APACHE FLINK™
STREAM AND BATCH PROCESSING IN
A SINGLE ENGINE

Akshaya Kalyanaraman
WHAT IS APACHE FLINK?

Batch Processing
process static and historic data

Data Stream Processing
realtime results from data streams

Event-driven Applications
data-driven actions and services
MOTIVATION

• In Lambda Architecture: Two separate execution engines for batch and streaming

• Unification of Batch and Stream Processing in a single framework, Flink.

• Apache Flink provides a highly flexible windowing mechanism.

• Flink supports different notions of time.
STREAM ANALYTICS
NOTIONS OF TIME

Event Time

Time when event happened.

12:23 am

Processing Time

Time measured by system clock

1:37 pm
STATEFUL STREAMING

Stateless Stream Processing

Stateful Stream Processing
<table>
<thead>
<tr>
<th>PROCESSING</th>
<th>SEMANTICS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>At-least once</strong></td>
<td><strong>Exactly Once</strong></td>
</tr>
<tr>
<td>May over-count after failure</td>
<td>Correct counts after failures</td>
</tr>
</tbody>
</table>

---

**End-to-end exactly once**
Correct counts in external system (e.g. DB, file system) after failure
Flink guarantees **exactly once**

**End-to-end exactly** once with specific sources and sinks (e.g. Kafka -> Flink -> HDFS)

Internally, Flink periodically takes **consistent snapshots** of the state without ever stopping computation
Windowing

- Window configured using **assigner** and optionally **trigger** and **evictor**.

- **Assigner:** assigns each record to logical windows.

- **Trigger:** defines when the operation associated with the window definition is performed.

- **Evictor:** determines which records to retain within each window.
• Below is a window definition with a range of 6 seconds that slides every 2 seconds (the assigner).
• The window results are computed once the watermark passes the end of the window (the trigger).

```java
stream
  .window(SlidingTimeWindows.of(Time.of(6, SECONDS), Time.of(2, SECONDS)))
  .trigger(EventTimeTrigger.create())
```
A global window creates a single logical group.

The example defines a global window (i.e., the assigner) that invokes the operation on every 1000 events (i.e., the trigger) while keeping the last 100 elements (i.e., the evictor).

```java
stream
  .window(GlobalWindow.create())
  .trigger(Count.of(1000))
  .evict(Count.of(100))
```
**FLINK STACK**

API & Libraries

- **FlinkML** - Machine Learning
- **Gelly** - Graph Processing
- **Table** - Relational
- **CEP** - Event Processing
- **Table** - Relational

Core

- **DataSet API** - Batch Processing
- **DataStream API** - Stream Processing

Deployment

- **Local** - Single JVM
- **Cluster** - Standalone, YARN, Mesos
- **Cloud** - AWS, Google

Runtime

- Distributed Streaming Dataflow
public class WordCount {
    public static void main(String[] args) throws Exception {
        // Flink's entry point
        StreamExecutionEnvironment env = StreamExecutionEnvironment
            .getExecutionEnvironment();
        DataStream<String> data = env.fromElements(
            "O Romeo, Romeo! wherefore art thou Romeo?",
            "Deny thy father and refuse thy name",
            "Or, if thou wilt not, be but sworn my love,",
            "And I'll no longer be a Capulet.");
        // split by whitespace to (word, 1) and sum up ones
        DataStream<Tuple2<String, Integer>> counts = data
            .flatMap(new SplitByWhitespace())
            .keyBy(0)
            .timeWindow(Time.of(10, TimeUnit.SECONDS))
            .sum(1);
        counts.print();
        // Today: What happens now?
        env.execute();
    }
}
Translates the API code to a data flow graph called **JobGraph** and submits it to the JobManager.
• All coordination via JobManager (master):
  • Scheduling programs for execution
  • Checkpoint coordination
  • Monitoring workers
TASK MANAGER

- All data processing in TaskManager (worker):
  - Communicate with JobManager via Actor messages
  - Exchange data between themselves via dedicated data connections
  - Expose task slots for execution
SCHEDULING

- Each ExecutionVertex will be executed one or more times
- The JobManager maps Execution to task slots
- Pipelined execution in same slot where applicable
SAMPLE QUERY

• dataStream
• count = input.map {m.split("1")};
• .keyBy(count%2); //keys by odd count (1) or even count (0)
• .window(TumblingEventTimeWindows.of(Time.seconds(3)));
• .apply (new CoGroupFunction () {...});
• .reduce(count);
SCHEDULING

Streaming Dataflow
(condensed view)

Operator chain

Task


Sink [1]

Subtask (= thread)


Streaming Dataflow
(parallelized view)
EXECUTION IN SLOTS

Processes

Threads
PIPELINED RESULTS

1101
0101
0100
1101
0101
PIPELINED RESULTS
PIPELINE RESULTS

0101
0100
1101
0101

Map → Pipelined Result

3
PIPELINED RESULTS

Map → Pipelined Result → KeyBy

0101
0100
1101
0101
PIPELINED RESULTS

0101
0100
1101
0101

Map → Pipelined Result → KeyBy
PIPELINE RESULTS

0100
1101
0101

0101

Pipelined Result

KeyBy
PIPELINED RESULTS

0100
1101
0101

0101

Pipelined Result

Odd (1 record)

KeyBy
PIPELINED RESULTS

Map -> Pipelined Result -> KeyBy

Odd (1 record)

0100
1101
0101
PIPELINED RESULTS

Map 0100

Pipelined Result

KeyBy

Odd (1 record)

1101
0101
0101
PIPELINED RESULTS

Map
0100

Pipelined Result

Odd (1 record)

Even (1 record)

1101
0101
0101
PIPELINED RESULTS

Map

1

Pipelined Result

1

Odd (1 record)

Even (1 record)

1101
0101
PIPELINED RESULTS

Map
1101

Pipelined Result
1

Odd (1 record)
Even (1 record)

0101
PIPELINED RESULTS

Map
1101

0101

Pipelined Result

Odd (2 records)
Even (1 record)
PIPELINED RESULTS

0101

Map → Pipelined Result → Odd (2 records) Even (1 record)

3
PIPELINED RESULTS

Map 0101 → Pipelined Result 3 → Odd (2 records) → Even (1 record)
PIPELINED RESULTS

Map
0101

Pipelined
Result

Odd (3 records)
Even (1 record)
PIPELINED RESULTS

Map → Pipelined Result

Pipelined Result
2

→

Odd (3 records)
Even (1 record)
PIPPCLINED RESULTS

Map -> Pipelined Result -> Odd (3 records), Even (2 records)
LATENCY AND THROUGHPUT

• When a data record is ready on the producer side, it is serialized and split into one or more buffers.

• A buffer is sent to a consumer either when it is full or when a timeout condition is reached.

• High throughput and low latency is achieved.
LATENCY AND THROUGHPUT

![Graph showing latency and throughput vs buffer timeout](image-url)

- **Latency**: 99th percentile in milliseconds
- **Throughput**: Average in millions of events/second

*Buffer timeout (milliseconds)*
FAULT TOLERANCE

ASYNCHRONOUS BARRIER SNAPSHOTTING

• An operator receives barriers from upstream and first performs an alignment phase.

• Then, the operator writes its state to durable storage.

• Once the state has been backed up, the operator forwards the barrier downstream.

• Eventually, all operators will register a snapshot of their state and a global snapshot will be complete.
FAULT TOLERANCE
COMPARISON WITH NAIAD

- Both Flink and Naiad make use of snapshotting mechanism for fault tolerance.

- Both Apache Flink and Naiad frameworks combine batch processing and stream processing.

- Both the frameworks support high throughput and low latency.

- NAIAD performs iterative and incremental computations, while Flink performs primarily data processing of stream and batch data.
CONCLUSION

- Apache Flink is designed to perform both stream and batch analytics.

- The streaming API provides the means to keep recoverable state and to partition, transform, and aggregate data stream windows.

- Flink treats batch computations by optimizing their execution using a query optimizer.
QUESTIONS?