Spark Streaming and GraphX

Amir H. Payberah
amir@sics.se

SICS Swedish ICT
Spark Streaming
Many applications must process large *streams of live data* and provide results in *real-time*.

- Wireless sensor networks
- Traffic management applications
- Stock marketing
- Environmental monitoring applications
- Fraud detection tools
- ...
Database Management Systems (DBMS): data-at-rest analytics

- Store and index data before processing it.
- Process data only when explicitly asked by the users.
Stream Processing Systems

- Database Management Systems (DBMS): data-at-rest analytics
  - Store and index data before processing it.
  - Process data only when explicitly asked by the users.

  - Processing information as it flows, without storing them persistently.
DBMS vs. SPS (1/2)

- **DBMS**: persistent data where updates are relatively infrequent.

- **SPS**: transient data that is continuously updated.
DBMS vs. SPS (2/2)

- **DBMS**: runs queries just once to return a complete answer.

- **SPS**: executes standing queries, which run continuously and provide updated answers as new data arrives.
Core Idea of Spark Streaming

- Run a streaming computation as a series of very small and deterministic batch jobs.
Run a streaming computation as a series of very small, deterministic batch jobs.
Run a streaming computation as a series of very small, deterministic batch jobs.

- Chop up the live stream into batches of $X$ seconds.
Run a streaming computation as a series of very small, deterministic batch jobs.

- Chop up the live stream into batches of $X$ seconds.
- Spark treats each batch of data as RDDs and processes them using RDD operations.
Spark Streaming

- Run a streaming computation as a **series** of very small, **deterministic** batch jobs.

  - **Chop up** the live stream into batches of $X$ seconds.
  
  - Spark treats each batch of data as **RDDs** and processes them using **RDD operations**.
  
  - Finally, the processed results of the RDD operations are returned in batches.
Spark Streaming

- Run a streaming computation as a **series** of very **small**, **deterministic** **batch** jobs.

  - **Chop up** the live stream into batches of $X$ seconds.
  - Spark treats each batch of data as **RDDs** and processes them using **RDD operations**.
  - Finally, the processed results of the RDD operations are returned in **batches**.
  - **Discretized Stream Processing (DStream)**
DStream

- **DStream**: sequence of RDDs representing a stream of data.

- Any operation applied on a DStream translates to operations on the underlying RDDs.
DStream

- **DStream**: sequence of RDDs representing a stream of data.

- Any operation applied on a **DStream** translates to operations on the underlying RDDs.
StreamingContext

- **StreamingContext**: the main entry point of all Spark Streaming functionality.

- To **initialize** a Spark Streaming program, a StreamingContext **object** has to be created.

```scala
val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))
```
Source of Streaming

- **Two** categories of streaming sources.

- **Basic sources** directly available in the StreamingContext API, e.g., file systems, socket connections, ....

- **Advanced sources**, e.g., Kafka, Flume, Kinesis, Twitter, ....

```scala
ssc.socketTextStream("localhost", 9999)

TwitterUtils.createStream(ssc, None)
```
DStream Transformations

- **Transformations**: modify data from on DStream to a new DStream.
- **Standard RDD operations**, e.g., map, join, ...
- **DStream operations**, e.g., window operations
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))

val lines = ssc.socketTextStream("localhost", 9999)

val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

wordCounts.print()
Window Operations

- Apply transformations over a sliding window of data: window length and slide interval.

```
val ssc = new StreamingContext(conf, Seconds(1))
val lines = ssc.socketTextStream(IP, Port)
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))
val windowedWordCounts = pairs.reduceByKeyAndWindow(_ + _, Seconds(30), Seconds(10))
```
MapWithState Operation

- Maintains **state** while **continuously updating** it with new information.
- It requires the **checkpoint** directory.
- A new operation after **updateStateByKey**.

```scala
val ssc = new StreamingContext(conf, Seconds(1))
ssc.checkpoint(".")

val lines = ssc.socketTextStream(IP, Port)
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))

val stateWordCount = pairs.mapWithState(
    StateSpec.function(mappingFunc))

val mappingFunc = (word: String, one: Option[Int], state: State[Int]) => {
    val sum = one.getOrElse(0) + state.getOption.getOrElse(0)
    state.update(sum)
    (word, sum)
}
```
Transform Operation

- Allows arbitrary **RDD-to-RDD functions** to be applied on a DStream.
- Apply any **RDD operation** that is not exposed in the DStream API, e.g., joining every RDD in a DStream with another RDD.

```scala
// RDD containing spam information
val spamInfoRDD = ssc.sparkContext.newAPIHadoopRDD(...)

val cleanedDStream = wordCounts.transform(rdd => {
  // join data stream with spam information to do data cleaning
  rdd.join(spamInfoRDD).filter(...)
  ...
})
```
val words: DStream[String] = ... 

words.foreachRDD { rdd =>
  // Get the singleton instance of SQLContext
  val sqlContext = SQLContext.getOrCreate(rdd.sparkContext)
  import sqlContext.implicits._

  // Convert RDD[String] to DataFrame
  val wordsDataFrame = rdd.toDF("word")

  // Register as table
  wordsDataFrame.registerTempTable("words")

  // Do word count on DataFrame using SQL and print it
  val wordCountsDataFrame =
    sqlContext.sql("select word, count(*) as total from words group by word")
  wordCountsDataFrame.show()
}
GraphX
Introduction

- **Graphs** provide a **flexible abstraction** for describing relationships between **discrete objects**.

- Many problems can be **modeled by graphs** and solved with appropriate **graph algorithms**.
Can we use platforms like MapReduce or Spark, which are based on data-parallel model, for large-scale graph proceeding?
Graph-Parallel Processing

- Restricts the **types of computation**.
- New techniques to **partition and distribute graphs**.
- Exploit graph structure.
- Executes graph algorithms orders-of-magnitude faster than more general **data-parallel** systems.
Data-Parallel vs. Graph-Parallel Computation (1/3)

Data-Parallel

Table

Row
Row
Row
Row

Result

Graph-Parallel

Property Graph

Pregel
GraphLab
GIRAPH

Spark Streaming and GraphX
Graph-parallel computation: restricting the types of computation to achieve performance.
Graph-parallel computation: restricting the types of computation to achieve performance.

But, the same restrictions make it difficult and inefficient to express many stages in a typical graph-analytics pipeline.
Data-Parallel vs. Graph-Parallel Computation (3/3)

- Moving between table and graph views of the same physical data.

- Inefficient: extensive data movement and duplication across the network and file system.
GraphX

- Unifies data-parallel and graph-parallel systems.
- Tables and Graphs are composable views of the same physical data.
- Implemented on top of Spark.
GraphX vs. Data-Parallel/Graph-Parallel Systems

Live-Journal: 69 Million Edges

- Mahout/Hadoop: 1340 seconds
- Naïve Spark: 354 seconds
- Giraph: 207 seconds
- GraphX: 68 seconds
- GraphLab: 22 seconds

Runtime (in seconds, PageRank for 10 iterations)

Raw Wikipedia → Hyperlinks → PageRank → Top 20 Pages

Spark Preprocess → Compute → Spark Post.

- Spark: 1492 seconds
- Giraph + Spark: 605 seconds
- GraphX: 342 seconds
- GraphLab + Spark: 375 seconds

Total Runtime (in Seconds)
Represented using two Spark RDDs:
- Edge collection: VertexRDD
- Vertex collection: EdgeRDD

```
// VD: the type of the vertex attribute
// ED: the type of the edge attribute
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```
The triplet view logically joins the vertex and edge properties yielding an RDD[EdgeTriplet[VD, ED]].
Example Property Graph (1/3)

Property Graph

Vertex Table

<table>
<thead>
<tr>
<th>Id</th>
<th>Property (V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>(rxin, student)</td>
</tr>
<tr>
<td>7</td>
<td>(jgonzal, postdoc)</td>
</tr>
<tr>
<td>5</td>
<td>(franklin, professor)</td>
</tr>
<tr>
<td>2</td>
<td>(istoica, professor)</td>
</tr>
</tbody>
</table>

Edge Table

<table>
<thead>
<tr>
<th>SrcId</th>
<th>DstId</th>
<th>Property (E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>Collaborator</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>Advisor</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>Colleague</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>PI</td>
</tr>
</tbody>
</table>
val sc: SparkContext

// Create an RDD for the vertices
val users: VertexRDD[(String, String)] = sc.parallelize(
    Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
           (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))

// Create an RDD for edges
val relationships: EdgeRDD[String] = sc.parallelize(
    Array(Edge(3L, 7L, "collab"), Edge(5L, 3L, "advisor"),
         Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))

// Define a default user in case there are relationships with missing user
val defaultUser = ("John Doe", "Missing")

// Build the initial Graph
val userGraph: Graph[(String, String), String] =
    Graph(users, relationships, defaultUser)
// Constructed from above
val userGraph: Graph[(String, String), String]

// Count all users which are postdocs
userGraph.vertices.filter((id, (name, pos)) => pos == "postdoc").count

// Count all the edges where src > dst
userGraph.edges.filter(e => e.srcId > e.dstId).count

// Use the triplets view to create an RDD of facts
val facts: RDD[String] = graph.triplets.map(triplet =>
  triplet.srcAttr._1 + " is the " +
  triplet.attr + " of " + triplet.dstAttr._1)
Property Operators

```scala
def mapVertices[VD2](map: (VertexId, VD) => VD): Graph[VD2, ED]
def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
```

```scala
val newGraph = graph.mapVertices((id, attr) => mapUdf(id, attr))
```
def reverse: Graph[VD, ED]
def subgraph(epred: EdgeTriplet[VD, ED] => Boolean, 
     vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]

// Run Connected Components
val ccGraph = graph.connectedComponents() // No longer contains missing field

// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")

// Restrict the answer to the valid subgraph
val validCCGraph = ccGraph.mask(validGraph)
Join Operators

```python
def joinVertices[U](table: RDD[(VertexId, U)])(map: (VertexId, VD, U) => VD):
    Graph[VD, ED]
def outerJoinVertices[U, VD2](table: RDD[(VertexId, U)])
    (map: (VertexId, VD, Option[U]) => VD2):
    Graph[VD2, ED]
```

```scala
val outDegrees: VertexRDD[Int] = graph.outDegrees

val degreeGraph = graph.outerJoinVertices(outDegrees) {
    (id, oldAttr, outDegOpt) =>
        outDegOpt match {
            case Some(outDeg) => outDeg
            case None => 0 // No outDegree means zero outDegree
        }
    }
```
def aggregateMessages(Msg: ClassTag)(
    sendMsg: EdgeContext[VD, ED, Msg] => Unit, // map
    mergeMsg: (Msg, Msg) => Msg, // reduce
    tripletFields: TripletFields = TripletFields.All):
    VertexRDD[Msg]

val graph: Graph[Double, Int] = ...

val olderFollowers: VertexRDD[(Int, Double)] =
    graph.aggregateMessages[(Int, Double)](triplet =>
        {
            // Map Function
            if (triplet.srcAttr > triplet.dstAttr) {
                // Send message to destination vertex containing counter and age
                triplet.sendToDst(1, triplet.srcAttr)
            }
        }, // Reduce Function
        (a, b) => (a._1 + b._1, a._2 + b._2)
    )

val avgAgeOfOlderFollowers: VertexRDD[Double] = olderFollowers.mapValues(
    (id, value) => value match {case (count, totalAge) => totalAge / count})
Summary
Spark streaming
  • Mini-batch processing
  • DStream (sequence of RDDs)
  • Transformations, e.g., stateful, window, join, transform, ...

GraphX
  • Unifies graph-parallel and data-parallel models
  • Property graph (VertexRDD and EdgeRDD)
Questions?