Architecture of Flink's Streaming Runtime

Robert Metzger
@rmetzger_
rmetzger@apache.org
What is stream processing

- Real-world data is unbounded and is pushed to systems
- Right now: people are using the batch paradigm for stream analysis (there was no good stream processor available)
- New systems (Flink, Kafka) embrace streaming nature of data
Flink is a **stream processor with many faces**
Flink's streaming runtime
Requirements for a stream processor

- Low latency
  - Fast results (milliseconds)

- High throughput
  - handle large data amounts (millions of events per second)

- Exactly-once guarantees
  - Correct results, also in failure cases

- Programmability
  - Intuitive APIs
Pipelining

Basic building block to “keep the data moving”

• Low latency
• Operators push data forward
• Data shipping as buffers, not tuple-wise
• Natural handling of back-pressure

Complete pipeline online concurrently

source tokenizer window count
Fault Tolerance in streaming

- **at least once**: ensure all operators see all events
  - Storm: Replay stream in failure case

- **Exactly once**: Ensure that operators do not perform duplicate updates to their state
  - Flink: Distributed Snapshots
  - Spark: Micro-batches on batch runtime
Flink’s Distributed Snapshots

- Lightweight approach of storing the state of all operators without pausing the execution
  → high throughput, low latency
- Implemented using barriers flowing through the topology

---

**Data Stream**

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

**Kafka Consumer**

- Offset = 162

**Element Counter**

- Value = 152

**Operator State**
Starting Checkpoint

Master

Checkpoint data

<table>
<thead>
<tr>
<th>Source 1:</th>
<th>State 1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source 2:</td>
<td>State 2:</td>
</tr>
<tr>
<td>Source 3:</td>
<td>Sink 1: (pending)</td>
</tr>
<tr>
<td>Source 4:</td>
<td>Sink 2: (pending)</td>
</tr>
</tbody>
</table>

State Backend

Start checkpoint message

Current position: 6791

Emit stream barriers

Current position: 7252

Current position: 5589

Current position: 6843
Master

Checkpoint data

| Source 1: 6791 | State 1: |
| Source 2: 7252 | State 2: |
| Source 3: 5589 | Sink 1: (pending) |
| Source 4: 6843 | Sink 2: (pending) |

State Backend

Checkpoint in Progress
Checkpoint in Progress
Sink acknowledges checkpoint after receiving all barriers.
Best of all worlds for streaming

- Low latency
  - Thanks to pipelined engine

- Exactly-once guarantees
  - Distributed Snapshots

- High throughput
  - Controllable checkpointing overhead

- Separates app logic from recovery
  - Checkpointing interval is just a config parameter
Throughput of distributed grep

aggregate throughput of **175 million elements** per second

aggregate throughput of **9 million elements** per second

- Flink achieves 20x higher throughput
- Flink throughput almost the same with and without exactly-once
Aggregate throughput for stream record grouping

<table>
<thead>
<tr>
<th></th>
<th>Aggregate throughput of 83 million elements per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flink, no fault tolerance</td>
<td>8,6 million elements/s</td>
</tr>
<tr>
<td>Flink, exactly once</td>
<td>309k elements/s</td>
</tr>
<tr>
<td>Storm, no fault tolerance</td>
<td>30 machines, 120 cores</td>
</tr>
<tr>
<td>Storm, at least once</td>
<td>Network transfer</td>
</tr>
</tbody>
</table>

→ Flink achieves 260x higher throughput with fault tolerance
Latency in stream record grouping

- Measure time for a record to travel from source to sink

<table>
<thead>
<tr>
<th>Throughput / Latency measure</th>
<th>Flink, no fault tolerance</th>
<th>Flink, exactly once</th>
<th>Storm, at least once</th>
</tr>
</thead>
<tbody>
<tr>
<td>Median latency</td>
<td>25 ms</td>
<td>1 ms</td>
<td></td>
</tr>
<tr>
<td>99th percentile latency</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flink, no fault tolerance</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flink, exactly once</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Storm, at least once</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data Generator
Receiver: Throughput / Latency measure
Latency-throughput tradeoff in Flink using different values of buffer timeout

Latency (median and 0.99-percentile) in milliseconds

Average aggregate throughput (Millions of events/sec)

Flink buffer timeout (milliseconds)
Exactly-Once with YARN Chaos Monkey

- Validate exactly-once guarantees with state-machine
“Faces” of Flink
Faces of a stream processor

Stream processing

Batch processing

Machine Learning at scale

Graph Analysis

Streaming dataflow runtime
The Flink Stack

Specialized Abstractions / APIs
Core APIs
Flink Core Runtime
Deployment

Streaming dataflow runtime

Flink Stack Diagram
APIs for stream and batch

```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()

DataSet API (batch):

DataSet API (streaming):

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

- **Experimental Python API also available**

- **Data Set (Java/Scala)**
- **DataStream (Java/Scala)**

- **Batch Optimizer**
- **Graph Builder**

**API independent Dataflow Graph representation**

**Streaming dataflow runtime**
Batch is a special case of streaming

- Batch: run a bounded stream (data set) on a stream processor
- Form a global window over the entire data set for join or grouping operations
Batch-specific optimizations

- **Managed memory** on- and off-heap
  - Operators (join, sort, …) with out-of-core support
  - Optimized serialization stack for user-types

- **Cost-based Optimizer**
  - Job execution depends on data size
The Flink Stack

Specialized Abstractions / APIs

Core APIs

Flink Core Runtime

Deployment
FlinkML: Machine Learning

- API for ML pipelines inspired by scikit-learn
- Collection of packaged algorithms
  - SVM, Multiple Linear Regression, Optimization, ALS, ...

```scala
val trainingData: DataSet[LabeledVector] = ...
val testingData: DataSet[Vector] = ...

val scaler = StandardScaler()
val polyFeatures = PolynomialFeatures().setDegree(3)
val mlr = MultipleLinearRegression()

val pipeline = scaler.chainTransformer(polyFeatures).chainPredictor(mlr)
pipeline.fit(trainingData)
val predictions: DataSet[LabeledVector] = pipeline.predict(testingData)
```
Gelly: Graph Processing

- Graph API and library
- Packaged algorithms
  - PageRank, SSSP, Label Propagation, Community Detection, Connected Components

```java
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

Graph<Long, Long, NullValue> graph = ...

DataSet<Vertex<Long, Long>> verticesWithCommunity = graph.run(
    new LabelPropagation<Long>(30)).getVertices();

verticesWithCommunity.print();

env.execute();
```
Flink Stack += Gelly, ML

DataSet (Java/Scala)  DataStream

Streaming dataflow runtime
Integration with other systems

- Use Hadoop Input/Output Formats
- Mapper / Reducer implementations
- Hadoop’s FileSystem implementations

- Run applications implemented against Google’s Data Flow API on premise with Flink
  - Run Cascading jobs on Flink, with almost no code change
  - Benefit from Flink’s vastly better performance than MapReduce

- Interactive, web-based data exploration
- Machine learning on data streams
  - Compatibility layer for running Storm code
  - FlinkTopologyBuilder: one line replacement for existing jobs
  - Wrappers for Storm Spouts and Bolts
  - Coming soon: Exactly-once with Storm
Deployment options

- Start Flink in your IDE / on your machine
- Local debugging / development using the same code as on the cluster
- “bare metal” standalone installation of Flink on a cluster
- Flink on Hadoop YARN (Hadoop 2.2.0+)
- Restarts failed containers
- Support for Kerberos-secured YARN/HDFS setups
The full stack

- Hadoop M/R
- Table
- Gelly
- ML
- Dataflow
- MRQL
- Cascading
- Zeppelin
- Table
- SAMOA
- Dataflow (WiP)
- Storm (WiP)

**DataFlow Diagram:**

- ** DataSet (Java/Scala)**
- **DataStream**
- ** Streaming dataflow runtime**

**Runtime Options:**

- Local
- Cluster
- Yarn
- Tez
- Embedded
Closing
Flink is a software stack of

- Streaming runtime
  - low latency
  - high throughput
  - fault tolerant, exactly-once data processing

- Rich APIs for batch and stream processing
  - library ecosystem
  - integration with many systems

- A great community of devs and users

- Used in production
What is currently happening?

- Features in progress:
  - Master High Availability
  - Vastly improved monitoring GUI
  - Watermarks / Event time processing / Windowing rework
  - Graduate Streaming API out of Beta

- 0.10.0-milestone-1 is currently voted
How do I get started?

**Mailing Lists:** (news | user | dev)@flink.apache.org

**Twitter:** @ApacheFlink

**Blogs:** flink.apache.org/blog, data-artisans.com/blog/

**IRC channel:** irc.freenode.net#flink

---

**Start Flink on YARN in 4 commands:**

```bash
# get the hadoop2 package from the Flink download page at
# http://flink.apache.org/downloads.html
wget <download url>
tar xzvf flink-0.9.1-bin-hadoop2.tgz
cd flink-0.9.1/
./bin.flink run -m yarn-cluster -yn 4 ./examples/flink-java-examples-0.9.1-WordCount.jar
```
Flink Forward: 2 days conference with free training in Berlin, Germany

- Schedule: [http://flink-forward.org/?post_type=day](http://flink-forward.org/?post_type=day)
Appendix
Managed (off-heap) memory and out-of-core support

Memory runs out
Cost-based Optimizer

Best plan depends on relative sizes of input files
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

Type extraction stack
Optimizer
Pre-flight (Client)

Dataflow Graph

Cluster: YARN, Standalone

JobManager

Task scheduling
Dataflow metadata

deploy operators
track intermediate results

TaskManagers
Iterative processing in Flink

Flink offers built-in iterations and delta iterations to execute ML and graph algorithms efficiently.
Example: Matrix Factorization

Factorizing a matrix with 28 billion ratings for recommendations

Batch aggregation

"Blocked" result partition

TaskManager 1

TaskManager 2

M1

M2

RP1

RP2

R1

R2

JobManager

ExecutionGraph
Streaming window aggregation

"Pipelined" result partition

TaskManager 1

TaskManager 2

M1 -> RP1 -> R1

M2 -> RP2

R2

JobManager

ExecutionGraph

1 -> 2 -> 3a -> 3b

4a -> 4b -> 5a

5b