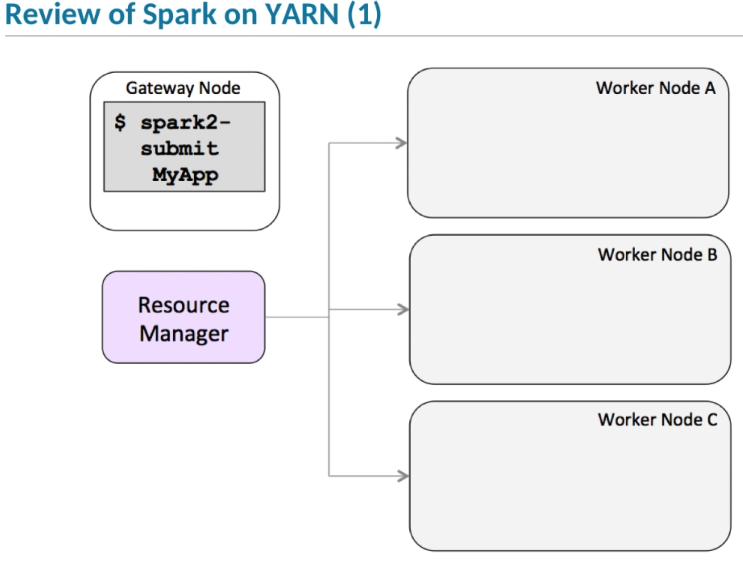
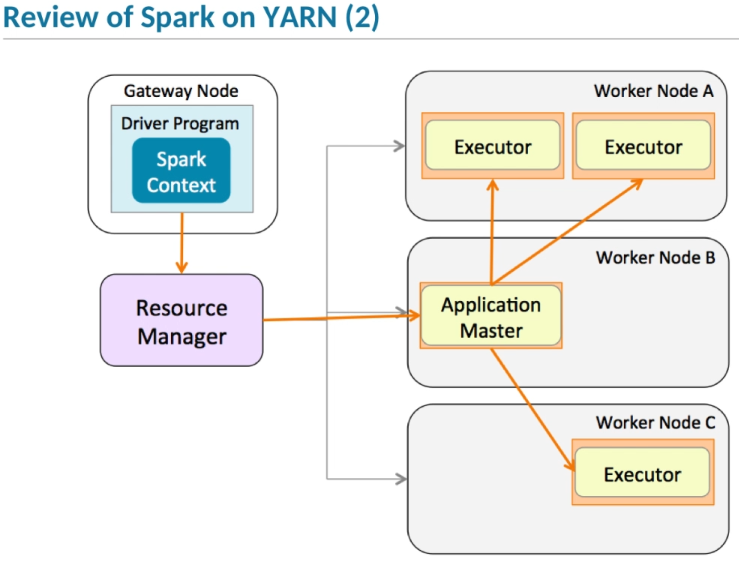
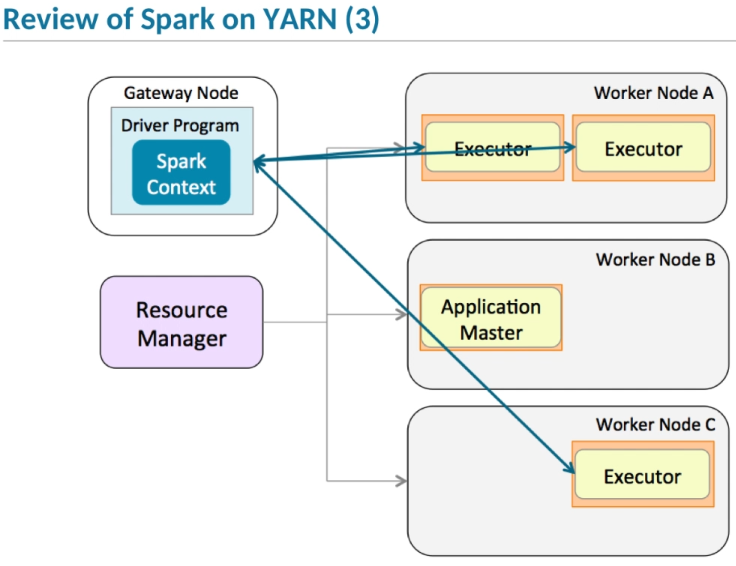
Apache Spark in a Cluster

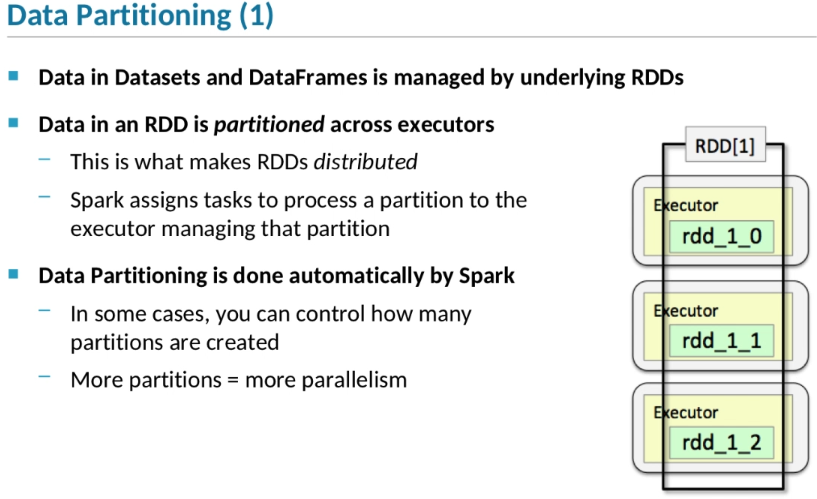


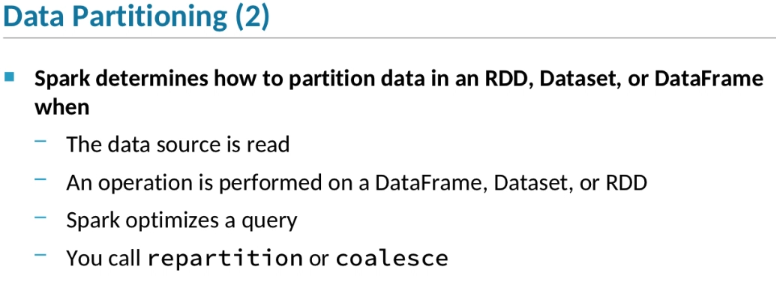


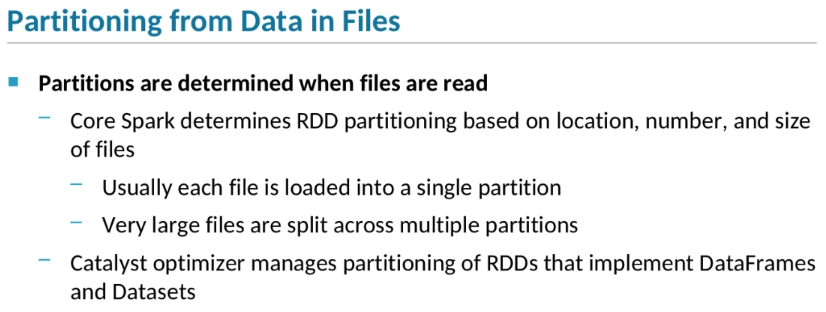


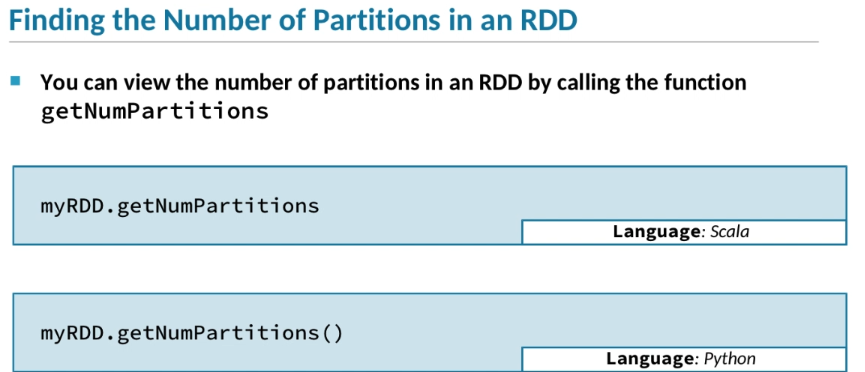
1. One of the great things about Spark is that we can treat DataFrames, Datasets, and RDDs as single monolithic collections of data.
2. We can operate on data without having to know the details of where or how that data is stored.
3. But to get the most from Spark, and to write the most efficient code,
4. you need to understand the basics of how data is distributed and processed by executors on a cluster.
5. Let’s start by briefly reviewing how Spark runs applications on a YARN cluster.
6. If you run in client mode, which is the default, the driver runs on the client machine—typically a cluster gateway node, as shown in this diagram.
7. If you run in cluster mode, the driver runs as part of the application master process on a cluster worker node.
8. **In either case, the startup script connects to the cluster's resource manager to request containers in which to run the application’s application master.**
9. The application master then requests more containers, and starts executor JVMs in those containers.
10. After the executors start, the Spark driver distributes tasks to run on them.
11. After the tasks complete, the executors either return a value to the driver or save the distributed data.
12. This architecture is the basis for running a distributed application.

Partitions



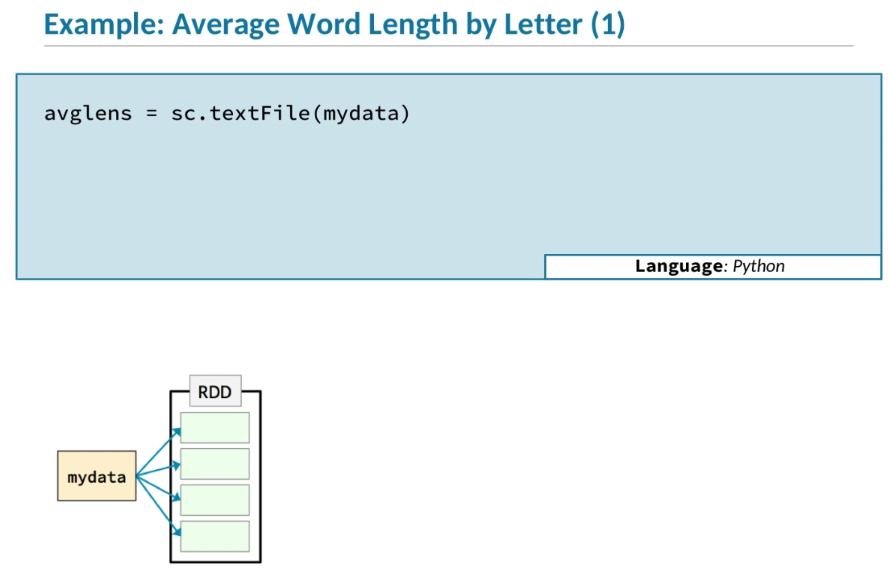


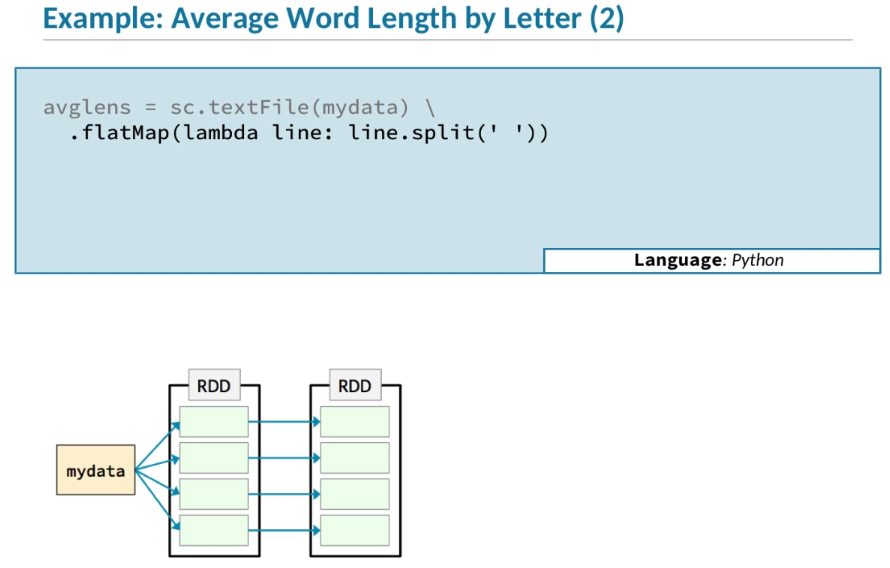


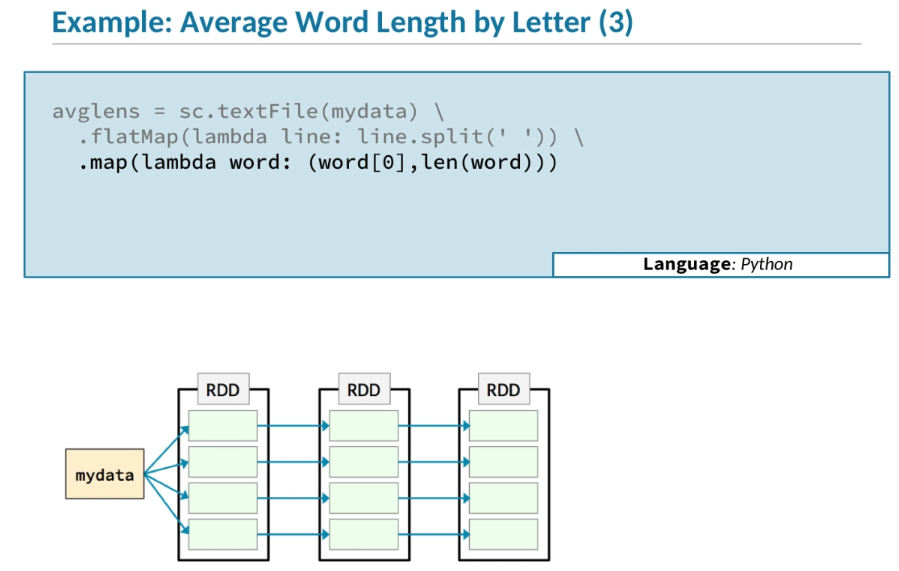


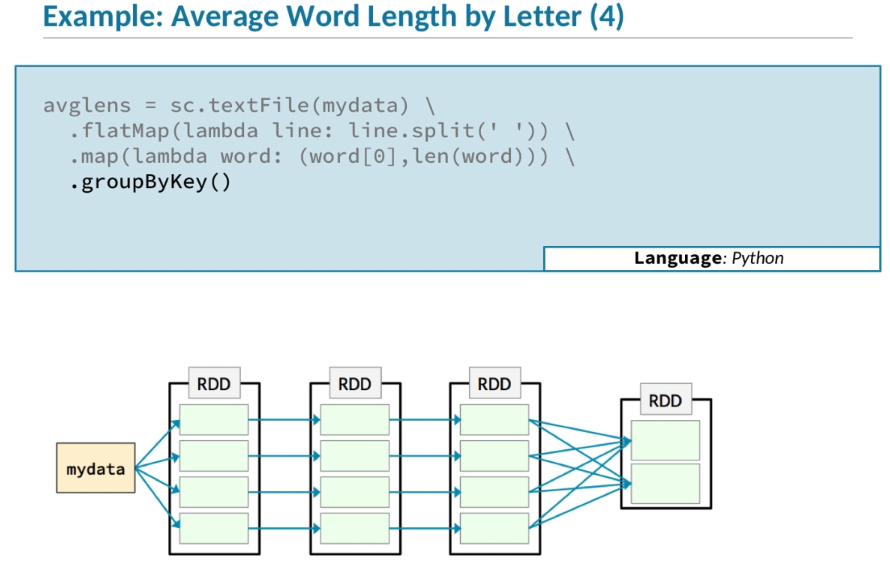
1. One of the most important aspects of Spark applications is that they can distribute their workloads across multiple worker nodes in a cluster,
2. so that application tasks can run in parallel for better performance.
3. RDDs form the basis for application distribution, whether you are working with RDDs directly or with DataFrames or Datasets.
4. **Recall that RDD stands for Resilient Distributed Dataset.**
5. RDDs are distributed by *partitioning*—or splitting up—the data across the application's executors.
6. The point of data partitioning is that tasks that operate on each partition can run in parallel.
7. Data partitioning is done automatically by Spark.
8. This allows developers to work with distributed data and tasks without having to explicitly handle the distribution themselves.
9. In some cases it is possible to manually control how many partitions are created.
10. If you have more partitions, more tasks will be able to run in parallel, which can improve performance.
11. There are a number of points when Spark will automatically partition data in an RDD.
12. Spark automatically partitions an RDD’s data when the RDD is created,
13. whether you create it explicitly or Spark automatically creates it when executing a Dataset or DataFrame query.
14. For instance, whenever Spark reads a set of data from a data source, it will automatically partition the data loaded from that source.
15. Spark also partitions data when a new RDD is created as a result of executing a DataFrame, Dataset, or RDD transformation.
16. When you perform a query on a DataFrame or Dataset, Catalyst will determine the best way to partition the data in order to optimize the query.
17. You can also call repartition or coalesce to manually create a new RDD with the specified number of partitions.
18. Let’s consider how Spark handles partitioning data that’s read from files.
19. When you create a file-based RDD directly using the core Spark API,
20. Spark determines the number and location of the partitions based on the number, location, and size of the files.
21. Typically, each file is loaded into its own partition.
22. Large files may be split across multiple partitions.
23. When you read data into a DataFrame or Dataset, the Catalyst optimizer determines how to partition the underlying RDDs.
24. Spark doesn’t determine where a partition is located—that is, which executor is processing that partition—until a query is actually executed.
25. When the query is executed, Spark will assign a partition to an executor running on the same machine where the file or file block resides, if possible.
26. You can see how many partitions an RDD has by calling getNumPartitions, as shown in these code snippets.
27. This doesn't work with Datasets or DataFrames because Catalyst doesn't determine the number of partitions until the query executes.

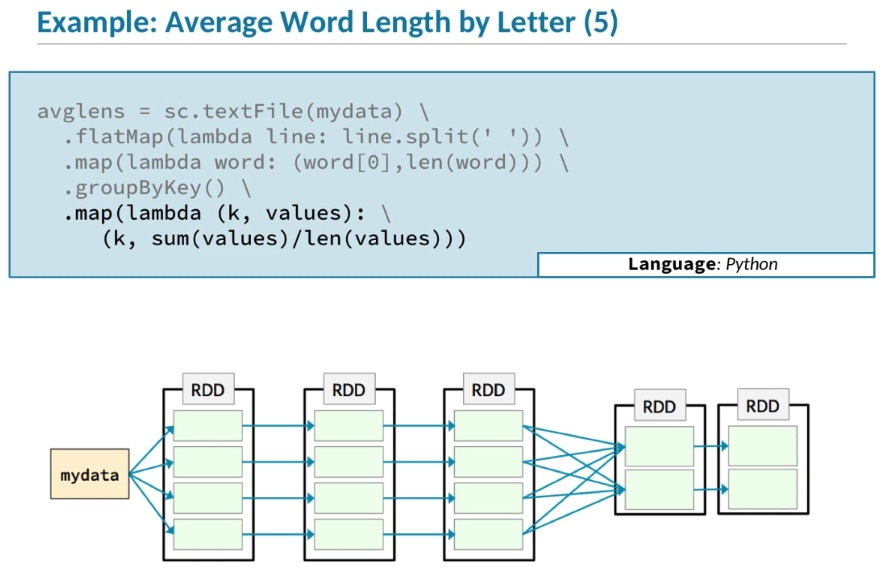
Partitioning in Queries





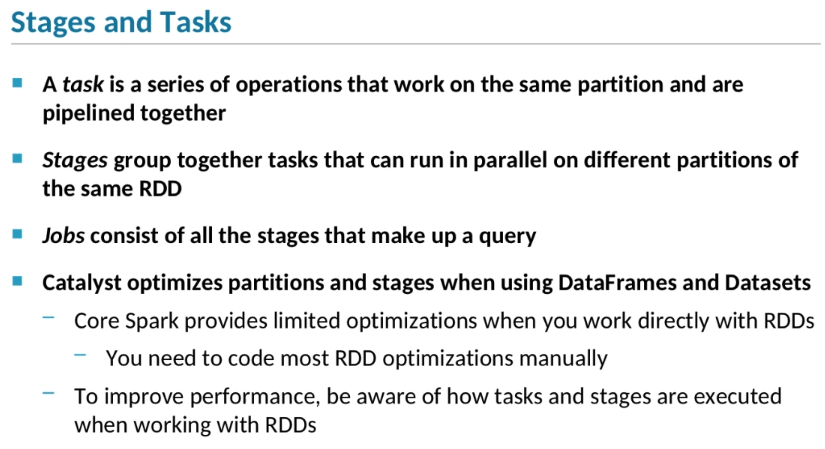


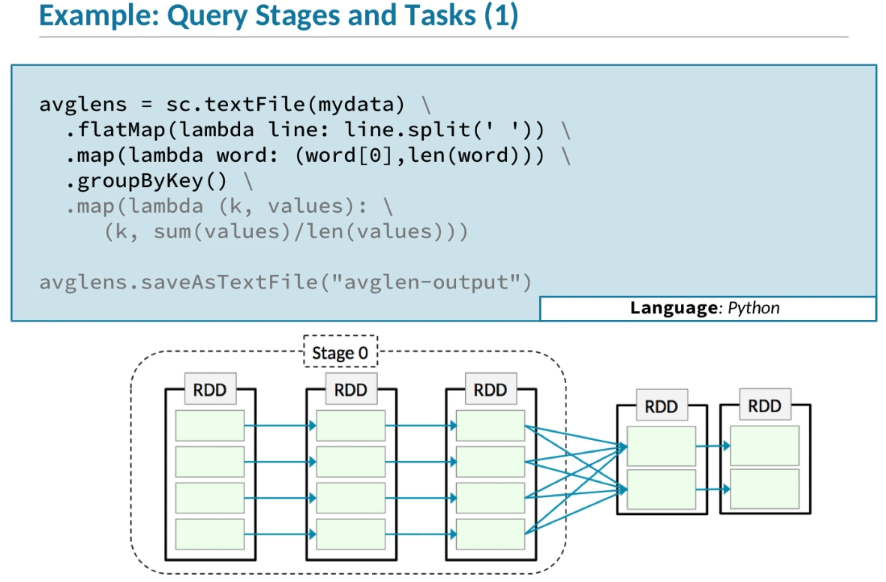


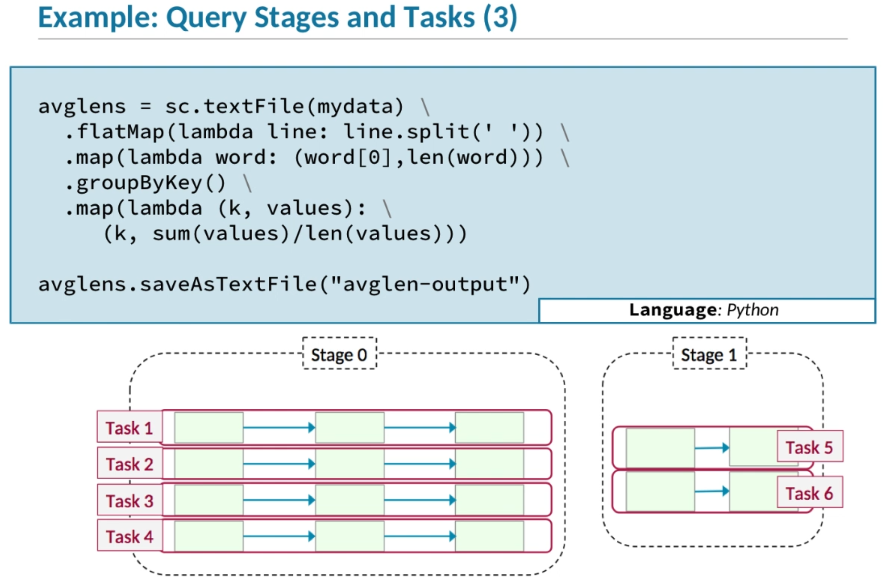


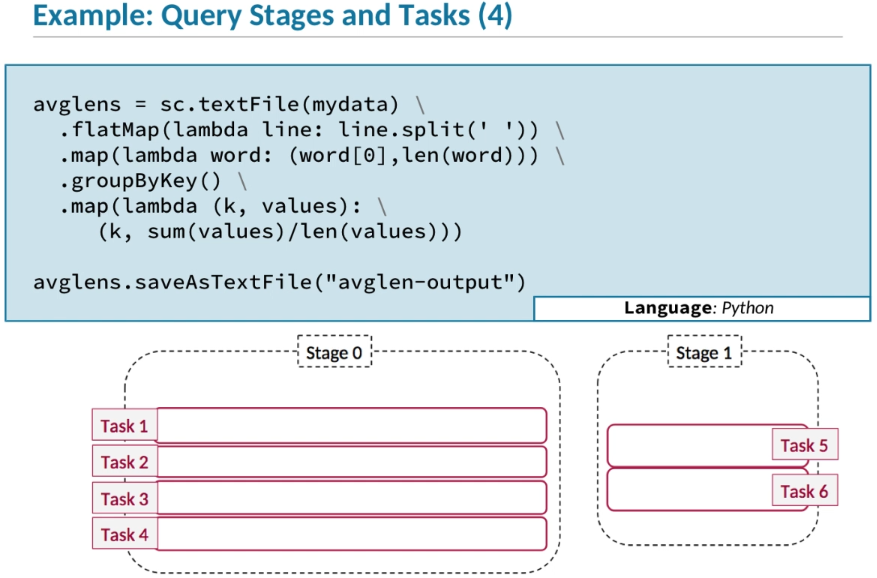
1. Let’s look at an example to see how different operations work with partitions.
2. This example analyzes a data set and calculates the average length of words beginning with each letter—a, b, c, and so forth.
3. **But what’s really important in this example isn’t the analysis we are doing, it’s the sequence of operations we’ll be using.**
4. We start by calling sc.textFile, which will load the file in.
5. In this example, the file is a large file that's distributed into four HDFS blocks, so it’s loaded into four partitions in the RDD.
6. We then do a flatMap, in which we split each line in the file up by spaces.
7. That will create a new RDD which also has four partitions.
8. Each of those partitions will be on the same executor as the original partition.
9. Then we do another map,
10. which creates a pair RDD in which the key is the initial character of the word and the value is the length of that word.
11. Again, that new RDD has four partitions.
12. Each partition is going to be assigned to the same executor as the previous partition.
13. Then we do a groupByKey.
14. Unlike map and flatMap, the groupByKey operation does not preserve partitioning.
15. In other words, it creates a new RDD with different partitioning than the RDD it started with.
16. That’s because groupByKey needs to group all the data in all the partitions,
17. so it has to shuffle the data from all the partitions in the base RDD into the new RDD it creates.
18. In this example, we are showing that the new RDD will have two partitions, but that actually depends on a number of factors.
19. For now let's assume that the new RDD has two partitions.
20. The final step in this query is another mapoperation that calculates the averages for each key—that is, each letter that starts a word.
21. The map preserves partitioning, so the new RDD will have two partitions, just like its parent.

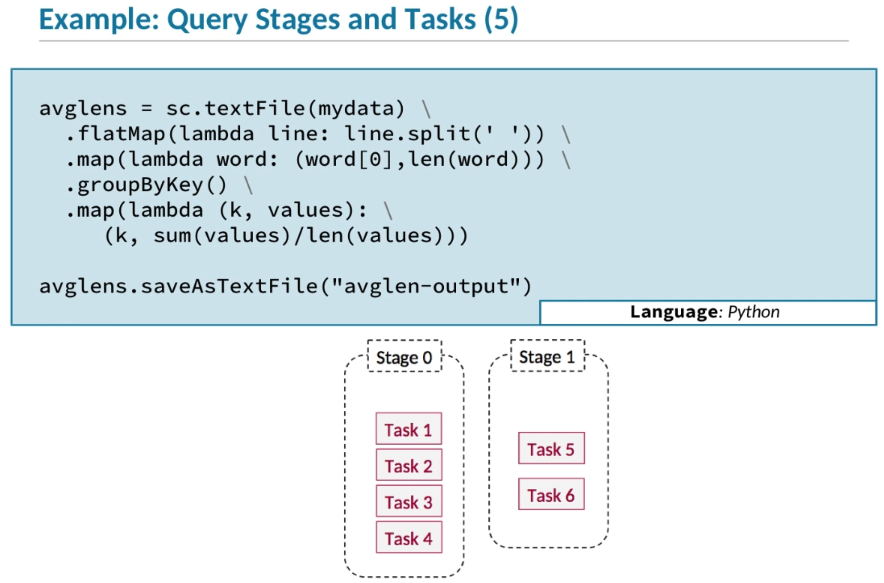
Stages and Tasks

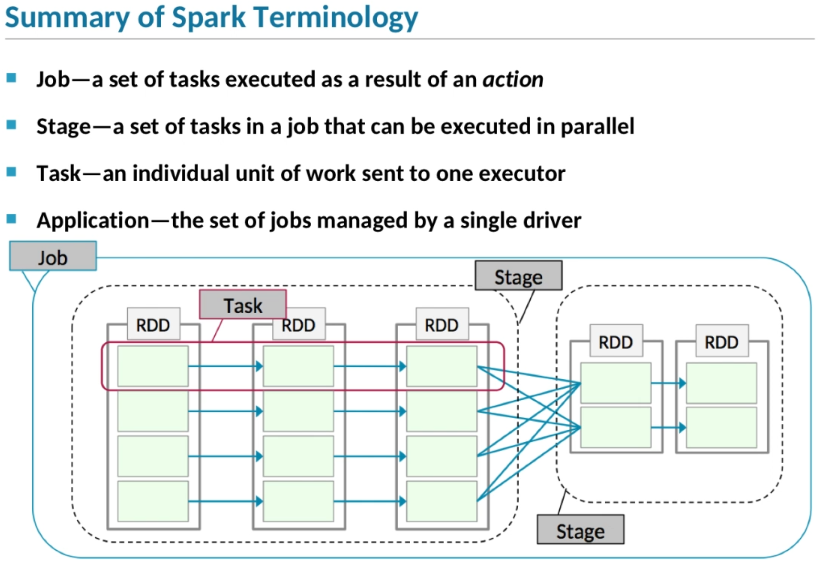






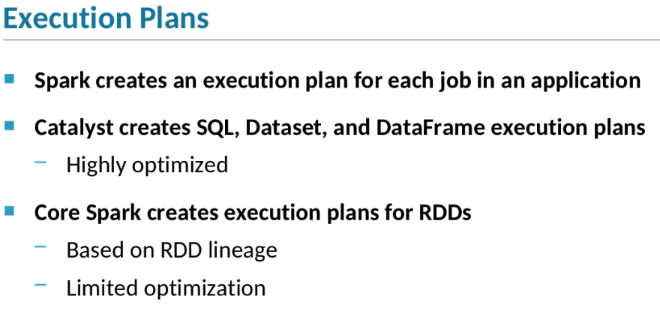


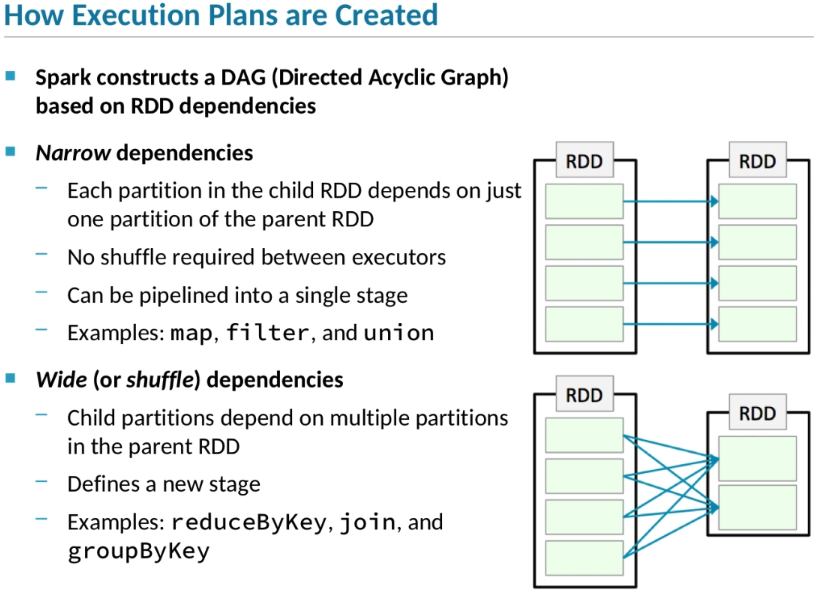


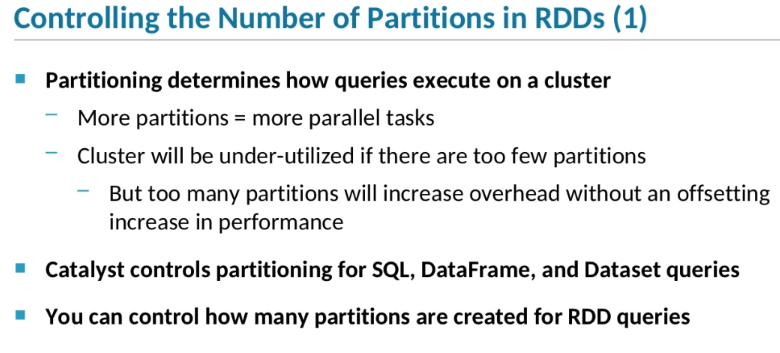


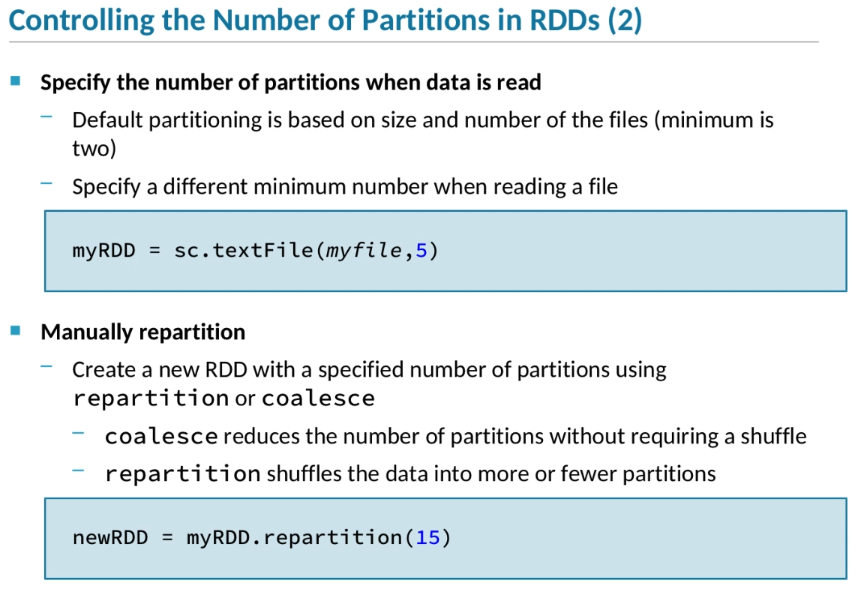
1. **The key to understanding how Spark operations execute on a cluster is to understand tasks, stages, and jobs, and how they relate to data partitions.**
2. Let’s start by defining what each term means.
3. A *task* is a series of operations that work on the same partition.
4. Operations in a task can be pipelined together.
5. A group of tasks that operate on the same sequence of RDDs is a *stage*.
6. A *job* is made up of all the stages in a query—that is, a series of transformations completed by an action.
7. For DataFrame and Dataset queries, Catalyst optimizes the job’s partitions, tasks, and stages.
8. When you create RDDs directly, core Spark might perform a few limited optimizations,
9. but in order to maximize performance, you'll need to optimize queries manually based on how stages execute.
10. The easiest way to understand how jobs are divided into stages is to look at an example.
11. Let’s start with an example query that calculates the average length of all the words beginning with each letter—a, b, c, and so on.
12. The calculation is performed using a series of transformations: flatMap, map, groupByKey, and another map.
13. The query executes when the final action—saveAsTextFile—is called.
14. Remember that Spark supports lazy execution,
15. so in fact nothing will happen until we perform an action on the final RDD in the sequence.
16. If you look at the diagram, you can see that the first three transformations all occur in the same stage, which we call Stage 0.
17. In each of those Stage 0 transformations—from the initial RDD to the second, from the second to the third—
18. the data in each partition of the new RDD is created from the data in just a single partition in the base RDD.
19. So these transformations can be grouped together into the first stage.
20. The final transformation in Stage 0—groupByKey—
21. coalesces the data from multiple partitions together to create a new RDD which may have a different number of partitions.
22. That’s when the first stage ends and the second stage—called Stage 1—begins.
23. Our second stage then has two transformations—groupByKey and the final map—followed by the [ action to trigger execution of the query.
24. Operations within a single stage are pipelined together.
25. Spark processes all the transformations in a pipeline for the first element of the RDD,
26. then all the transformations for the second element, and so on.
27. Together, the operations that are done within a stage on a particular partition are treated as a single task.
28. In other words, Stage 0 has four tasks, one for each of its four partitions.
29. Stage 1—the second stage—has two tasks, one for each of its two partitions.
30. To sum up, here’s how Spark will run this application.
31. It will have two stages.
32. The first stage will have four tasks, one for each of its four partitions,
33. and the second stage will have two tasks, one for each of its two partitions.
34. Here’s a summary of the Spark terminology related to parallel execution.
35. A job is a set of tasks which are executed as the result of an action.
36. A stage is a set of tasks that can be executed in parallel within that job.
37. A task is the individual unit of work which is sent to one executor.
38. A Spark application runs a single driver managing one or more jobs.

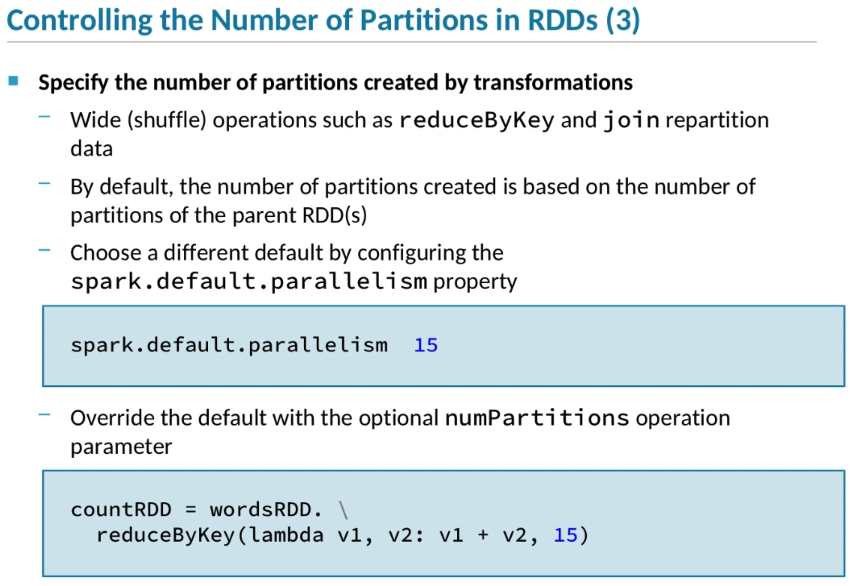
Job Execution Planning

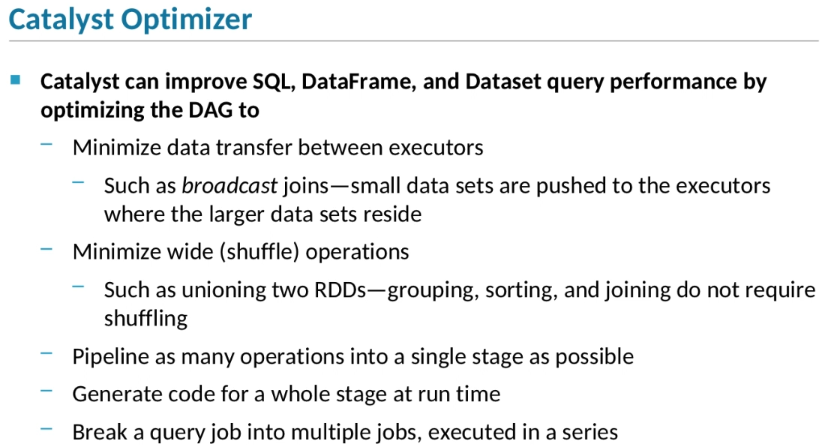


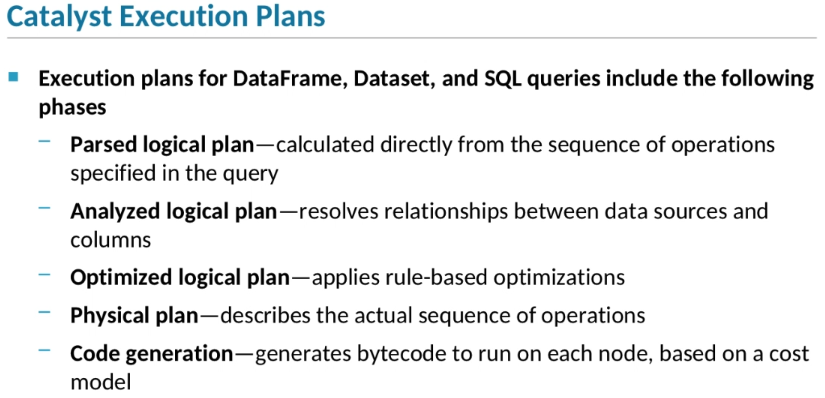






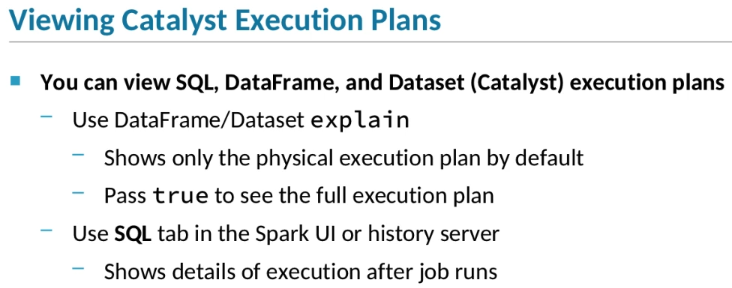


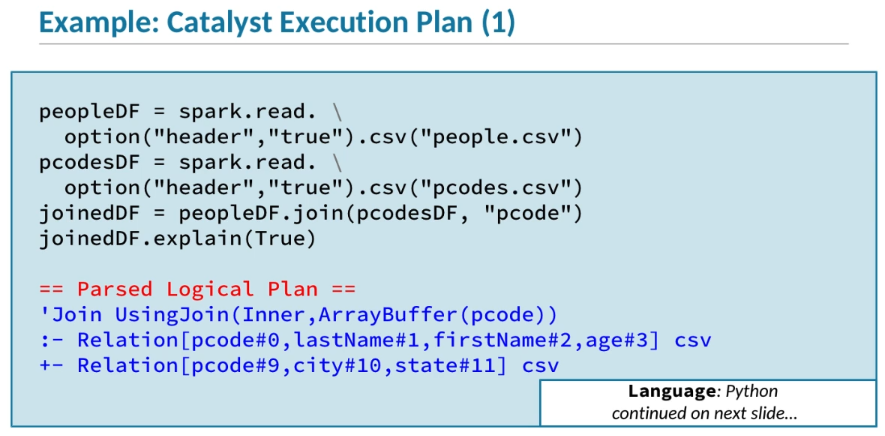


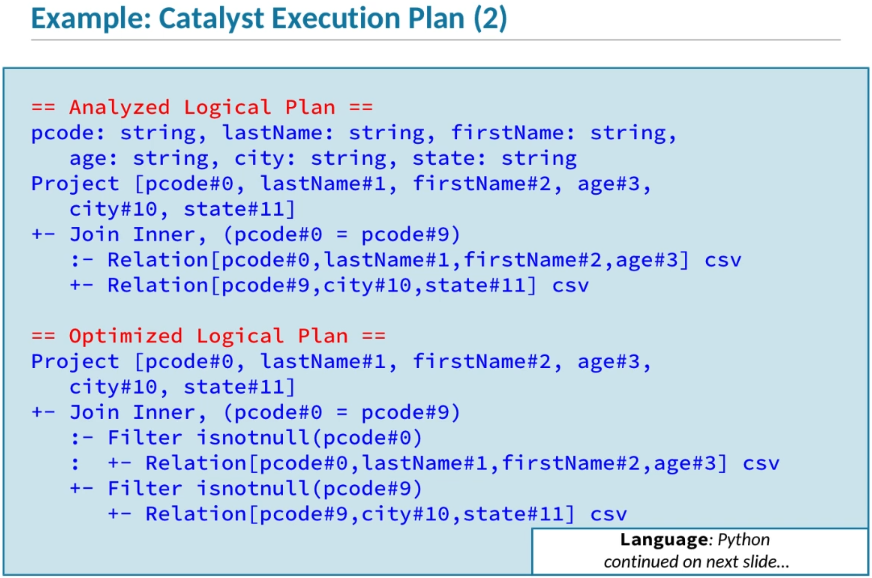


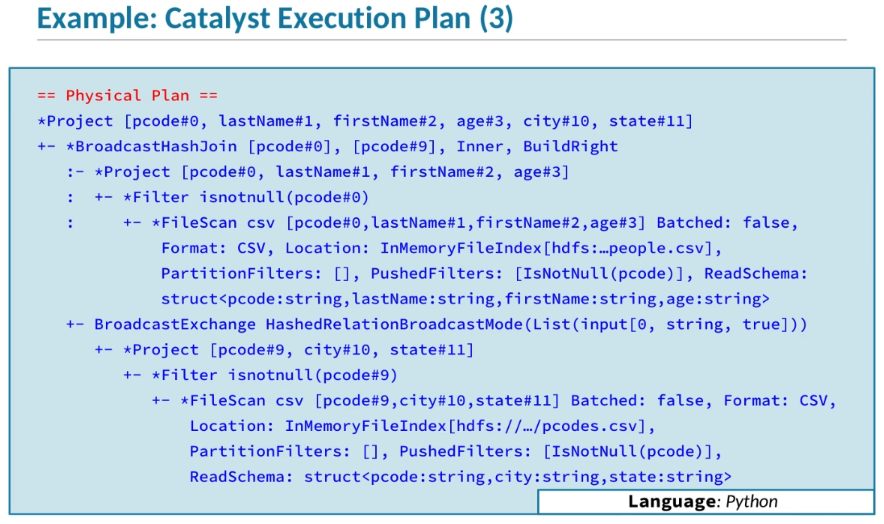
1. [Start of transcript. Skip to the end.](https://ondemand.cloudera.com/courses/course-v1:Cloudera+DevSH+180515/courseware/56ee904bcd91420ea0e1b73732758e0f/08e77c94fce040759ea7ca5bafdb2b8f/?child=first#transcript-end-85902fc1d029423ead01159beb67a5c7)
2. When Spark executes a job—that is, a single query—it follows an *execution plan*.
3. The execution plan for a SQL, Dataset, or DataFrame query is created by Catalyst and is highly optimized.
4. **Execution plans for core Spark queries are based on the lineage of the final RDD.**
5. Spark may be able to apply some limited optimizations to the execution plan.
6. All execution plans define a Directed Acyclic Graph, or *DAG*, that reflects the dependencies between a job’s stages.
7. Dependencies between stages are defined by dependencies between the RDDs created when a stage executes.
8. *Narrow* dependencies result from transformations in which each partition of the new RDD depends on a single partition of the base RDD.
9. In other words, no data needs to be shuffled between partitions on different executors.
10. A single stage consists of a series of transformations that create RDDs with narrow dependencies.
11. These include transformations such as map, filter, and union.
12. A *wide* dependency, sometimes called a shuffle dependency,
13. is one where the partitions in the child RDD contain data from multiple partitions in the parent RDD.
14. A wide dependency defines the end of one stage and the start of a new one.
15. Operations like reduceByKey, join, and groupByKeyresult in wide dependencies.
16. How the data in an RDD is partitioned determines how Spark distributes the tasks in a job to executors.
17. The more partitions you have, the more tasks can execute in parallel, up to the number of executors in the application.
18. If the number of tasks running in parallel is much less than the processing and memory capacity of a cluster, the cluster will be under-utilized.
19. On the other hand, if there’s not much data in each partition,
20. the overhead of creating and tracking many partitions may outweigh the performance advantage you get from parallelization.
21. Catalyst takes into account the amount of data and the capacity of the cluster when constructing a job’s execution plan.
22. If you manually create core Spark RDDs, you may need to adjust how many partitions Spark creates in order to get optimal performance.
23. By default, when Spark creates an RDD by reading one or more files, it assigns one partition for each block in the file.
24. The default minimum number of partitions is two, assuming your application is running with at least two executors or threads.
25. You can override that default by passing an optional parameter to the function that reads the files to specify a different number of partitions.
26. In this example, we set the minimum to 5.
27. If we read files with a total of less than five blocks, some of the blocks will be split into multiple partitions.
28. You can also create a new RDD with more or fewer partitions than the original by using coalesce or repartition.
29. Use coalesce to decrease the number of partitions without requiring a full data shuffle between executors.
30. This is often useful after a filter operation that eliminates a lot of data, resulting in too-small partitions.
31. The repartition function can increase or decrease the number of partitions, but shuffles the data when doing so, which can be expensive.
32. Wide or shuffle operations such as reduceByKeyand join repartition the data in the resulting RDD.
33. By default, the operation will repartition the new RDD based on the number of RDDs in the parent RDDs.
34. You can change the default behavior by setting the spark.default.parallelism property.
35. New RDDs created by wide operations will contain the number of partitions specified.
36. The first example on this slide shows how to configure the spark.default.parallelism property in a properties file,
37. but you could also configure the value on the command line when you start Spark.
38. You can override the default number of partitions when you execute a transformation by passing the number of partitions you want as an additional parameter.
39. The second example shows how you would call reduceByKey,
40. specifying that the result RDD should contain 15partitions.
41. Catalyst creates highly optimized execution plans for Dataset, DataFrame, and SQL queries.
42. One way it optimizes queries is by minimizing how much data needs to be moved between executors.
43. For instance, when the query joins two sets of data and one of the sets is very small, it will broadcast the smaller set—
44. that is, it will copy the smaller set to each of the executors, so that the first stage of the join operation can be performed on data local to each executor.
45. Another way it optimizes queries is to minimize how many shuffle operations the query does.
46. For instance, if two sets of data can be unioned into one, wide operations can work with data in a single partition instead of multiple partitions.
47. Catalyst will also reorder operations optimally to pipeline as many operations into a single stage as possible.
48. A key optimization strategy Catalyst uses is generating code for a whole stage at runtime and sending the compiled byte code to each executor to run.
49. Catalyst will sometimes even break a single job into a series of jobs for better performance.
50. When the Catalyst optimizer creates a query execution plan, the plan goes through a series of phases before it’s executed.
51. In the first stage, Catalyst parses the sequence of operations into a logical plan that represents the unoptimized query that the developer wrote.
52. In the next phase, Catalyst analyzes the parsed logical plan to resolve relationships between the data sources, tables, and columns referenced in the query.
53. This is the stage where partial column references are resolved into full references relative to a DataFrame.
54. The next phase optimizes the logical plan by applying a set of rules to the query operations.
55. For instance, Catalyst might detect a whereoperation followed by a select operation
56. and reorder those so that only the data in the selected columns gets evaluated by the filter.
57. After the logical plan is optimized, Catalyst generates a physical plan that specifies the exact sequence of data sources to be loaded and transformed.
58. The physical plan is defined when the DataFrame or Dataset is first created,
59. but the final phase of the execution plan—code generation—isn't defined until the query actually executes as a result of an action.
60. In that phase, Catalyst generates byte code for each stage to create the necessary RDDs and perform the data transformations.

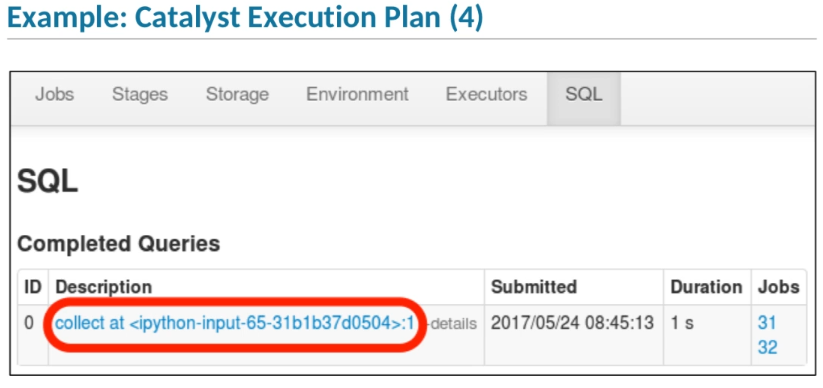
Example: Catalyst Execution Plans

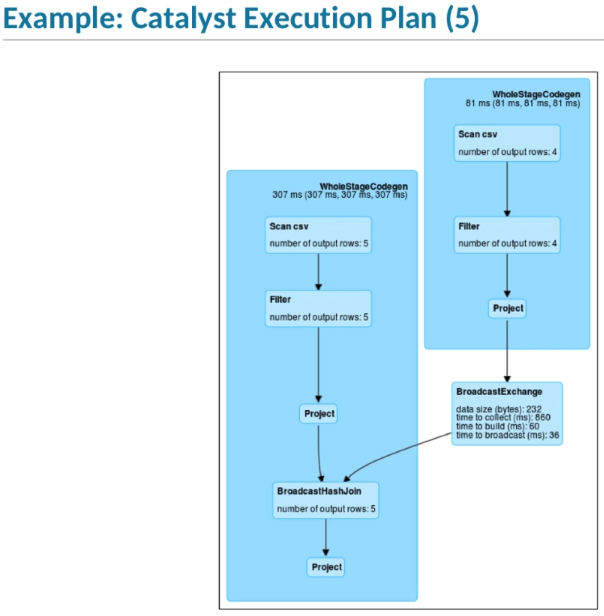






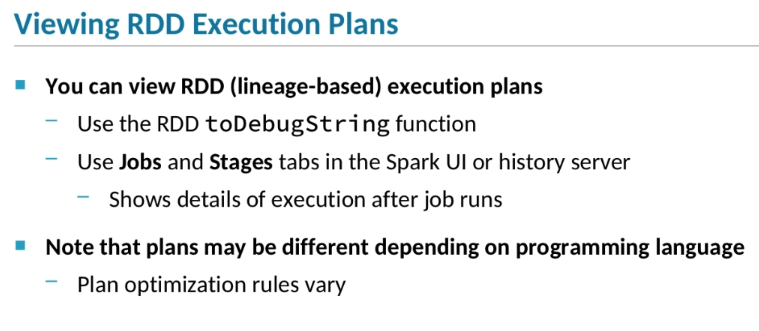


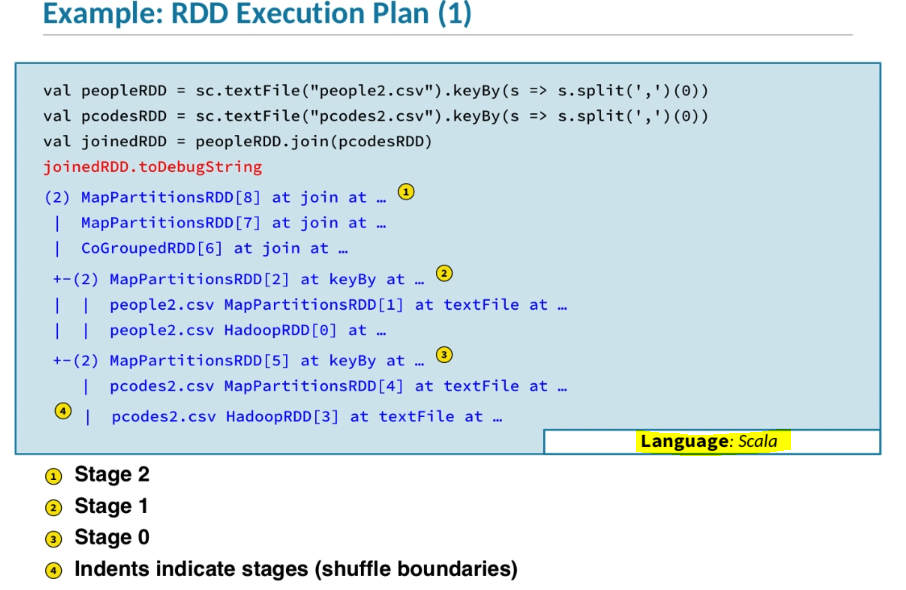


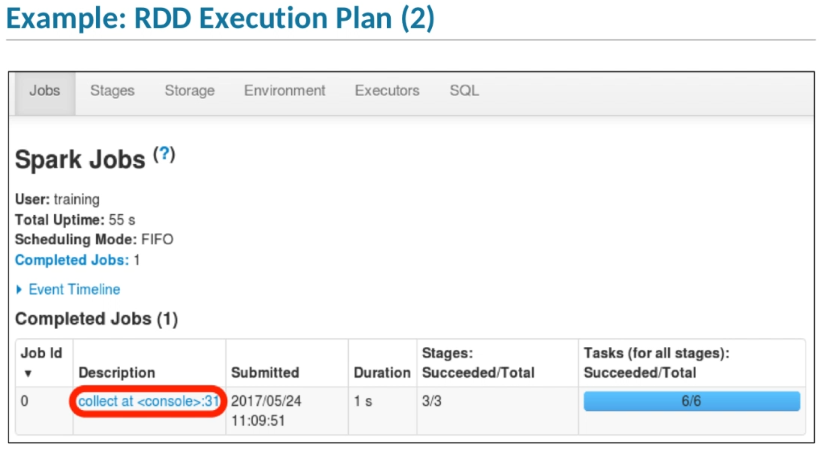


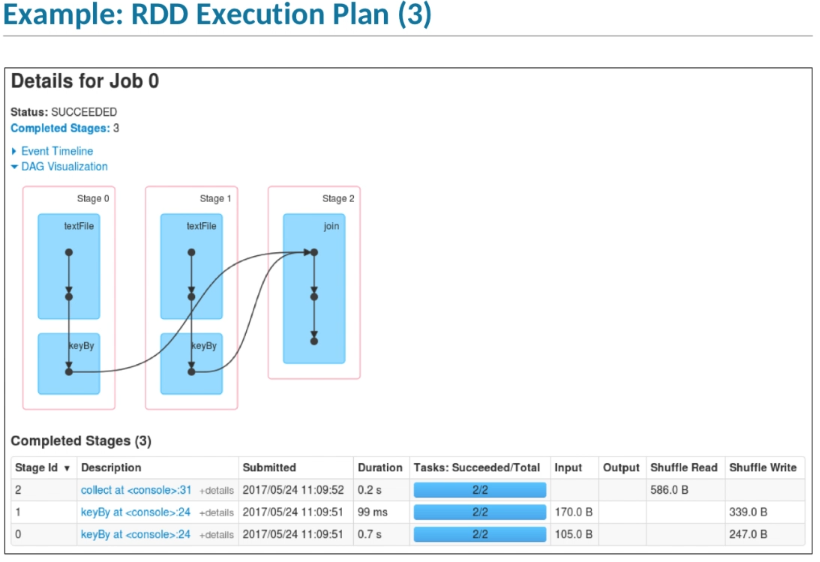
1. As a developer,
2. you may find it useful to view the execution plan for a query you’ve created or executed.
3. The best way to examine an execution plan depends on whether
4. the plan was created by Catalyst based on a SQL, DataFrame, or Dataset query,
5. or is a lineage-based execution plan created by core Spark for an RDD query.
6. **Let’s take a look at how to view Catalyst execution plans.**
7. Within the Spark shell or a Spark application,
8. you can use the explain function on a DataFrame or Dataset
9. to display its execution plan.
10. By default, it only shows the optimized physical plan,
11. but if you pass true or 1,
12. it will display all phases of the optimization process.
13. However,
14. explain can only show you the execution plan
15. relative to a query definition,
16. because Catalyst doesn't define the final execution plan until the job is executed.
17. You can use the Spark Application UI or history server UI
18. to see the details of how a query was actually executed.
19. Here’s an example of a query execution plan.
20. First we define the query.
21. For this example we define a simple query in which we create two DataFrames based on CSV files
22. and join them by a column called pcode.
23. Then we call the explain function
24. with the argument True,
25. which displays the full plan, including all the optimization phases in order.
26. Note that the full optimization plan includes a lot of information
27. that won’t be useful to most developers,
28. and is primarily for use by Spark contributors to debug execution planning.
29. So in this example, we’ll just point out a few key points of the plan phases.
30. The first plan step shown is the parsed logical plan.
31. The elements of this plan correspond directly to our query,
32. which reads data from two CSV files
33. and joins them on the pcode column.
34. The next phase is the analyzed logical plan.
35. In this phase, Catalyst resolves the column references using the DataFrame schemas
36. and analyzes the query to determine the schema of the result DataFrame.
37. After analyzing the plan, Catalyst optimizes it.
38. In our simple example, not much optimization is needed.
39. But Catalyst does perform one optimization.
40. Because we are doing an inner join,
41. the final results will not include any rows where the pcode value is null,
42. so the Catalyst added a step to filter out any of those rows
43. before doing the join.
44. This reduces the amount of data that needs to be joined.
45. The final phase shown is the physical plan.
46. This specifies the sequence of operations that will be executed,
47. such as scanning the CSV files,
48. determining the schemas,
49. and filtering out unnecessary rows.
50. Another optimization is added in this phase.
51. The BroadcastHashJoin instruction
52. says that the small amount of data from the pcode.csv file
53. will be pushed to each executor
54. so the data can be joined to the people.csv data locally without shuffling.
55. You can get a more detailed view of a query that has already been executed
56. by viewing the application in the Spark or history server UI.
57. The SQL tab shows a list of all DataFrame, Dataset, and SQL queries
58. that have been executed as part of this application.
59. In this example, we’ve executed the simple join query we looked at a moment ago
60. by calling the show action, which collects and displays the first 20 rows of the result.
61. Click on the query description to view the details of the execution.
62. The UI displays a visualization of the execution process.
63. In this example,
64. the large darker blue boxes represent the two stages of the query
65. —one which reads, filters, and broadcasts the pcode.csv data,
66. and the other which scans and reads the people.csv data and joins it with the pcode data.
67. Catalyst generated byte code
68. for both stages and distributed that to be run by the executors.
69. Although it isn’t shown here,
70. you can get even more detail on each execution step
71. by hovering your pointer over the step's box.

Example: RDD Execution Plans









1. RDD execution plans are usually simpler to understand than the highly optimized execution plans generated by Catalyst.
2. To view RDD execution plans programmatically, use the RDD toDebugString function.
3. Or you can use the Spark or history server UI to view the metrics and DAG for a job after it was executed.
4. Partitioning and execution sometimes vary slightly between Scala and Python.
5. **However, the process of viewing and analyzing the plan is the same in both cases, so we will just show a Scala example for brevity.**
6. In this example, we start by defining a simple RDD query.
7. In the first line, we read in a CSV file containing people data, split the line by commas,
8. and map the data to key-value pairs using the first field, which is the postal code, as the key.
9. In the second line, we do the same with postal code data.
10. Finally, we join the two RDDs by key.
11. Then we call toDebugString on the joined RDD to display the execution plan, which is based on the RDD's lineage.
12. The output shows the tasks within each stage, in order of execution, from the bottom up.
13. In this example, there are three stages.
14. Indentation denotes stage dependency;
15. Stage 2 is dependent on Stages 0 and 1.
16. Therefore Stages 0 and 1 are executed first and must complete before the final stage.
17. Within each stage are transformations that operate on the same partitions.
18. The RDDs in the first and second stages have two partitions each, as indicated by the number in parentheses.
19. During or after the execution of a query, you can see details of the execution by viewing the application in the Spark or history server UI.
20. The Jobs tab shows a list of the application’s running and completed jobs.
21. Note that next to the job description it shows job metrics, including how long the job took and how many stages and tasks ran.
22. In the example, we already know that the job had three stages because we reviewed the job's execution plan.
23. Here we see that all three stages executed correctly.
24. The number of tasks executed is six—one for each partition in each stage.
25. All three stages had two partitions, so that means a total of six tasks.
26. Clicking the job description takes you to the Stagestab where you can view details about the job's stages and tasks.
27. On the job details page, you can view a visualization of the execution plan DAG.
28. Here we see the three stages of our query, with arrows indicating that Stage 2 depends on Stage 0 and 1.
29. Below are metrics for each stage in the job, including how long the stage took,
30. how many tasks were completed, and how much data was read, saved, and shuffled.
31. Although it’s not shown here, you can also view metrics and other details for each task and executor involved in the job.