Distributed Stream Data Processing

Pelle Jakovits

23 November 2018, Tartu
Outline

• Stream data processing
  – Use cases
  – Models

• Stream data processing frameworks
  – Spark Streaming
  – Spark Structured Streaming
  – Apache Storm
  – Apache Kafka
Stream data processing use cases

• **Anomaly Detection**
  – Detect problems in real-time (cyber intrusions, financial fraud, etc)
  – Continuously collect and analyse network traffic, transactions, user behaviour

• **Predictive maintenance**
  – Collect and process performance data from deployed devices
  – Forecast potential faults and service disruptions, predict maintenance cycles

• **Clickstream analytics**
  – Collect and analyse user clicks, routes and behaviour
  – Extract frequent patterns to improve user engagement
  – Personalized recommendations
How to manage stream data in Hadoop?

- Collect data to HDFS
- Process every **N hours** using MapReduce, Pig or Spark

**How long before incoming data is processed?**

**What about time critical applications?**

**How to push results to external systems as output stream?**

**Process only the newly added data or the whole data set?**
Stream data processing frameworks

• Frameworks/extensions specifically designed for:
  – Low latency data processing
  – Dynamically process incoming data streams
  – Aggregate data processed at different time periods
  – Push results to external systems as output streams

• Two main approaches/models:
  1. Micro-Batch processing
  2. Real-Time stream data processing
Stream processing models

• **Micro-Batch processing**
  – Collect incoming data into a batch/buffer
  – Processing one batch at a time
  – High throughput, High latency
  – Spark Streaming

• **Real-time processing**
  – Process each incoming message right away
  – Low latency, lower throughput
  – Apache Storm, Apache Flink
Spark Streaming
Spark Streaming

• Stream data processing extension on-top of Spark Resilient Distributed Datasets (RDD)
  – Inherently parallel and scalable
• Utilizes the micro-batch processing model
  – Batch size can be freely user-configured or dynamically optimized
  – Batch size of 1 is possible, but not recommended.
• Can reuse most of the existing Spark code
• Lineage based fault tolerance can't handle unbounded dependency chain
  – Automatic fault recovery requires checkpointing to HDFS
Spark streaming concepts

- **Micro-Batch** is a collection of input records processed at once
  - Contains all incoming data that arrived in the last batch interval
- **Batch interval** is the duration in seconds between micro-batches
- Spark **Streaming job** periodically executes a separate **Spark job** for each **Micro-Batch**
- A new collection of output records are returned for each batch
  - Even if the micro-batch is empty (by default)
Input streams

- File systems & TCP socket connections are directly available in StreamingContext API as streaming sources
- File based streaming
  - Reading data from HDFS compatible systems (HDFS, S3, NFS)
    - `streamingContext.textFileStream(dataDirectory)`
  - Spark monitors input directory for any new files
  - A file is considered as input if its modification time is inside the current window
  - Spark tracks processed files, later modifications have no effect
  - Should avoid writing incomplete files into the input directory
Advanced input streams

- Spark uses receivers to manage streams from external systems
- Receivers run inside Spark cluster, parse incoming streams and push data into Spark Streaming
- Advanced sources are available through extra utility libraries that implement a custom receiver
  - Kafka, Flume, Kinesis, MQTT, ZeroMQ
- User-defined receivers can be implemented for custom sources
  - Reliable and un-reliable receivers
  - Advanced features are required for fault tolerance and 24/7
    - Data arrival acknowledgements
    - Block generation and rate control
Creating Streaming Applications

```
sc = SparkContext(appName= "WordCount")
lines = sc.textFile(sys.argv[1])
counts = lines.flatMap(lambda line: line.split(" "))
    .map(lambda x: (x, 1))
    .reduceByKey(lambda a, b: a+b)
counts.saveAsTextFile(sys.argv[2])
sc.stop()
```

```
sc = SparkContext(appName= "StreamWordCount")
ssc = StreamingContext(sc, 1)
lines = ssc.textFileStream(sys.argv[1])
counts = lines.flatMap(lambda line: line.split(" "))
    .map(lambda x: (x, 1))
    .reduceByKey(lambda a, b: a+b)
counts.saveAsTextFiles(sys.argv[2], "txt")
ssc.start()
ssc.awaitTermination()
sc.stop()```
Window operations

- It is possible to use Window functions over micro-batches
- Window functions take 2 window size parameters:
  - **Window length** - The duration of the window (3 on the figure).
  - **Sliding interval** - The interval or step of the window operation (2 on the figure).
Windowed WordCount

• Generate word counts over the last 30 seconds of data, every 10 seconds:

```python
pairs = lines.flatMap(lambda l: l.split(" ")).map(lambda x: (x, 1))
counts = pairs.reduceByKeyAndWindow(lambda x, y: x + y, None, 30, 10)
```

• Can define an inverse reduction function for removing elements from the window
• Allows to avoid recomputing the whole window each time

```python
counts = pairs.reduceByKeyAndWindow(
    lambda x, y: x + y,
    lambda x, y: x - y,
    30, 10
)
```
Manage output

• Save to file storage (HDFS) by default
  – SaveAsTextFiles()
  – SaveAsObjectFiles() *(Not available in Python)*
  – SaveAsHadoopFiles() *(Not available in Python)*

• Pushing data to external systems
  – ForeachRDD(VoidFunction)
  – Apply a user defined function on every RDD partition
  – Defines how/where that partition's data is migrated
ForeachRDD(Function)

• Apply a void function on each RDD partition
• Useful for creating output streams or pushing data to an external systems
  – Data brokers, work queues, databases
• Create a connection to an external system and push the content of RDD partitions there

• ForeachRDD example
  – Publish results to MQTT broker
  – Implement a function that uses MQTT library to publish RDD content as MQTT messages
  – rooms.foreachRDD( publishToMQTT( MQTT_BROKER_URL, MQTT_TOPIC) )
ForeachRDD example

```
VoidFunction<JavaRDD<String>> publishToMQTT(String broker, String topic) {
    return rdd -> {
        rdd.foreachPartition(recordIterator -> {
            if (recordIterator.hasNext()){
                MqttConnectOptions options = new MqttConnectOptions();
                MqttClient mqttClient = new MqttClient(broker,
                                                      MqttClient.generateClientId());
                mqttClient.connect(options);
                recordIterator.forEachRemaining(record -> {
                    MqttMessage msg = new MqttMessage(record.getBytes());
                    mqttClient.publish(topic, msg);
                });
                mqttClient.disconnect();
                mqttClient.close();
            }
        });
    }
};;
```
Streaming web interface

• Changes in the input data stream can have significant effects on the performance
• Spark provides graphical web interface for tracking the performance of streaming applications
• Two most important metrics are:
  – **Processing Time** - The time it took process a micro-batch.
  – **Scheduling Delay** - How long batches wait in the queue for the previous batches to finish.
• Must avoid Infinitely increasing queues - when **processing time** is consistently longer than **batch interval**
  – Increase allocated computing resources
  – Increase the batch size
  – Turn on backpressure
## Streaming Statistics

Running batches of 1 second for 23 minutes 55 seconds since 2018/01/25 07:47:29 (1420 completed batches, 16860 records)

### Timelines (Last 1000 batches, 0 active, 1000 completed)

#### Input Rate
- Receivers: 1 / 1 active
- Avg: 16.53 records/sec

#### Scheduling Delay
- Avg: 3 minutes 21 seconds

#### Processing Time
- Avg: 847 ms

#### Total Delay
- Avg: 3 minutes 22 seconds
Checkpointing based Fault tolerance

• Spark Lineage would require unbounded partition dependencies

• **Metadata checkpointing**
  – **Configuration** of the streaming application
  – **DStream operations** of the streaming application
  – Incomplete & queued micro-batches

• **Data checkpointing**
  – **Required for stateful** transformations and window operations that combine data across multiple batches

• Can enable **Write ahead logs** to achieve fault-tolerance guarantees
  – Received data gets written into a write-ahead log inside checkpoint directory
  – Avoid data loss on driver failures
Dynamic allocation

- Enables **adaptive streaming applications** by scaling computing resources based on incoming load variations
  - `spark.dynamicAllocation.enabled`
  - `spark.dynamicAllocation.minExecutors`
  - `spark.dynamicAllocation.maxExecutors`
- Automatically adjusts the number of Spark **executors** when processing time becomes longer than batch interval
- Prevents resources from being wasted when the processing time is short
- Requires external **shuffle service** to avoid data loss on scale down
- Can set **minExecutors** to 0, but this introduces cold-start problem
Back-Pressure

- Dynamic Allocation has no effect when already running on max capacity.
- Backpressure can be enabled to make sure that Spark Streaming application stays stable and can handle sudden spikes in load.

- **Back-Pressure** is the concept of **Reactive Streams**
  - 2013 initiative between engineers at Netflix, Pivotal, and Lightbend
  - Receiving side should not be forced to buffer unbounded amount of data.

- Upstream components (Spark Receivers) are notified when Spark cluster can't handle current stream load.
- Maximum rate of Spark Receivers is dynamically adjusted
  - Receiver should then modify how it ingests data from the external system.
- Decisions are made based on current streaming metrics
  - Processing Time, Scheduling Delay.
Structured Streaming

- Based on Spark DataFrames instead of RDD
- Input is a table that is being continuously extended by new rows from an incoming data stream
- Users write Batch-like DataFrame or SQL queries
- Spark executes user operations incrementally on the unbounded input table
- Has semantics to deal with data that arrives late
- Intermediate aggregation dataframes can be kept in memory
  - Continuously updated by preceding queries
Structured Streaming concept

Data stream as an unbounded table

new data in the data stream =
new rows appended to a unbounded table

https://spark.apache.org/docs/2.2.0/structured-streaming-programming-guide.html
Streaming WordCount example

```scala
# Define input stream
lines = spark.readStream.format("socket") \\
    .option("host", "localhost").option("port", 9999) \\
    .load()

# Split the lines into words
words = lines.select(explode(split(lines.value," ")).alias("word"))
    .groupBy("word").count()
    
# Define output and initiate computation
query = wordCounts \\
    .writeStream \\
    .outputMode("complete") \\
    .format("console") \\
    .start()

query.awaitTermination()
```

# Split the lines into words
# Generate running word count
# Wait until terminated
Triggers and Output modes

- **Trigger** – Controls when the streaming query will be executed
  - *Fixed interval micro-batches* – run once for every micro-batch *(Default)*
  - *One-time micro-batch* – Run streaming application in Batch mode
  - *Continuous trigger* – Experimental Real-Time mode (~ 1 ms goal)

- **Output Mode**
  - *Append mode* - Only new rows since the last trigger will be written out *(Default)*
  - *Complete mode* - The whole DataFrame will be written out after every trigger
  - *Update mode* - Only rows modified since the last trigger will be written out.

```scala
query = wordCounts 
  .writeStream 
  .outputMode("complete") 
  .format("console") 
  .trigger(processingTime='2 seconds') 
  .start()
```
Spark Streaming issues

- Apache Spark is a *Jack of all trades*
  - *Streaming Machine Learning* (native Streaming Linear Regression & KMeans)
- Not a good support for real Real-Time streaming
  - Micro batching causes high latency
  - Small batches reduce efficiency of Spark parallelization
  - Real-Time triggers are currently in experimental stage in Spark
- Default Spark configuration options may not be suitable for many streaming applications
  - Especially for long running (24/7) streaming jobs
  - Choosing the right batch interval is not a simple task
  - May need significant effort to find suitable configuration
Dealing with issues

- Dynamic allocation when below maximum capacity of the cluster
- BackPressure helps with spikes in stream arrival rate
  - Depend on the framework for optimizing streaming configuration
  - May not work on all receivers
  - Receiver fault tolerance and memory issues
- Use frameworks specialized on real-time stream processing to achieve low latency until Spark Real-Time triggers are out of experimental stage
  - Such as Apache Storm and Apache Flink
- Data flow and queue systems can be used between data sources and Apache Spark to manage incoming and outgoing data streams
  - Such as Apache Kafka
  - Can avoid having to create Spark Receivers by depending on Kafka adapters
Apache Storm

• Designed for low-latency Real-time stream processing
• Uses Apache Thrift for creating language independent applications
  – Any individual service may use a different language
  – May be implemented in multiple languages for portability
  – C++, Java, Python, PHP, Ruby, Erlang, Haskell, C#, JavaScript, etc.
• Storm applications manipulate streams of tuples
  – Tuples can contain labelled objects of any type
• Data Stream processing is defined by a **Topology**, which consists of DAG of **Spots** and **Bolts**
Storm Topology

- Storm **Topology** is a network of spouts and bolts
- **Spout** is a source of data stream
- **Bolt** defines a single (scalable) data processing operation
Storm applications

• Storm integrates with most common queueing and database technologies
  – Kestrel, RabbitMQ / AMQP, Kafka, Amazon Kinesis
• Users can implement Spouts for other external systems
• Topologies, Spouts and Bolts can be defined in any language
  – Natively support for JVM languages
  – Spouts and bolts written in other languages communicate to Storm using JSON-based protocol
  – Adapters for Ruby, Python, JavaScript, Perl.
Scalability

- Storm topologies run across a cluster of nodes.
- Each Spout or Bolt can be scaled individually.
- Parallelism can be modified using Storm command-line rebalance command.

http://storm.apache.org/releases/current/Tutorial.html
Storm Example

```java
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("spout", new RandomSentenceSpout(), 5);
builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("spout");

conf.setMaxTaskParallelism(3);
LocalCluster cluster = new LocalCluster();
cluster.submitTopology("word-count", conf, builder.createTopology());
```

- **Shuffle grouping**: Distribute tuples randomly but evenly across the bolt instances
- **Fields grouping**: Distribute tuples by a field. Tuples with the same field are always routed to the same task.
Fault tolerance

- Workers that execute a subset of a topology are restarted on failure
- If the whole node dies, workers are moved to other nodes
- Storm guarantees at-least-once processing of input tuples.
  - Re-processing happens only when there are failures
- It is possible to have exactly-once processing guarantee by using Trident
  - Higher-level abstraction on top of Storm
  - Similar to having Apache Pig over MapReduce
  - However, it has significant performance overhead
Apache Kafka

- Distributed streaming data broker, queue and storage
- Manage and control data streams
- Publish-Subscribe model with consumer groups
  - Subscribed data is load balanced between consumers in one group
  - Different consumer groups receive same data
- Initially designed for "moving" data between systems
- But also supports stream data processing
  - Filtering, joins, aggregations, etc.
Creating Kafka applications

- Kafka has four main APIs:
  - **Producer API** - Publishing messages to Kafka topics, creating a stream
  - **Consumer API** - Subscribing to Kafka topics for receiving the data stream
  - **Streams API** - Creating stream processor applications that consume input streams, transform data and produce output streams
  - **Connector API** - Building and running adapters that connect Kafka topics to external systems

- Kafka runs as a cluster that can be deployed across multiple datacenters
- Messages are stored to persistent storage and replicated
- Producers can wait for acknowledgement until messages are fully replicated and fault tolerance is guaranteed.
Kafka & Spark
Multi mode integration scenario

- Store all event in an historical archive
- low-latency event-time aggregation
- Alerts
- Real-time Dashboards
- batch reporting
- Exactly once aggregation
- Exactly once

That’s All

• Next weeks practice session is
  – Stream Data Processing with Spark Streaming

• Next week's lecture is
  – Graph processing with Bulk Synchronous Parallel model