**Lesson Overview**

We’ll be reviewing action and transformation functions in RDDs/DataFrames/Datasets, and how they affect DAGs and lineage graphs. We’ll also take a look at how to use query plans to best optimize your queries. Lastly, we’ll learn how to join and aggregate on streaming datasets. We’ll use built-in-sinks to view the results of these joins and aggregations.

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

* **Lazy evaluation**: An evaluation method for expressions in which expressions are not evaluated until their value is needed.
* **Action**: A type of function for RDDs/DataFrames/Datasets where the values are sent to the driver. A new RDD/DataFrame/Dataset is not formed.
* **Transformation**: A lazily evaluated function where a new RDD/DataFrame/Dataset is formed. There are narrow and wide types of transformations - narrow transformations occur within the same partition, whereas wide transformations may occur across all partitions.
* **Sink**: Place for streaming writes (output).

## Action and Transformation Functions in RDDs and DataFrames

There are two types of Apache Spark RDD operations - transformations and actions. Transformations produce a new RDD from an existing RDD. Actions trigger evaluation of expressions and send data back to the driver.

Transformations build an RDD lineage. For example, this code rdd.map().filter().join().repartition() builds a lineage of map -> filter -> join -> repartition. You can see this lineage through the .explain operation or through the Spark UI.

The resulting RDD from a transformation is always different from its parent RDD. It can be smaller or bigger or the same size (e.g. map).

Transformations also have two types of operations - narrow and wide. Narrow is map-like, and wide is reduce-like.

When the action is triggered, the computation is sent back to the driver. You may persist an RDD in memory using the persist method if there are many transformations and actions applied, so that Spark keeps the RDD around the cluster for faster access the next time.

## Lazy Evaluation

Before we dive further into actions and transformations we need to talk about lazy evaluation. Spark utilizes lazy evaluation to evaluate the expressions on RDD/DataFrame. It's also one of the key features in Scala, in which Spark is built.

**Lazy Evaluation - Key Points**

Lazy evaluation means an expression is not evaluated until a certain condition is met. In Spark, this is when an action triggers the DAG. Transformations are lazy and do not execute immediately. Spark adds them to a DAG of computation, and only when the driver requests some data (with an action function) does this DAG actually get executed.

These are some advantages of lazy evaluation:

* Users can organize their Apache Spark program into smaller operations. It reduces the number of passes on data by grouping operations.
* Saves resources by not executing every step. It saves the network trips between driver and cluster. This also saves time.
* here are a couple of cases I’ve thought of where you could put lazy evaluation to work:
* When you know your Spark job has many actions. This means that your Spark job will already have tons of network and machine shuffles due to existing action functions and this will materialize your intermediate datasets every time you execute action functions. Evidently, this will spike your GC (garbage collection) cost. In this case, you want to minimize network and machine shuffles using as much transformation functions as possible to reduce network and machine shuffles.
* Let’s say you have two tables - one table has the information about the 50 states in the US, like state in two letters (NY, CA, ..) and all the state’s zip codes, and the other table has all the residents in the US, with a column of which zip code they live in. Say you want to join two tables, and query only the people in California. In SQL, you would have to join them together, and then filter on “state = ‘CA’”. But in Spark, having lazy evaluation helps - since that one table is relatively small, Spark can switch the order of operations so that the smaller subset of dataframes is filtered first and then joined.

## Transformations

Transformations are one of the two types of Apache Spark RDD operations. Let’s look at a demonstration of how they work.

**Transformations - Key Points**

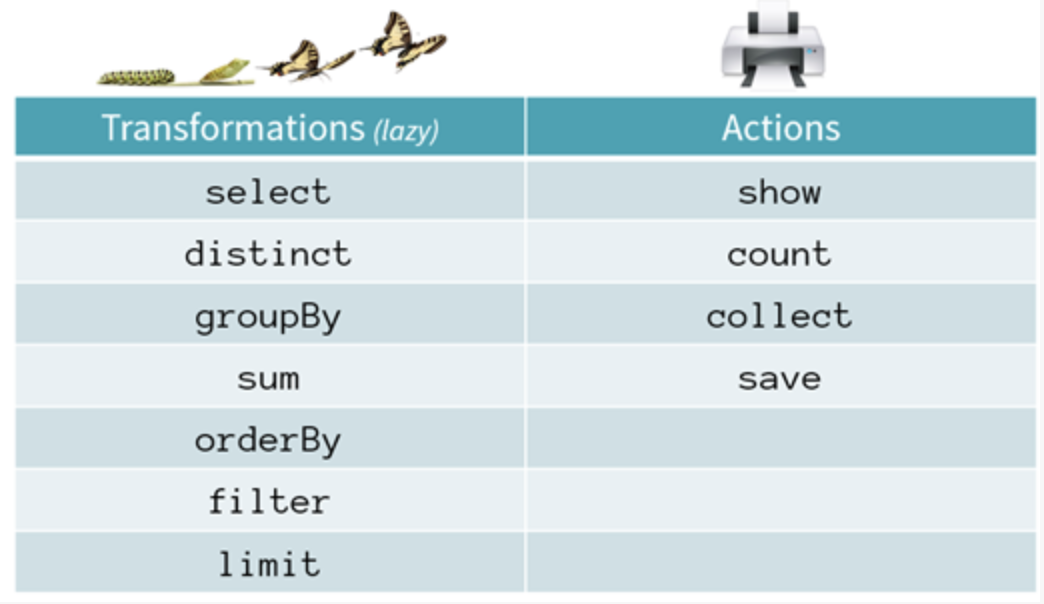
* Transformations in Spark build RDD lineage.
* Transformations are expressions that are lazily evaluated and can be chained together. Then, as we discussed in the previous lesson, Spark Catalyst will use these chains to define stages, based on its optimization algorithm.
* There are two types of transformations - wide and narrow. Narrow transformations have the parent RDDs in the same partition, whereas parent RDDs of the wide transformations may not be in the same partition.

**Actions - Key Points**

Actions are operations that produce non-RDD values. Actions send the values from the executors to be stored to the driver. Since the value is sent back to the driver, the final value must fit into the driver JVM. Unlike transformations, which are lazily evaluated, actions evaluate the expression on RDD.

A few action functions that are used often are:

* save()
* collect()
* count()
* reduce()
* getNumPartitions()
* aggregate()

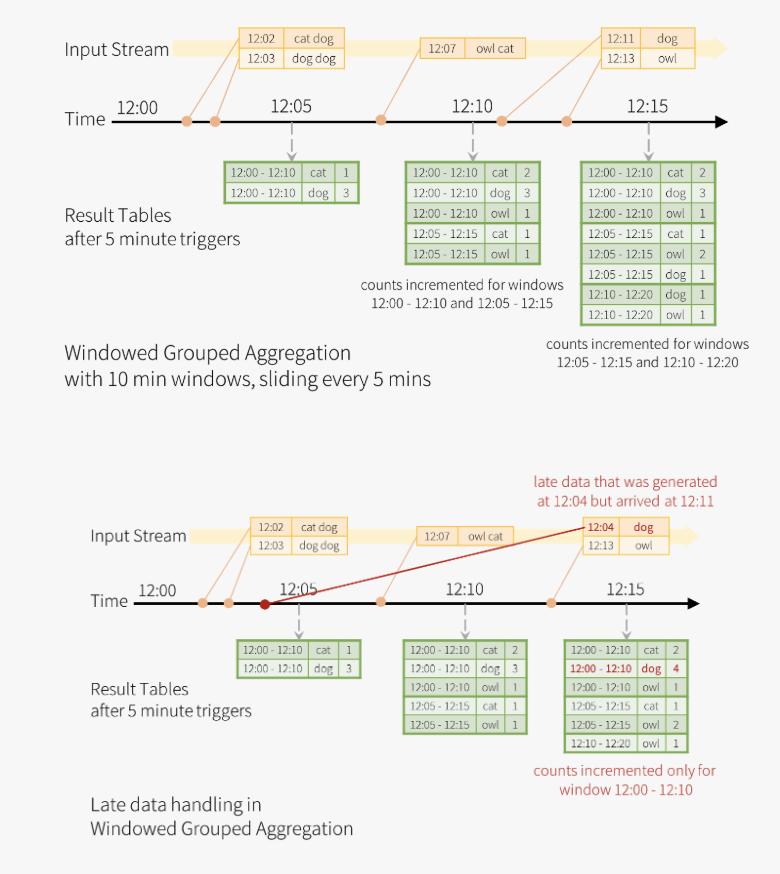
**[[](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/6cd0369e-32ae-42ab-aa26-550e66a4a986/concepts/feffd3c7-99e4-4d0e-aa19-869e340e48d8)](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/6cd0369e-32ae-42ab-aa26-550e66a4a986/concepts/feffd3c7-99e4-4d0e-aa19-869e340e48d8)**

**[Some Commonly Used Transformation and Action Functions](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/6cd0369e-32ae-42ab-aa26-550e66a4a986/concepts/feffd3c7-99e4-4d0e-aa19-869e340e48d8)**

\

**[[](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/6cd0369e-32ae-42ab-aa26-550e66a4a986/concepts/dd4bf956-8152-4dba-a637-c2b951e7d186)](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/6cd0369e-32ae-42ab-aa26-550e66a4a986/concepts/dd4bf956-8152-4dba-a637-c2b951e7d186)**

**[Join Types Supported for Stream and Static Joins (‘x’ means supported)](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/6cd0369e-32ae-42ab-aa26-550e66a4a986/concepts/dd4bf956-8152-4dba-a637-c2b951e7d186)**

**[[](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/6cd0369e-32ae-42ab-aa26-550e66a4a986/concepts/e7e8e44d-8dbc-4c1c-9d61-099179d9190a)](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/6cd0369e-32ae-42ab-aa26-550e66a4a986/concepts/e7e8e44d-8dbc-4c1c-9d61-099179d9190a)**

**[Example with Random Input Data Illustrating How Watermark Works](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/6cd0369e-32ae-42ab-aa26-550e66a4a986/concepts/e7e8e44d-8dbc-4c1c-9d61-099179d9190a)**

**Recap on JOINs, Aggregation, and Watermark**

JOINs, Aggregation, and Watermark are all commonly used Structured Streaming APIs.

* Types of JOINs
* In streaming, we have to think about static vs. streaming DataFrames. Inner JOINs can be easily applied to static and streaming DataFrames, regardless of which one is the left or right table. Outer JOINs are much more difficult because we have to think about late arriving data, and to join on late arriving data, the results will be significantly inconsistent.
* Aggregation vs. Stateful Aggregation
* You can think of aggregation as a plain sum() or count() in a SQL query. Stateful aggregation involves intervals, which means aggregation over certain intervals. We apply the concept of watermark to achieve stateful aggregation.
* Watermark
* Watermark addresses two main problems - dealing with late arriving data and stateful aggregation. Watermark decides which data can be dropped or included - this implies that the application doesn’t have to worry about deciding which data should be ingested in the pipeline, rather, watermark already acknowledges which data can be ingested. Also you can achieve stateful aggregation by setting watermark.

**JOIN Query Plans - Recap**

For any query plan, you can apply .explain() at the end of the query to visualize the query plan. Unfortunately, this is only available in the Scala version and we’ll have to run this in the Spark-shell to visualize the query plan.

* FileScan informs how the original file was scanned and loaded into the RDD. You can see the format of the file, the location of the file, and the column names (if appropriate).
* Spark uses BroadcastHashJoin when the size of the data is below BroadcastJoinThreshold. In the demo, we’re using small sized data and thus we see that the JOIN query is using BroadcastHashJoin.
* Project limits number of columns to those required by a union (or JOINs).

**Aggregate Query Plans - Recap**

Aggregate queries can also be viewed using the .explain() operator.

* Spark uses HashPartitioning on rather smaller datasets (which is what we’re using). Hash partitioning is nothing more than a partitioner using Java’s Object.hashcode.
* HashAggregate is used for aggregation with HashPartitioning. If a larger dataset is used, then TungstenAggregate is used to manage data outside the JVM to save some GC (garbage collection) overhead.

**What are Sinks?**

Spark Structured Streaming uses sinks to add the output of each batch to a destination.

Currently these are the following types of sinks supported:

* File sinks
* Kafka sinks
* Foreach and foreachBatch sinks
* Memory sinks
* Console sinks

**Modes for Using Built-in Output Sinks**

There are a few modes for writing to output sinks:

* Append - Only new rows are written to the output sink.
* Complete - All the rows are written to the output sink. This mode is used with aggregations.
* Update - Similar to complete, but only the updated rows will be written to the output sink.

## Explain in detail which output sink you'd use given a certain business model

Explain in detail which output sink you'd use given a certain business model: Suppose you have an application that requires streaming 24/7 in micro-batches, and the total count of events per day is about ~2 billion. How would you start setting applications and what kind of state management and sink that we learned could be applicable?

Billions of records are pretty decent amount of data volume. We can start thinking about how much data is coming in per second or per minute (so 2B / 2460 minutes of 2B / 2460\*60 seconds). There might be peak time of data ingestion depending on your business model, but we can assume the average rate. With these values calculated, now you may be able to gear your thoughts towards to think about what kind of state management and sink that we can use. Consider how many states we’ll need to manage - this will give you a better idea of when and how state management should occur.

**State Store Recap**

State management became important when we entered the streaming realm. You always want to save the metadata and data for the state for many purposes - like logging, metrics, metadata about your data, etc.

Storing the location of your state also became important - where would you want to save these data so that you can retrieve them later?

We’ll begin looking at strategies of using checkpointing and proper state storage next.

**State Management in DStream and Structured Streaming**

Since DStream is built on top of Spark RDD, we can operate transformation and action functions on DStream like we do on Spark RDD. Now that we're dealing with interval, another concept in transformation is introduced - stateless transformation and stateful transformation. We'll go over what is considered "state" in Spark Streaming in future lessons.

Stateless transformations are like map(), filter(), and reduceByKey(). As previously stated, each DStream is a continuous stream of RDDs. This type of transformation is applied to each RDD.

Stateful transformations track data across time. This means that the stateful transformation requires some shuffles between keys in key/value pair. The two main types of shuffles are windowed operations:

* updateStateByKey() - Used to track state across events for each key. updateStateByKey() iterates over all incoming batches and affects performance when dealing with a large dataset.
* mapWithState() - Only considers a single batch at a time and provides timeout mechanism.
* Further optional reading on stateful transformations: [**https://databricks.com/blog/2016/02/01/faster-stateful-stream-processing-in-apache-spark-streaming.html**](https://databricks.com/blog/2016/02/01/faster-stateful-stream-processing-in-apache-spark-streaming.html)

## Recovery of State

The accumulating result of the state should be stored in a fault-tolerant state store of your choice.

The purpose of state store is to provide a reliable place in your services so that the application (or the developer) can read the intermediary result of stateful aggregations. Even in the case of driver or worker failures, Spark is able to recover the processing state at the point right before the failure. The state stored is supported by HDFS compatible file system. To guarantee recoverability, Spark recovers the two most recent versions. If batch number 10 fails, then the batch number 9 and 8 are both recovered.

We can implement the state store by implementing org.apache.spark.sql.execution.streaming.state.StateStore properties.

## Sample Code for Checkpointing

We are providing the code here rather than in a workspace, because checkpoint requires a HDFS-compatible file system but our classroom workspaces do not currently offer HDFS-compatible file systems.

**from** pyspark.sql **import** SparkSession

**def** **checkpoint\_exercise**():

"""

note that this code will not run in the classroom workspace because we don't have HDFS-compatible file system

:return:

"""

spark = SparkSession.builder \

.master("local") \

.appName("Checkpoint Example") \

.getOrCreate()

df = spark.readStream \

.format("rate") \

.option("rowsPerSecond", 90000) \

.option("rampUpTime", 1) \

.load()

rate\_raw\_data = df.selectExpr("CAST(timestamp AS STRING)", "CAST(value AS string)")

stream\_query = rate\_raw\_data.writeStream \

.format("console") \

.queryName("Default") \

.option("checkpointLocation", "/tmp/checkpoint") \ *#this checkpoint location requires HDFS-like filesystem*

.start()

**if** \_\_name\_\_ == "\_\_main\_\_":

checkpoint\_exercise()

### Things to think about

You always want to use checkpointing. It depends on which data store you want to use to checkpoint (probably not console because it’s volatile) but something stable so that you can query your checkpoint later.

You always want to store the state.

**Configurations/Tuning Key Points**

There are a few ways to tune your Spark Structured Streaming application. But before that, go through your application and try to answer these questions.

* Study the memory available for your application. Do you have enough memory to process your Spark job? If not, consider vertical scaling. If you do have enough memory but limited resources, consider horizontal scaling.
* Study your query plans - do they make sense? Are you doing unnecessary shuffles/aggregations? Can you reduce your shuffles?
* What’s the throughput/latency of your data?
* How are you saving your data? Are you persisting your data to memory or to disk only, or to both memory and disk?
* Are you breaking your lineage anywhere?

When tuning your service you always want to optimize your code first.

## Lesson Summary

In this lesson, we learned the basic but most useful operations that can be applied to Spark RDDs and DataFrames. We covered in depth how lazy evaluation is built into Spark’s transformations and how it can affect the overall performance of the application.

We have also started to learn operations that could be applied to Spark Streaming and Structured Streaming. These will be useful in the next lesson, and at your job. In the next lesson we will continue to learn about Structured Streaming, and what kind of methods we can use to fine tune the application to satisfy your business needs.

## Further Optional Research

In case you are interested in learning more about any of these topics, here are some good resources for you!

* [**Action/Transformations**](https://medium.com/@aristo_alex/how-apache-sparks-transformations-and-action-works-ceb0d03b00d0)
* [**Stream-Stream JOINs**](https://databricks.com/blog/2018/03/13/introducing-stream-stream-joins-in-apache-spark-2-3.html)
* [**Spark journal**](https://cacm.acm.org/magazines/2016/11/209116-apache-spark/abstract)
* [**Spark SQL journal**](https://cs.stanford.edu/~matei/papers/2015/sigmod_spark_sql.pdf)
* [**Catalyst Optimizer**](https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html)