

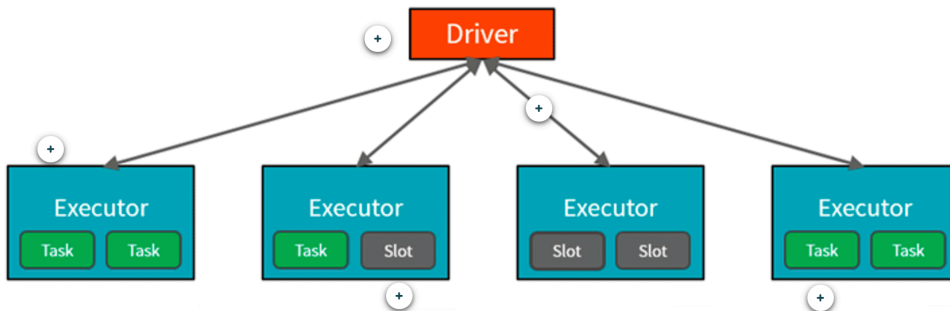
SPARK theory 2

Reference : <https://partner-academy.databricks.com/learn/course/127/play/1720:311/quick-reference-spark-architecture>

Match the terms on the left with the best description on the right.

✓	Driver	Responsible for assigning work that will be completed in parallel
✓	Executor	Reports the state of some computation back to a central system
✓	Spark Partition	A chunk of data (collection of rows) that sit on a single machine in a cluster

Spark uses clusters of machines to process big data by breaking a large task into smaller ones and distributing the work among several machines. Click on each marker in the diagram below to identify the components that Spark uses to coordinate work across a cluster of computers.



Transformations, Actions, and Execution

In the Databricks environment, you and your team will access and manipulate data in **tables**. Databricks tables are structured data that you and your team will use for analysis. Tables are equivalent to Apache Spark **DataFrames**. A **DataFrame** is an immutable, distributed collection of data organized into named columns. In concept, a **DataFrame** is equivalent to a table in a relational database, except that DataFrames carry important metadata that allows Spark to optimize queries. In this lesson, click on each of the concepts below to learn about how to perform operations on DataFrames in Spark.

Lazy Evaluation

Lazy Evaluation refers to the idea that Spark waits until the last moment to execute a series of operations. Instead of modifying the data immediately when you express some operation, you build up a plan of transformations that you will apply to your source data. That plan is executed when you call an action.

Lazy evaluation makes it easier to parallelize operations and allows Spark to apply various optimizations.

Transformations

Transformations are at the core of how you express your business logic in Spark. They are the instructions you use to modify a DataFrame to get the results that you want. We call them **lazy** because they will not be completed at the time you write and execute the code in a cell - they will only get executed once you have called an action.

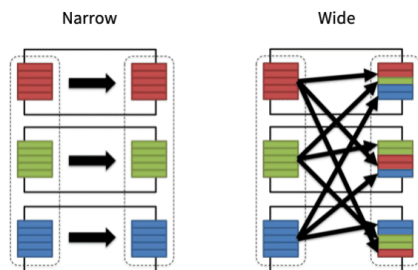
There are two types of transformations: Narrow and Wide.

For **narrow** transformations, the data required to compute the records in a single partition reside in at most one partition of the parent dataset.

For **wide** transformations, the data required to compute the records in a single partition may reside in many partitions of the parent dataset.

Remember, **spark partitions** are collections of rows that sit on physical machines in the cluster. Narrow transformations mean that work can be computed and reported back to the executor without changing the way data is partitioned over the system. Wide transformations require that data be redistributed over the system. This is called a shuffle.


Shuffles are triggered when data needs to move between executors.



Actions

Actions are statements that are computed AND executed when they are encountered in the developer's code. They are not postponed or wait for other code constructs. While transformations are lazy, actions are eager.

coalesce is narrow transformation - This operation results in a narrow dependency, e.g. if you go from 1000 partitions to 100 partitions, there will not be a shuffle, instead each of the 100 new partitions will claim 10 of the current partitions. If a larger number of partitions is requested

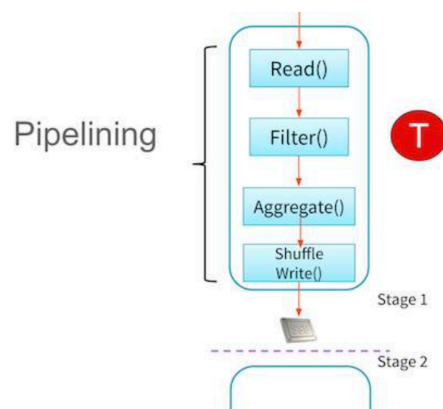


Transformations (<i>lazy</i>)	Actions
<code>select</code>	<code>show</code>
<code>distinct</code>	<code>count</code>
<code>groupBy</code>	<code>collect</code>
<code>sum</code>	<code>save</code>
<code>orderBy</code>	
<code>filter</code>	
<code>limit</code>	

Example commands that plan transformations or trigger actions.

Pipelining

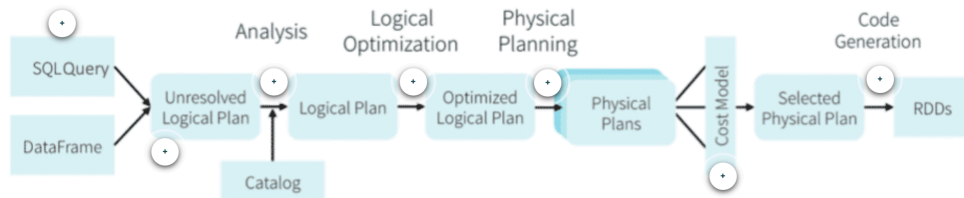
Lazy evaluation allows Spark to optimize the entire pipeline of computations as opposed to the individual pieces. This makes it exceptionally fast for certain types of computation because it can perform all relevant computations at once. Technically speaking, Spark pipelines this computation, which we can see in the image below. This means that certain computations can all be performed at once (like a map and a filter), rather than having to do one operation for all pieces of data and then the following operation. Additionally, Apache Spark can keep results in-memory as opposed to other frameworks that immediately write to disk after each task.



Source: A Gentle Introduction to Apache Spark on Databricks

Catalyst Optimizer

The Catalyst Optimizer is at the core of Spark SQL's power and speed. It automatically finds the most efficient plan for applying your transformations and actions. Click the markers on the image below to understand what's happening as your query travels through the optimization process.



Unresolved Logical Plan



This is your logical plan for how you want to transform the data. It is called unresolved at this stage because some elements, like column names and table names for example, have not been resolved. They might not exist.

Analysis



At this stage, column and table names are validated against the **catalog**. The catalog is how we refer to the metadata about the data stored in your tables. The unresolved logical plan becomes the logical plan.

Logical Optimization



This is where the first set of optimizations take place. Your query may be optimized by reordering the sequence of commands.

Physical Planning



The Catalyst Optimizer generates 1 or more physical plans that can be used to execute your query.

Each physical plan represents what the query engine will actually do after all optimizations have been applied.

Cost Model



Each physical plan is evaluated according to its cost model. The best performing model is selected. This gives us the selected physical plan.

Code Generation



The selected physical plan is compiled to Java bytecode and executed.

At which stage do the first set of optimizations take place?



Analysis



Logical Optimization



Physical Planning



Code Generation



Correct

TAKE AGAIN



Caching

You are probably aware that moving data around is expensive, both in time and money. In practice, your work will be connected to a central data store and the queries you run will pull data from there. In general, each time you run a query, you're going all the way back to the original store to get the data. All of this overhead can be avoided by **caching**.

One of the significant parts of Apache Spark is its ability to store things in memory during computation. This is a neat trick that you can use as a way to speed up access to commonly queried tables or pieces of data. This is also great for iterative algorithms that work over and over again on the same data. While many see this as a panacea for all speed issues, think of it much more like a tool that you can use. Other important concepts like data partitioning, clustering and bucketing can end up having a much greater effect on the execution of your job than caching. However, remember these are all tools in your tool kit!

In applications that reuse the same datasets over and over, one of the most useful optimizations is **caching**. **Caching** will place a `DataFrame` or table into temporary storage across the executors in your cluster and make subsequent reads faster.

The use case for caching is simple: as you work with data in Spark, you will often want to reuse a certain dataset. It is important to be careful how you use caching, because it is an expensive operation itself. If you're only using a dataset once, for example, the cost of pulling and caching it is greater than that of the original data pull.

Once data is cached, the Catalyst optimizer will only reach back to the location where the data was cached.

Congratulations! You have completed the "Quick Reference: Spark Architecture" course.

By now, you should be comfortable:

- Describing basic Spark architecture
 - Remember that there is a driver that sends tasks to multiple executors. The secret to Spark's performance is parallelism; the ability assign work to multiple virtual machines.
- Explaining lazy evaluation
 - Recall that Spark does not execute operations as soon as they are read. Instead, it builds up a plan for data transformations that will be applied to your source data. That plan is executed only when you call an action.
- Identifying events for each stage in the optimization process
 - The Catalyst Optimizer is designed to find the most efficient plan for applying the transformations and actions you called for in your code.

A Spark application with dynamic allocation enabled requests more executors when it has pending tasks waiting to be scheduled. This condition necessarily implies that the existing set of executors is insufficient to simultaneously saturate all tasks that have been submitted but not yet finished.