Outline of Data Streaming Lessons

* Apache Spark fundamentals (RDD/DataFrame/Dataset)
* Actions/Transformations
* Spark Streaming/Structured Streaming
* Integration of Spark Streaming with Apache Kafka

**Lesson Overview**

In this lesson, we’ll go over the components of Apache Spark and focus on the ones that we will be using throughout the course. We’ll specifically look at

* Apache Spark Ecosystem
* Overview on Apache Spark’s building blocks (RDD/DataFrame/DataSet)
* Apache Spark Streaming and Structured Streaming
* Usage of Spark UI
* Concepts of Spark DAGs and Stages

**Slides for Each Lesson Available in Resources**

The slide deck for each lesson, seen in the videos, can be found in the Resources tab of the left sidebar of your classroom here.

**Glossary of Key Terms You Will Learn in this Lesson**

* **RDD** (Resilient Distributed Dataset) : The fundamental data structure of the Spark Core component. An immutable distributed collection of objects.
* **DataFrame** : A data structure of the Spark SQL component. A distributed collection of data organized into named columns.
* **Dataset** : A data structure of the Spark SQL component. A distributed collection of data organized into named columns **and also strongly typed**.
* **DAG (Directed Acyclic Graph)**: Each Spark job creates a DAG which consists of task stages to be performed on the clusters.
* **Logical plan** : A pipeline of operations that can be executed as one stage and does not require the data to be shuffled across the partitions — for example, map, filter, etc.
* **Physical plan** : The phase where the action is triggered, and the DAG Scheduler looks at lineage and comes up with the best execution plan with stages and tasks together, and executes the job into a set of tasks in parallel.
* **DAG Scheduler**: DAG scheduler converts a logical execution plan into physical plan.
* **Task** : A task is a unit of work that is sent to the executor.
* **Stage** : A collection of tasks.
* **State** : Intermediary and arbitrary information that needs to be maintained in streaming processing.
* **Lineage Graph**: A complete graph of all the parent RDDs of an RDD. RDD Lineage is built by applying transformations to the RDD.

## park Components

#### Core

Contains the basic functionality of Spark. Also home to the API that defines RDDs, which is Spark's main programming abstraction.

#### SQL

Package for working with structured data. It allows querying data via SQL as well as Apache Hive. It supports various sources of data, like Hive tables, Parquet, JSON, CSV, etc.

#### Streaming

Enables processing of live streams of data. Spark Streaming provides an API for manipulating data streams that are similar to Spark Core's RDD API.

#### MLlib

Provides multiple types of machine learning algorithms, like classification, regression, clustering, etc. This component will not be a focus of this course.

#### GraphX

Library for manipulating graphs and performing graph-parallel computations. This library is where you can find PageRank and triangle counting algorithms. This component will not be a focus of this course.

**Partitioning in Spark**

By default in Spark, a partition is created for each block of the file in HDFS (128MB is the default setting for Hadoop) if you are using HDFS as your file system. If you read a file into an RDD from AWS S3 or some other source, Spark uses 1 partition per 32MB of data. There are a few ways to bypass this default upon creation of an RDD, or reshuffling the RDD to resize the number of partitions, by using rdd.repartition(<the partition number you want to repartition to>). For example, rdd.repartition(10) should change the number of partitions to 10.

In local mode, Spark uses as many partitions as there are cores, so this will depend on your machine. You can override this by adding a configuration parameter spark-submit --conf spark.default.parallelism=<some number>.

So hypothetically, if you have a file of 200 MB and if you were to load this into an RDD, how many partitions will this RDD have? If this file is on HDFS, this will produce 2 partitions (each of them being 128MB). If the file is on AWS S3 or some other file system, it will produce 7 partitions.

**Hash Partitioning**

Hash partitioning in Spark is not different than the normal way of using a hash key in the data world to distribute data across partitions uniformly.

Usually this is defined by

partition = key.hashCode() % numPartitions

This mode of partitioning is used when you want to evenly distribute your data across partitions.

**Range Partitioning**

Range partitioning is another well-known partitioning method in the data world. Range partitioning divides each partition in a continuous but non-overlapping way.

Let's pretend there is a table called employees, and it has the following schema:

In reality, you'd want to use range partition over a timestamp, but this example gives you a rough idea of what range partitioning means.

You can use the partitionByRange() function to partition your data into some kind of group. Range partitioning in Spark ensures that every range is contained in a single partition. This becomes useful when you want to reduce shuffles across machines, for example when you know for sure all your parent RDDs need to stay in the same partition.

**DataFrames and Datasets - Key Points**

You can think of DataFrames as tables in a relational database, or dataframes in Python’s pandas library. DataFrames provide memory management and optimized execution plans.

**DataFrames**

DataFrames appeared in Spark Release 1.3.0. We already know that both Datasets and DataFrames are an organized collection of data in columns, but the biggest difference is that DataFrames do not provide type safety. DataFrames are similar to the tables in a relational database. Unlike RDDs, DataFrames and Datasets are part of the spark.sql library, which means you can create a temporary view of these tables and apply SQL queries.

DataFrames allow users to process a large amount of structured data by providing Schema. The Schema is another feature that is very similar to a relational database, indicating types of data that should be stored in the column (String, Timestamp, Double, Long, etc... these are available in spark.sql.types library), and also whether the column can be nullable or not. The aspect that is different from relational databases is that DataFrames and Datasets have no notion of primary/foreign keys - you as a developer define these as you create your DataFrame or Dataset.

**Datasets**

A Dataset is a core building block in SparkSQL that is strongly typed, unlike DataFrames, You can think of Datasets as an extension of the DataFrame API with type-safety. The Dataset API has been available since the release of Spark 1.6. Although Datasets and DataFrames are part of the Spark SQL Component, RDDs, Datasets, and DataFrames still share common features which are: immutability, resilience, and the capability of distributed computing in-memory.

A Dataset provides the features of an RDD and a DataFrame:

* The convenience of an RDD, as it is an extended library of a Spark DataFrame
* Performance optimization of a DataFrame using Catalyst
* Enforced type-safety

Datasets are not available in Python, only in Java and Scala. So we won’t be spending much time with Datasets in this course, since we focus on Python.

**DataFrames Demo**