Stream Processing with Apache Flink

Robert Metzger
@rmetzger_
rmetzger@apache.org

QCon London, March 7, 2016
Talk overview

- My take on the stream processing space, and how it changes the way we think about data
- Discussion of unique building blocks of Flink
- Benchmarking Flink, by extending a benchmark from Yahoo!
Apache Flink

- Apache Flink is an open source stream processing framework
  - Low latency
  - High throughput
  - Stateful
  - Distributed

- Developed at the Apache Software Foundation, 1.0.0 release available soon, used in production
Entering the streaming era
Streaming is the biggest change in data infrastructure since Hadoop
1. Radically simplified infrastructure
2. Do more with your data, faster
3. Can completely subsume batch
Traditional data processing

- Log analysis example using a batch processor

- Periodic (custom) or continuous ingestion (Flume) into HDFS
- Periodic log analysis job
- Batch job(s) for log analysis
- Serving layer
- Job scheduler (Oozie)
Traditional data processing

- **Latency** from log event to serving layer usually in the **range of hours**

![Diagram showing traditional data processing flow]

- Web server -> Logs
- Periodic (custom) or continuous ingestion (Flume) into HDFS
- HDFS / S3
- Periodic log analysis job
- Batch job(s) for log analysis
- Serving layer
- Job scheduler (Oozie) every 2 hrs

Latency from log event to serving layer usually in the range of hours.
Data processing without stream processor

- This architecture is a hand-crafted micro-batch model

<table>
<thead>
<tr>
<th>Approach</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manually triggered periodic batch job</td>
<td>hours</td>
</tr>
<tr>
<td>Batch processor with micro-batches</td>
<td>minutes</td>
</tr>
<tr>
<td>Stream processor</td>
<td>seconds</td>
</tr>
<tr>
<td></td>
<td>milliseconds</td>
</tr>
</tbody>
</table>
Downsides of stream processing with a batch engine

- Very high latency (hours)
- Complex architecture required:
  - Periodic job scheduler (e.g. Oozie)
  - Data loading into HDFS (e.g. Flume)
  - Batch processor
  - (When using the “lambda architecture”: a stream processor)

→ All these components need to be implemented and maintained

- Backpressure: How does the pipeline handle load spikes?
Log event analysis using a stream processor

- Stream processors allow to analyze events with **sub-second latency**.

**Options:**
- Apache Kafka
- Amazon Kinesis
- MapR Streams
- Google Cloud Pub/Sub

**Options:**
- Apache Flink
- Apache Beam
- Apache Samza
Real-world data is produced in a continuous fashion.

New systems like Flink and Kafka embrace streaming nature of data.
What do we need for replacing the “batch stack”?

Options:
- Apache Kafka
- Amazon Kinesis
- MapR Streams
- Google Cloud Pub/Sub

Forward events immediately to pub/sub bus

Stream Processor
- Windowing / Out of order events
- State handling
- Fault tolerance and correctness
- Low latency
- High throughput

Process events in real time & update serving layer

Serving layer

High throughput publish/subscribe
Apache Flink stack

DataStream (Java / Scala)

DataSet (Java/Scala)

Streaming dataflow runtime

YARN

Cluster

Local
Needed for the use case

DataStream (Java / Scala)

DataSet (Java/Scala)

Streaming dataflow runtime

YARN  Cluster  Local
Windowing / Out of order events

- Windowing / Out of order events
- State handling
- Fault tolerance and correctness
- Low latency High throughput
Building windows from a stream

- "Number of visitors in the last 5 minutes per country"

```java
// create stream from Kafka source
DataStream<LogEvent> stream = env.addSource(new KafkaConsumer());
// group by country
DataStream<LogEvent> keyedStream = stream.keyBy("country");
// window of size 5 minutes
keyedStream.timeWindow(Time.minutes(5))
// do operations per window
.apply(new CountPerWindowFunction());
```
Building windows: Execution

```java
// window of size 5 minutes
dStream.timeWindow(Time.minutes(5));
```

**Job plan**

- Kafka Source
- group by country
- Window Operator

**Parallel execution on the cluster**

- S -> W
- S -> W
- S -> W

**Time**
Window types in Flink

- Tumbling windows
- Sliding windows
- Custom windows with window assigners, triggers and evictors

Time-based windows

Windows are created based on the real world time when the event occurred

{ "accessTime": "1457002134", "userId": "1337", "userLocation": "UK" }
A look at the reality of time

- Events arrive out of order in the system
- Use-case specific low watermarks for time tracking
Time characteristics in Apache Flink

- **Event Time**
  - Users have to specify an event-time extractor + watermark emitter
  - Results are deterministic, but with latency

- **Processing Time**
  - System time is used when evaluating windows
  - low latency

- **Ingestion Time**
  - Flink assigns current system time at the sources

- **Pluggable, without window code changes**
State handling

- Windowing / Out of order events
- State handling
- Fault tolerance and correctness
- Low latency High throughput
State in streaming

- Where do we store the elements from our windows?

- In stateless systems, an external state store (e.g. Redis) is needed.
Managed state in Flink

- Flink automatically backups and restores state
- State can be larger than the available memory
- State backends: (embedded) RocksDB, Heap memory
Managing the state

- How can we operate such a pipeline 24x7?
- Losing state (by stopping the system) would require a replay of past events
- We need a way to store the state somewhere!
Savepoints: Versioning state

- Savepoint: Create an addressable copy of a job’s current state.
- Restart a job from any savepoint.

Fault tolerance and correctness

- Windowing / Out of order events
- State handling
- Fault tolerance and correctness
- Low latency
  High throughput
Fault tolerance in streaming

- How do we ensure the results (number of visitors) are always correct?
- Failures should not lead to data loss or incorrect results
Fault tolerance in streaming

- **at least once**: ensure all operators see all events
  - Storm: Replay stream in failure case (acking of individual records)

- **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
- Implemented using barriers flowing through the topology

Further reading: http://blog.acolyer.org/2015/08/19/asynchronous-distributed-snapshots-for-distributed-dataflows/
Wrap-up: Log processing example

- How to do something with the data? **Windowing**
- How does the system handle large windows? **Managed state**
- How do operate such a system 24x7? **Safepoints**
- How to ensure correct results across failures? **Checkpoints, Master HA**
Performance: Low Latency & High Throughput

- Windowing / Out of order events
- State handling
- Fault tolerance and correctness
- Low latency High throughput
Performance: Introduction

- Performance always depends on your own use cases, so test it yourself!
- We based our experiments on a recent benchmark published by Yahoo!
- They benchmarked Storm, Spark Streaming and Flink with a production use-case (counting ad impressions)

Full Yahoo! article: https://yahooseng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at
Yahoo! Benchmark

- Count ad impressions grouped by campaign
- Compute aggregates over a 10 second window
- Emit current value of window aggregates to Redis every second for query

Full Yahoo! article: https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at
Yahoo’s Results

“Storm […] and Flink […] show sub-second latencies at relatively high throughputs with Storm having the lowest 99th percentile latency. Spark streaming 1.5.1 supports high throughputs, but at a relatively higher latency.”

(Quote from the blog post’s executive summary)

Full Yahoo! article: https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at
Extending the benchmark

- Benchmark stops at Storm’s throughput limits. Where is Flink’s limit?
- How will Flink’s own window implementation perform compared to Yahoo’s “state in redis windowing” approach?

Full Yahoo! article: https://yahoocore.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at
Windowing with state in Redis

KafkaConsumer
map()
filter()
group
windowing & caching code
realtime queries

kafka
Redis
Rewrite to use Flink’s own window

KafkaConsumer
map()
filter()
group
Flink event
time
windows
realtime queries
Results after rewrite

Throughput: msgs/sec

- Storm: 400k msgs/sec
- Flink: 3,000,000 msgs/sec
Can we even go further?

Network link to Kafka cluster is bottleneck! (1GigE)

KafkaConsumer
  map()
  filter()

Flink event
time
windows

Solution: Move data generator into job (10 GigE)
Results without network bottleneck

10 GigE end-to-end
Throughput: msgs/sec

Storm
- 400k msgs/sec

Flink
- 3m msgs/sec

Flink (10 GigE)
- 15m msgs/sec
Benchmark summary

- Flink achieves **throughput of 15 million messages/second** on 10 machines
- **35x higher throughput** compared to Storm (80x compared to Yahoo’s runs)
- Flink ran with **exactly once** guarantees, Storm with **at least once**.
Closing
Other notable features

- **Expressive DataStream API** (similar to high level APIs known from the batch world)
- Flink is a full-fledged **batch processor** with an optimizer, managed memory, memory-aware algorithms, build-in iterations
- Many **libraries**: Complex Event Processing (CEP), Graph Processing, Machine Learning
- **Integration** with YARN, HBase, ElasticSearch, Kafka, MapReduce, …
Questions?

- Ask now!
- eMail: rmetzger@apache.org
- Twitter: @rmetzger_

- Follow: @ApacheFlink
- Read: flink.apache.org/blog, data-artisans.com/blog/
- Mailinglists: (news | user | dev)@flink.apache.org
Apache Flink stack

CEP | Table | Storm API | Apache Beam | SAMOA | Hadoop M/R | Table | Gelly | ML | Apache Beam | Cascading | Zeppelin

DataStream (Java / Scala) | DataSet (Java/Scala)

Streaming dataflow runtime

YARN | Cluster | Local
Appendix
Roadmap 2016

- SQL / StreamSQL
- CEP Library
- Managed Operator State
- Dynamic Scaling
- Miscellaneous
Miscellaneous

- Support for Apache Mesos
- Security
  - Over-the-wire encryption of RPC (akka) and data transfers (netty)
- More connectors
  - Apache Cassandra
  - Amazon Kinesis
- Enhance metrics
  - Throughput / Latencies
  - Backpressure monitoring
  - Spilling / Out of Core
Fault Tolerance and correctness

- How can we ensure the state is always in sync with the events?
Naïve state checkpointing approach

- Process some records:

- Stop everything, store state:

- Continue processing …

<table>
<thead>
<tr>
<th>Operator</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
</tr>
</tbody>
</table>
Distributed Snapshots

Initial state

Start processing

Trigger checkpoint

<table>
<thead>
<tr>
<th>Operator</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
</tr>
</tbody>
</table>
Distributed Snapshots

- Valid snapshot without stopping the topology
- Multiple checkpoints can be in-flight
Analysis of naïve approach

- Introduces latency
- Reduces throughput

  - Can we create a **correct snapshot while keeping the job running**?
  - Yes! By creating a distributed snapshot
Handling Backpressure

Backpressure might occur when:
- Operators create checkpoints
- Windows are evaluated
- Operators depend on external resources
- JVMs do Garbage Collection

Slow down upstream operators

Operator not able to process incoming data immediately
Data sources slow down pulling data from their underlying system (Kafka or similar queues)
How do latency and throughput affect each other?

- High throughput by batching events in network buffers
- Filling the buffers introduces latency
- Configurable buffer timeout

Send buffer when full or timeout

30 Machines, one repartition step
Aggregate throughput for stream record grouping

→ Flink achieves 260x higher throughput with fault tolerance
Performance: Summary

Continuous streaming + Latency-bound buffering + Distributed Snapshots = High Throughput & Low Latency

*With configurable throughput/latency tradeoff*

---

**Latency-throughput tradeoff in Flink using different values of buffer timeout**

- **X-axis:** Flink buffer timeout (milliseconds)
- **Y-axis:** Latency (median and 0.99-percentile in milliseconds)
- **Graph:** Shows the tradeoff between latency and throughput for different buffer timeouts in Flink. The tradeoff curve indicates how increasing buffer timeouts affects latency and throughput.

---

**Aggregate throughput for stream record grouping**

- **X-axis:** Average throughput (millions of events per second)
- **Y-axis:** Stream record grouping methods
- **Legend:**
  - Storm, at least once: 310,000 events per second
  - Flink, exactly once
  - Flink, at most once

The graph compares the aggregate throughput for different record grouping methods in Flink, showing significant differences in performance for each method.
The building blocks: Summary

Windowing / Out of order events
- Tumbling / sliding windows
- Event time / processing time
- Low watermarks for out of order events

Fault tolerance and correctness
- Exactly-once semantics for managed operator state
- Lightweight, asynchronous distributed snapshotting algorithm

State handling
- Managed operator state for backup/recovery
- Large state with RocksDB
- Savepoints for operations

Low latency High throughput
- Efficient, pipelined runtime
- no per-record operations
- tunable latency / throughput tradeoff
- Async checkpoints
Low Watermarks

- We periodically send low-watermarks through the system to indicate the progression of event time.

For more details: “MillWheel: Fault-Tolerant Stream Processing at Internet Scale” by T. Akidau et. al.
Low Watermarks

Operators with multiple inputs always forward the lowest watermark

For more details: “MillWheel: Fault-Tolerant Stream Processing at Internet Scale” by T. Akidau et. al.
Bouygues Telecom

Mobile . Fixed . TV . Internet . Cloud

14M Clients

11.4M Mobile subscriber

2.6M Fixed customer

A very Innovative company
Leader 4G/4G+/UHMD
First Android based TV BOX
Bouygues Telecom

- Produce Mobile **QoE** indicators from massive network equipment’s event logs (**4 Billions/day**).

- **Goals:**
  - QoE (User) instead of QoS (Machine).
  - Real-time Diagnostic (**<60sec.** end-to-end latency)
  - Business Intelligence
  - Real-time alarming
  - Reporting
We ran stress tests on our biggest raw Kafka topic:
- A day of Data.
- **2 Billions** events (480Gib compressed).
- 10 Kafka partitions
- 10 Flink TaskManagers (Only 1GB Memory each)
Capital One
Fault Tolerance in streaming

- Failure with “at least once”: replay
Fault Tolerance in streaming

- Failure with “exactly once”: state restore

Restore from:

Final result:
Latency in stream record grouping

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

<table>
<thead>
<tr>
<th></th>
<th>Median latency</th>
<th>99th percentile latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flink, no fault tolerance</td>
<td>25 ms</td>
<td>50 ms</td>
</tr>
<tr>
<td>Flink, exactly once</td>
<td>1 ms</td>
<td></td>
</tr>
<tr>
<td>Storm, at least once</td>
<td>1 ms</td>
<td></td>
</tr>
</tbody>
</table>

Data Generator

Receiver: Throughput / Latency measure
Savepoints: Simplifying Operations

- Streaming jobs usually run 24x7 (unlike batch).
- **Application bug fixes**: Replay your job from a certain point in time (savepoint)
- **Flink bug fixes**
- **Maintenance** and system migration
- **What-If simulations**: Run different implementations of your code against a savepoint
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

(source) -> (tokenizer) -> (window count)

Complete pipeline online concurrently