Apache Flink - A System for Batch and Realtime Stream Processing
Introduction to Apache Flink

- Apache Flink is an open source Stream Processing Framework
- Low latency
- High throughput
- Stateful Operators
- Distributed Execution
- Developed at the Apache Software Foundation
- 1.0.0 released in March 2016, used in production
Flink Software Stack

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

**APIs & Libraries**

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

**Core**

- **Runtime**
  - Distributed Streaming Dataflow

**Deploy**

- **Local**
  - Single JVM
- **Cluster**
  - Standalone, YARN
- **Cloud**
  - GCE, EC2
Architecture

Flink Client

Code using API

Graph Builder & Optimizer

Dataflow Graph

Actor System

Job Manager

dataflow graph

Scheduler

Checkpoint Coordinator

Actor System

• task status
• heartbeats
• statistics
• trigger checkpoints

Actor System

Memory/IO Manager

Network Manager

Actor System

task slot

task slot

task slot

data streams

Actor System

task slot

task slot

task slot

Actor System

Memory/IO Manager

Network Manager
Dataflow Graphs

- all APIs (e.g. DataSet, DataStream,) compile to Dataflow Graphs
- (stateful) operators (filter, joins,..) = nodes
- data streams = links
- in parallel processing split into:
  - operators are executed in subtasks
  - stream partitions
- streams may p2p, broadcast, merge, fan-out, repartitions
Intermediate Data Streams

- **core abstraction for data exchange**
- **may or may not be materialized on disk**
- **pipelined execution**: data is continuously produced, buffered and consumed

**blocking data exchange**: output is generated, stored and then exchanged with the consumer. (->complete intermediate results of a stream must be stored)

![Diagram of data exchange](image)
Latency and Throughput

Data exchange based on buffers:
- data record ready => one/many buffers
- buffer is sent to consumer when it is full / time out

⇒ the large buffers increase throughput (less overhead)
⇒ low time out enable low latencies
  (real time processing = data is processed within a guaranteed time limit)
Control Events and Fault Tolerance

- Exemplary types of **control events**:
  - **check point barrier**: coordinate checkpoints by dividing stream into pre-checkpoint and post-checkpoint
  - **watermarks**: signaling the progress of event-time within the stream partition
  - **iteration barriers**: signals end of a superstep for iterative processing
- Control events are **injected into the stream** and provide operator nodes the position in the data set.
- Reliable execution with **exactly once**
- **consistency is guaranteed** (no availability on all nodes)
- Check-pointing and partial re-execution
- Based on the assumption that data source is **persistent and replayable** (e.g. files, Apache Kafka)
- Regular snapshots to **prevent unbounded recomputation**
Asynchronous Barrier Snapshotting

- barrier corresponds to a logical time => separate the stream to mark the snapshotted part
- barriers are injected into the stream
- wait until all barriers from input are received
- write out state to durable storage (=disk)
- checkpoint barriers are sent from upstream to downstream after checkpoint
- recovery: restart computation from the last successful snapshot
Iterative Data Flows

• Iterative algorithms are often employed for Data Mining, Machine Learning or Graph processing

• in other cloud-based computation frameworks (e.g. Hadoop, Spark):
  • run a loop in the client program
  • in each iteration a parallel execution is started
    (compare to k-Means on Hadoop)

• Flink provides an integrated iteration processing

• iteration step = special operators contain execution graphs

• iteration head and iteration tail are connected via feedback stream
  (handles what to keep between iterations)
Stream Processing with Dataflows

- **Flink manages time**: out-of-order events, windows, user-defined states
- **two notions of time**:
  - event time: time when the event is originated (e.g. timestamp)
  - processing time: wall-clock time of processing the event at worker X
- **Skew between both is possible** in distributed environments: objects may arrive out of order with respect to event time
- **low watermarks**: mark global progress measure (e.g. all events lower than timestamp t have entered an operator)
- Watermarks **originate at the sources of the graph**
- **operators decide** how to react
- operators with multiple inputs forward **minimal watermarks**
Stateful Streams Processing

- **stateless operators**: operator works independent for all inputs
  - for example simple map function in word count: \( \text{lambda } x: (x, 1) \)
  - no memory, not depending on the input order

- **stateful operators**: operator has an internal state
  - for example: regression function: \( a \cdot x + t \).
    \((a \text{ and } t \text{ are trained over the stream of input data})\)
  - the state stores models parameters

- **states are incorporated into the API by**:
  - operator interfaces registering local variables
  - operator-state abstractions for declaring portioned key-value states as there associated operations

- **states can be checkpointed**
Stream Windows

- Stateful operator configured via:
  - **assigner**: assigns each record to one/many logical windows
  - **trigger**(optional): states the time an operation on the windows is performed
  - **evictor**(optional): defines which records to retain in each window

- Predefined operator available e.g. sliding time window
- user-defined functions allow flexible customizing

Examples:

```java
stream
  .window(SlidingTimeWindows.of(Time.of(6, SECONDS), Time.of(2, SECONDS)))
  .trigger(EventTimeTrigger.create())

stream
  .window(GlobalWindow.create())
  .trigger(Count.of(1000))
  .evict(Count.of(100))
```
Batch Processing

- batch processing can be considered as special case of streams (bounded streams)
- Syntax for batch processing can be defined in a simpler way
- additional options for optimizing the processing might be possible

⇒ Flink offers additional functionality for batch processing
⇒ Blocked execution: break up large computations to isolated stages
⇒ No periodic snapshotting when overhead is large instead use last materialized intermediate stream
⇒ blocking is implemented as an operator explicitly waiting until the complete input is consumed => runtime environment does not distinguish
⇒ disk spill-off might become necessary
⇒ Flink provides a dedicated DataSet API with familiar functions e.g. map
⇒ Query optimization is used to transform API programs into efficient graphs
Query Optimization

• query optimizer is built on techniques from parallel databases:
  • plan equivalence
  • cost modeling
  • interesting-property propagation
• problem the operators have no predefined semantics (user defined functions!)
• cardinality and cost-estimation are hard to perform for the same reasons
• support execution strategies such as:
  • repartition and broadcast
  • sort-based grouping
  • sort- and hash-based joins
• Optimizer evaluated physical plans by interesting property propagation
• costs include disk I/O and CPU cost
• to handle user defined functions, hints are allowed
Memory Management

- Flink serializes data into memory segments instead of using the JVM heap.
- Operations work as much as possible on the binary data => reduces the overhead for serialization/deserialization.
- For arbitrary objects, Flink uses type inference and custom serialization.
- Binary representation and storing data off-heap reduces garbage collection overhead.
- Spilling data to disk is still fallback in case.

```java
public class WC {
    public String word;
    public int count;
}
```
Batch Iterations

- iterative methods are common in data analytics:
  - parallel gradient descent
  - expectation maximization
- Parallelization methods for iterative methods
  - Bulk Synchronous Parallel (BSP)
  - Stale Synchronous Parallel (SSP)
- Flink allows various iteration methods by providing iteration control events
- For example: in BSP mark begin and end of supersteps
- includes novel optimization concepts:
  - delta iterations: exploit sparse computational dependencies
**Word Count in Java**

```java
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
DataSet<String> text = readTextFile (input);
DataSet<Tuple2<String, Integer>> counts= text
.map (l -> l.split(“\W+”))
.flatMap ((String[] tokens, Collector<Tuple2<String, Integer>> out) -> {
    Arrays.stream(tokens)
        .filter(t -> t.length() > 0)
        .forEach(t -> out.collect(new Tuple2<>(t, 1)));
})
.groupBy(0)
.sum(1);
env.execute("Word Count Example");
```
### k-Means in Java

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

DataSet<Point> points = getPointDataSet(params, env);
DataSet<Centroid> centroids = getCentroidDataSet(params, env);

IterativeDataSet<Centroid> loop = centroids.iterate(params.getInt("iterations", 10));

DataSet<Centroid> newCentroids = points.map(new SelectNearestCenter()).withBroadcastSet(loop, "centroids").map(new CountAppender()).groupBy(0).reduce(new CentroidAccumulator()).map(new CentroidAverager());

DataSet<Centroid> finalCentroids = loop.closeWith(newCentroids);

DataSet<Tuple2<Integer, Point>> clusteredPoints = points.map(new SelectNearestCenter()).withBroadcastSet(finalCentroids, "centroids");
```
References

- https://flink.apache.org/