Reading Assignment 3

October 14, 2018

Anna Hedstrom annaheds@kth.se
Sandra Pico Oristrell sandrapo@kth.se

1 Motivation

1.1 Spark Streaming

The paper *"Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters"* by Zaharia et al., (2013) [2], proposes a stream processing model called Discretized Streams (`DStreams`). Very briefly, this model breaks streaming data into a series of *short-interval*, *deterministic* batch jobs and stores its *state* in a sequence of lineage-based data abstractions (read: `RDDs`). The `DStreams` are embedded in a larger system API, which the authors appropriately name, Spark Streaming.

The programming model is built in response to the new demands that ever-larger stream processing applications pose. Today, we know that data is most often received in real-time and also, most valuable at the time of its arrival. This means that automatic recovery from faults and stragglers as well as maintenance of low latency system is critical for usability. In this context, the authors explain why traditional stream processing models, such as Storm, MapReduce Online and TimeStream, does not suffice - both the option of *replication* and *upstream backup* schemes are too expensive (and slow). Thus to improve efficiency (and avoid replication overhead), they built Spark Streaming.

1.2 Apache Flink

The paper *"Apache Flink: Stream and Batch Processing in a Single Engine"* by Carbone et al., (2015) [1], presents a event-based streaming model with an architecture of a distributed computing system, built with the goal to unify stream- and batch data processing under a single shared execution engine. The model follows a data flow graph and is motivated by the philosophy that seemingly different types of processing (think: interactive-, iterative-, bounded (batch-), unbounded (stream-) processing) can be expressed under one (fault-tolerant) roof.

In our view, what makes Flink interesting is primarily the fact that it provides such wide applicability (and flexibility) in its use cases, while still keeps a relatively simple interface, that is the the common processor ("the runtime engine") that connects all the seemingly diverse components of the Flink ecosystem.

2 Contribution

2.1 Spark Streaming

While park Streaming is built on top of Spark, the authors mention that significant optimisations and changes were necessary. We list the most important contributions below.

- **Parallelise recovery.** In case of node failure, rather than (slowly) replicating or executing upstream backups, Spark Streaming's *parallel recovery* makes the whole cluster to partake in the recovery process. Each node in the cluster recompute parts of the lost `RDD` and thus enable fast recovery.
- **Minimises stragglers.** To mitigate stragglers, *speculative execution* is implemented. Rather than executing slow re-computations or following an expensive synchronisation protocol, the method proposed in Spark Streaming is based on simple threshold detection (read: time limits for nodes).
- **Keeps latency low on scaled systems.** By chopping incoming streaming data into micro-sized batches, seconds-range latency can be maintained. Since Spark Streaming is built on top of `RDD` re-computation, we further note that *lineage graph* is sufficient for recovery (no replication needed).
- **Integrates well with batch- and interactive models.** Since the Spark Streaming follows similar API and execution model as batch platforms, combining queries from such models is easy. Also, by adding a `Scala` console to the Spark Streaming program, users can interact with the running stream i.e., run ad-hoc queries.

- **Supports stateful processing**. Apart from offering a rich set of operators (implied by the use of `RDDs`), Spark Streaming also handles stateful processing. Since the `DStreams` are deterministic to its nature, the user can accumulate and aggregate the processing from the start of the streaming job, without expensive synchronisation schemes.

2.2 Apache Flink

Flink makes many important contributions such as **high through-put** and **low latency** in the range of milliseconds, and we have listed a selected few below.

- **Provides specialised query-optimiser for batch-processing**. While the authors recognise that batch computations are truly just a special case of streaming computations, to ensure high performance - Flink treats batch processing differently. To choose the best physical plan available for execution, Flink uses a query optimiser that makes a cost-based decision.
- **Offers various control events.** Flink provides different types of control events including; *checkpoint barriers* to coordinate checkpoints in pre- and post-checkpoint, *watermarks* to get the current progress of an event time within a stream (expecting some late elements) as well as *iteration barriers* which signals the reach of a so-called superstep.
- **Light-weight fault tolerance.** Flink handles faulty nodes by taking global snapshots of the state of the operators, including the position of the input stream, at regular intervals. To ensure consistency in snapshots, Flink provides a mechanism that periodically inject checkpoints into the data stream via *asynchronous barriers* that trigger operators to emit all records that depend only on records before the barrier. By this, Flink can guarantee **exactly-once state updates**, i.e., pauses become redundant.
- **Offers custom user-defined window semantics.** Flink provides different means to specify window semantics. Examples include *tumbling window*, *sliding window* and *session windows*.

3 Solution

3.1 Spark Streaming

So far we know that Spark Streaming splits up the live stream input data into smaller batches of n seconds and then treats each batch as a `RDD` abstraction. While this treatment implies that that the user can leverage `RDD` operations to manipulate the `Dstream` batches, one should note that they are categorised in a slightly different way. Rather than labelling after *transformations* and *actions*, `DStream` operations are divided into *input-* and *output operations* and *transformations*. Next, we note that the **system architecture** also differs, consisting of three main components:

- **A master** which schedules tasks for workers and records the `DStreams` *lineage graph*.
- **Worker nodes** that receive input data and store partitions and compute `RDDs`.
- **A client library** that send data to the system.

For the user, the `StreamingContext` is the main entry point for all streaming functionality (e.g., setting the time interval for batches). To enable stateful processing, Spark Streaming offers two main functions:

- The `updateStateByKey` operator that manages the state per key and executes `DStream` on the whole range of keys, making performance proportional to the size of the state.
- The `mapWithState` operator that processes partial updates, executing only on the keys available in the last batch, making performance proportional to the size of the batch.

3.2 Apache Flink

While constantly evolving, the architecture of Flink is built out of four main components, namely; **deployment, core, APIs**, and **libraries**.

The **core runtime engine** of Flink is a directed acyclic graph (DAG), made up of stateful operators connected to data streams.

There are two main APIs that make up Flink namely, the **DataSet API** (for batch processing) and the **DataStream API** (for stream processing). Since both APIs make runtime programs executable for the processor, Flink eventually boils down to one common representation - the data flow graph. We find this smart representation of the core processor the most interesting (and important) aspect of Flink's architecture.

In addition to the core APIs, Flink provides **domain-specific libraries** e.g., `Gelly` for graph processing, `FlinkML` for machine learning and `Table` for "SQL-like" queries.

More broadly, the Flink process model is made up of three main components:

- **A client** which takes and transform the program code and then submits it to the JobManager.
- **The Job Manager nodes** which coordinates and schedules the distributed execution of the data flow as well as tracks the state, progress and cases of recovery (while persisting only minimal amount of meta data).
- **(At least) one Task Manager** which is responsible for the actual data processing, meaning that it executes the streams and maintains buffer pools for one (or more) operators and then reports back the status to the JobManager.

4 Discussion

4.1 Spark Streaming

Most generally, we very much sympathise with the authors' intent to *"blur the line between live and offline processing"*, that is simplifying data processing at large. However, as every programming model, there are drawbacks. We discuss the most important ones below.

- **Manual control of time-intervals.** As user of Spark Streaming, one has to determine the batch interval manually, which is both error prone and costly with respect to time. This is even more alarming bearing in mind that the batch interval is directly influencing the trade-off between latency and throughput of the whole workload.
- **In-efficient memory usage.** Given that the system actually generates a completely new `RDD` to store after each batch is processed, costs associated with memory usage increase. Although the user could store only the deltas between states, it is still is major drawback of the system.
- **Fixed minimum latency.** The fact that `DStreams` operates on small batches, raises the minimum latency. While most real-time applications today accepts some seconds of delay, the fact that it is at minimum 1-2 seconds reduces the performance of the system.

4.2 Apache Flink

Although Flink often seems to shine in comparison to Spark, we also recognise its limitations.

- **Costly memory-usage.** Similar to Spark Streaming, to run in-memory, Flink also requires a lot of RAM, making costs gradually increase.
- **Prefers mid-high level hardware.** In comparison to MapReduce, Flink typically needs mid to high-level hardware to enhance its performance by caching data in-memory.
- **Less established**. While Flink often showcase best performance, compared to Spark, it is not as well integrated with vendors, so less support in case of errors can be expecteds.

References

[1] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[2] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.