyBy(0).window(TumblingEventTimeWindows.of(Time.seconds(5)));``
Example Analysis: Windowed Aggregation

\[\text{StockStream} \quad \rightarrow \quad \text{10 sec window every 5 secs} \quad \rightarrow \quad \text{MinBy Price} \quad \rightarrow \quad \text{MaxBy Price} \quad \rightarrow \quad \text{Mean Price} \]

\(1\) \quad \text{val windowedStream} = \text{stockStream}\text{.window(Time.of(10, SECONDS)).every(Time.of(5, SECONDS))}

\(2\) \quad \text{val lowest} = \text{windowedStream}\text{.minBy("price")}

\(3\) \quad \text{val maxByStock} = \text{windowedStream}\text{.groupBy("symbol")}\text{.maxBy("price")}

\(4\) \quad \text{val rollingMean} = \text{windowedStream}\text{.groupBy("symbol")}\text{.mapWindow(mean \_)}

- StockPrice(SPX, 2113.9)
- StockPrice(FTSE, 6931.7)
- StockPrice(HDP, 23.8)
- StockPrice(HDP, 26.6)
Current Benchmark Results

Performed by Yahoo! Engineering, Dec 16, 2015

[...]Storm 0.10.0, 0.11.0-SNAPSHOT and Flink 0.10.1 show sub-second latencies at relatively high throughputs[...]. Spark streaming 1.5.1 supports high throughputs, but at a relatively higher latency.

Flink achieves highest throughput with competitive low latency!
Flink Community

# unique contributor ids by Git commits
Conclusion

The case for Flink as a stream processor

– Proper streaming engine foundation
– Flexible Windowing
– Fault Tolerance with exactly once guarantees
– Integration with batch
– Large (and growing!) community
Thank You

Contact:
Tilmann Rabl
rabl@tu-berlin.de