DSP Frameworks

Corso di Sistemi e Architetture per Big Data
A.A. 2016/17

Valeria Cardellini
DSP frameworks we consider

- **Apache Storm**
- **Twitter Heron**
  - From Twitter as Storm and compatible with Storm
- **Apache Spark Streaming**
  - Reduce the size of each stream and process streams of data (*micro-batch processing*)
  - Lab on Spark Streaming
- **Apache Flink**
- **Cloud-based frameworks**
  - Google Cloud Dataflow
  - Amazon Kinesis
Twitter Heron

• Realtime, distributed, fault-tolerant stream processing engine from Twitter
• Developed as direct successor of Storm
  – Released as open source in 2016
    https://twitter.github.io/heron/
  – De facto stream data processing engine inside Twitter, but still in beta
• Goal of overcoming Storm’s performance, reliability, and other shortcomings
• Compatibility with Storm
  – API compatible with Storm: no code change is required for migration
Heron: in common with Storm

• Same terminology of Storm
  – Topology, spout, bolt
• Same stream groupings
  – Shuffle, fields, all, global
• Example: WordCount topology
Heron: design goals

• Isolation
  – Process-based topologies rather than thread-based
  – Each process should run in isolation (easy debugging, profiling, and troubleshooting)
  – Goal: overcoming Storm’s performance, reliability, and other shortcomings

• Resource constraints
  – Safe to run in shared infrastructure: topologies use only initially allocated resources and never exceed bounds

• Compatibility
  – Fully API and data model compatible with Storm
Heron: design goals

• Back pressure
  – Built-in back pressure mechanisms to ensure that topologies can self-adjust in case components lag

• Performance
  – Higher throughput and lower latency than Storm
  – Enhanced configurability to fine-tune potential latency/throughput trade-offs

• Semantic guarantees
  – Support for both at-most-once and at-least-once processing semantics

• Efficiency
  – Minimum possible resource usage
Heron topology architecture

• Master-work architecture
• One Topology Master (TM)
  – Manages a topology throughout its entire lifecycle
• Multiple Containers
  – Each Container multiple Heron Instances, a Stream Manager, and a Metrics Manager
  – Containers communicate with TM to ensure that the topology forms a fully connected graph
Heron topology architecture

- Topology Master
- Logical Plan, Physical Plan, and Execution State
- Sync Physical Plan
- ZK CLUSTER
- Stream Manager
- Metrics Manager
- CONTAINER

Valeria Cardellini - SABD 2016/17
Heron topology architecture

- Stream Manager (SM): routing engine for data streams
  - Each Heron connects to its local SM, while all of the SMs in a given topology connect to one another to form a network
  - Responsible for propagating back pressure
Valeria Cardellini - SABD 2016/17
Heron environment

- Heron supports deployment on Apache Mesos
- Heron can also run on Mesos using Apache Aurora as a scheduler
Batch processing vs. stream processing

- Batch processing is just a special case of stream processing
Batch processing vs. stream processing

- Batched/stateless: scheduled in batches
  - Short-lived tasks (Hadoop, Spark)
  - Distributed streaming over batches (Spark Streaming)

- Dataflow/stateful: continuous/scheduled once (Storm, Flink, Heron)
  - Long-lived task execution
  - State is kept inside tasks
Native vs. non-native streaming

Non-native streaming

while (true) {
    // get next few records
    // issue batch computation
}

Native streaming

while (true) {
    // process next record
}
Apache Flink

• Distributed data flow processing system
• One common runtime for DSP applications and batch processing applications
  – Batch processing applications run efficiently as special cases of DSP applications
• Integrated with many other projects in the open-source data processing ecosystem
• Derives from Stratosphere project by TU Berlin, Humboldt University and Hasso Plattner Institute
• Support a Storm-compatible API
Flink: software stack

- On top: libraries with high-level APIs for different use cases, still in beta
Flink: programming model

- Data stream
  - An unbounded, partitioned immutable sequence of events
- Stream operators
  - Stream transformations that generate new output data streams from input ones
Flink: some features

- Supports stream processing and windowing with **Event Time** semantics
  - Event time makes it easy to compute over streams where events arrive out of order, and where events may arrive delayed.

- Exactly-once semantics for stateful computations

- Highly flexible streaming windows
Flink: some features

• Continuous streaming model with backpressure
• Flink's streaming runtime has natural flow control: slow data sinks backpressure faster sources
Flink: APIs and libraries

- Streaming data applications: **DataStream API**
  - Supports functional transformations on data streams, with user-defined state, and flexible windows
  - Example: how to compute a sliding histogram of word occurrences of a data stream of texts

```scala
case class Word(word: String, freq: Long)
val texts: DataStream[String] = ...
val counts = text
  .flatMap { line => line.split("\W+") }
  .map { token => Word(token, 1) }
  .keyBy("word")
  .timeWindow(Time.seconds(5), Time.seconds(1))
  .sum("freq")
```
Flink: APIs and libraries

- Batch processing applications: **DataSet API**
- Supports a wide range of data types beyond key/value pairs, and a wealth of operators

```scala
case class Page(pageId: Long, rank: Double)
case class Adjacency(id: Long, neighbors: Array[Long])

val result = initialRanks.iterate(30) { pages =>
  pages.join(adjacency).where("pageId").equal To("id") {
    (page, adj, out: Collector[Page]) => {
      out.collect(Page(page.pageId, 0.15 / numPages))
    }
    val nLen = adj.neighbors.length
    for (n <- adj.neighbors) {
      out.collect(Page(n, 0.85 * page.rank / nLen))
    }
  } .groupBy("pageId").sum("rank")
```
Flink: program optimization

- Batch programs are automatically optimized to exploit situations where expensive operations (like shuffles and sorts) can be avoided, and when intermediate data should be cached.
Flink: control events

• Control events: special events injected in the data stream by operators

• Periodically, the data source injects **checkpoint barriers** into the data stream by dividing the stream into pre-checkpoint and post-checkpoint
  • More coarse-grained approach than Storm: acks sequences of records instead of individual records

• **Watermarks** signal the progress of event-time within a stream partition

• Flink does not provide ordering guarantees after any form of stream repartitioning or broadcasting
  – Dealing with out-of-order records is left to the operator implementation
Flink: fault-tolerance

- Based on Chandy-Lamport distributed snapshots
- Lightweight mechanism
  - Allows to maintain high throughput rates and provide strong consistency guarantees at the same time
Flink: performance and memory management

• High performance and low latency

• Memory management
  – Flink implements its own memory management inside the JVM
Flink: architecture

• The usual master-worker architecture
Flink: architecture

- **Master (Job Manager):** schedules tasks, coordinates checkpoints, coordinates recovery on failures, etc.
- **Workers (Task Managers):** JVM processes that execute tasks of a dataflow, and buffer and exchange the data streams
  - Workers use task slots to control the number of tasks it accepts
  - Each task slot represents a fixed subset of resources of the worker
Flink: application execution

- Jobs are expressed as data flows
- The job graph is transformed into the execution graph
- The execution graph contains information to schedule and execute a job
Flink: infrastructure

• Designed to run on large-scale clusters with many thousands of nodes

• Provides support for YARN and Mesos
DSP in the Cloud

• Data streaming systems are also offered as Cloud services
  – Amazon Kinesis Streams
  – Google Cloud Dataflow
• Abstract the underlying infrastructure and support dynamic scaling of the computing resources
• Appear to execute in a single data center
Google Cloud Dataflow

- Fully-managed data processing service, supporting both stream and batch execution of pipelines
  - Transparently handles resource lifetime and can dynamically provision resources to minimize latency while maintaining high utilization efficiency
  - On-demand and auto-scaling
- Provides a unified programming model and a managed service for developing and executing a wide range of data processing patterns including ETL, batch computation, and continuous computation
  - Apache Beam model
Google Cloud Dataflow

• Seamlessly integrates with other Google cloud services
  – Cloud Storage, Cloud Pub/Sub, Cloud Datastore, Cloud Bigtable, and BigQuery

• Apache Beam SDKs, available in Java and Python
  – Enable developers to implement custom extensions and choose other execution engines
Apache Beam

• A new layer of abstraction
• Provides advanced unified programming model
  – Allows to define batch and streaming data processing pipelines that run on any execution engine (for now: Flink, Spark, Google Cloud Dataflow)
  – Well suited for embarrassingly parallel data processing tasks
• Translates the data processing pipeline defined by the user with the Beam program into the API compatible with the chosen distributed processing engine
• Developed by Google and recently released as open-source top-level project (May 2017)
Towards strict delivery guarantees

• Most frameworks provide weaker delivery guarantees (e.g., at-least-once in Storm)
• Flink and Google Dataflow offer stronger delivery guarantees (i.e., exactly-once)
• MillWheel: Google’s internal version of Google Dataflow
  – **Exactly-once** low latency stream processing as follows:
    • The record is checked against de-duplication data from previous deliveries; duplicates are discarded
    • User code is run for the input record, possibly resulting in pending changes to timers, state, and productions
    • Pending changes are committed to the backing store
    • Senders are ACKed
    • Pending downstream productions are sent

Elaborazione real-time di data streaming nel Cloud

Definisce:

- **Stream**: sequenza di record
- **Shard**: numero di “nodi” su cui suddividere lo stream, determinato in base al data rate desiderato in input ed output
Amazon Kinesis Streams

• Allows to build custom applications that process and analyze streaming data
Kinesis Streams: components

- **Stream**: ordered sequence of data records
  - Data producers write data records to Kinesis streams
  - Data records in the stream are distributed into shards

- **Data record**
  - Record \(=\) \{sequence, partition key, data blob\}
  - Data blob: immutable sequence of bytes (up to 1 MB)
  - Kinesis Streams **does not** inspect, interpret, or change the data in the blob

- **Shard**: uniquely identified group of data records in a stream
  - It is the base unit of capacity: up to 1MB/sec of data and 1000 PUT transactions/sec
  - Partition key used to group data by shard within a stream
  - Used also for service pricing [http://amzn.to/2szRTkG](http://amzn.to/2szRTkG)
  - Data records are stored in shards temporarily (24 hours by default)
Kinesis Streams: consuming data

- Kinesis Streams is used to capture streaming data
- An application reads data from a Kinesis stream as data records, then uses the Kinesis Client Library (KCL) for the processing logic
  - KCL takes care of: load-balancing across multiple EC2 instances, responding to instance failures, check-pointing processed records, reacting to re-sharding (that adjusts the number of shards)
A new breadth of frameworks

• Lambda architecture
  – Data-processing design pattern to handle massive quantities of data and integrate batch and real-time processing within a single framework

Source: https://voltdb.com/products/alternatives/lambda-architecture
References


• Overview on DSP frameworks