# Spark

Apache Spark is a unified computing engine and a set of libraries for parallel data processing on

computer clusters.

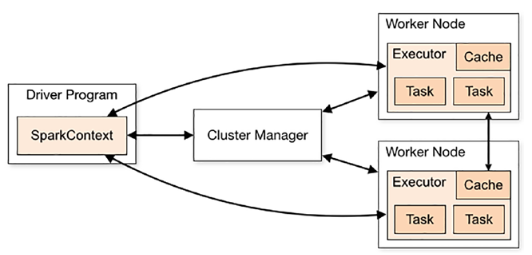
Spark programs, more commonly just called applications, are made possible due to

the following three main components:

• The driver program

• The cluster manager

• The program executors



The driver program is simply the Apache Spark application. Each application is

supervised by the driver program, and the stages of execution (the work) is divided

and distributed across the program executors using simple RPC communication for

each stage of execution, typically some kind of transformation, along the journey to the

application’s desired outcome, which is referred to as an action. Behind the scenes, the

cluster manager is hard at work simply keeping tabs on the state of the cluster, checking

in on the running applications, and watching the available compute capacity remaining

in the cluster.

# Driver Program

The driver program wears many hats and manages the complexities of running your

application with the help of the SparkContext.

## SparkContext

SparkContext controls

the following aspects of the driver program:

• Resource management: The Spark driver requests and releases

cluster resources by interfacing with the cluster manager (see

Figure 2-3). This enables your program logic to be distributed

across a sub-cluster of assigned compute nodes, called executors.

Given that the driver also handles the distribution of work across

these nodes, it can also coordinate with the cluster manager in the

case of lost executor nodes (if these machines go offline or become

unreachable), or in the case of task failure. Work can be rescheduled

again on any of the available executors to ensure that partial failures

don’t result in full application failure (within reason).

• Application state management: The SparkContext keeps track of the

active and completed jobs. This way, the application state machine

can control the execution of the driver program while also handling

failures within the application.

• Configuration manager: The SparkContext is also responsible for

synchronizing the initial application configuration within an object, named

the SparkConf. This is an initial snapshot of the configuration of a Spark

application, and it can be modified at runtime in a separate object named

the RuntimeConfig (which we will work with in later chapters as well).

• Job scheduler: The SparkContext controls the scheduling work across

the cluster as a series of jobs, stages, and tasks. We go into much more

detail in Chapter 3 as you start to write your first Spark applications

# Cluster Manager

the cluster manager is the cluster coordinator and the delegate in charge of managing and maintaining the state of the cluster, as well as the executors

assigned to each active Spark application.

# Spark Executors

Act as the compute delegate for tasks assigned by the Spark

driver program. Simply speaking, most of your work will be run across the executors and

not the driver program itself

# The Resilient Distributed Data Model

At a high level, you can think of the RDD as a read-only, immutable collection of

data partitioned across a set of network-connected servers that are bound to a

Spark application, called executors. The RDD object itself encodes the lineage of

transformations required to achieve a desired outcome.

the RDD acts like a

graph of transformational pointers, so in the case of a partial failure, the RDD representing

a specific phase of processing can be recomputed efficiently. The RDD itself stores no

physical data, simply metadata, making it a means to coordinate data processing across a

distributed cluster of network connected computers.

## Partitions

To allow every executor to perform work in parallel, Spark breaks up the data into chunks called partitions. A partition is a collection of rows that sit on one physical machine in your cluster. If you have one partition, Spark will have a parallelism of only one, even if you have thousands of executors. If you have many partitions but only one executor, Spark will still have a parallelism of only one because there is only one computation resource.

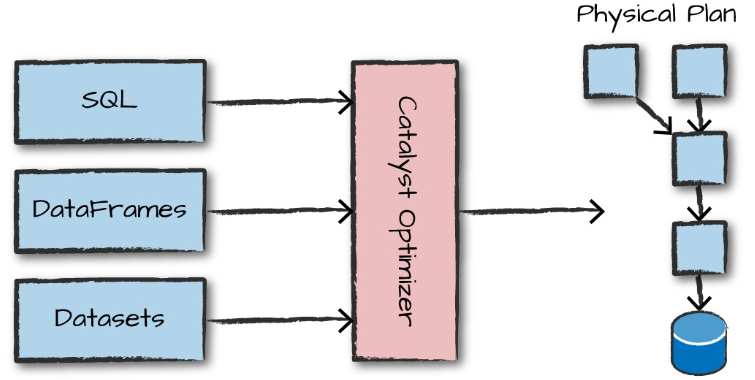
# Catalyst

Catalyst in Spark is a powerful query optimizer that is at the core of Spark SQL. It leverages advanced programming language features, particularly from Scala, to build an extensible and efficient query optimization framework. Catalyst is designed to optimize queries expressed in SQL or through the DataFrame/Dataset APIs, enhancing the performance and efficiency of Spark applications.

Catalyst supports both rule-based and cost-based optimization methods. Rule-based optimization applies predefined rules to optimize queries, while cost-based optimization generates multiple plans and selects the most efficient one based on cost estimates.

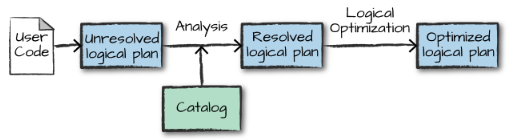
**Phases of Optimization**:

* **Analysis**: Converts the unresolved logical plan into a logical plan by resolving references using a catalog. This phase checks the validity of column names and data types.
* **Logical Optimization**: Applies standard rule-based optimizations to the logical plan, such as predicate pushdown and projection pruning.
* **Physical Planning**: Generates multiple physical plans and compares them based on cost to select the most efficient plan.
* **Code Generation**: Compiles parts of the query into Java bytecode using Scala's quasiquotes feature to improve execution speed.



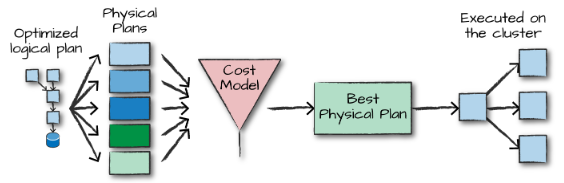
## Logical Planning

In Apache Spark, query processing involves multiple stages, starting with logical planning, where the query is parsed, analyzed, and optimized to produce an optimized plan[3](https://www.chashnikov.dev/post/spark-understanding-physical-plans). The logical plan is an abstract representation of the transformations needed without detailing how they are executed on the driver or worker nodes. The Catalyst Optimizer optimizes the plan by applying its own rules. For instance, it checks which tasks can be computed together in one stage, determines the execution order of queries for better performance in multi-join queries, and optimizes the query by evaluating the filter clause before any project.



## Physical Planning

The logical plan describes *what* needs to be done, using relational operators like Filter and Join, along with respective expressions[3](https://www.chashnikov.dev/post/spark-understanding-physical-plans). Physical planning then determines *how* the logical plan will be executed on the cluster[2](https://blog.knoldus.com/understanding-sparks-logical-and-physical-plan-in-laymans-term/). It specifies the execution sequence of operations like filter, where, and group By clauses. It determines the specific algorithms and strategies for executing the query, such as the order of joins and the type of join, … . The physical plan operates on Resilient Distributed Datasets. Spark evaluates multiple physical plans and selects the most optimal one based on a cost model that estimates execution time and resource usage.



# Transformations

In Spark, the core data structures are immutable, meaning they cannot be changed

after they’re created. This might seem like a strange concept at first: if you cannot

change it, how are you supposed to use it? To “change” a DataFrame, you need to

instruct Spark how you would like to modify it to do what you want. These instruc‐

tions are called transformations.

Spark will not act on transformations until we call an action (we

discuss this shortly). Transformations are the core of how you express your business

logic using Spark. There are two types of transformations: those that specify narrow

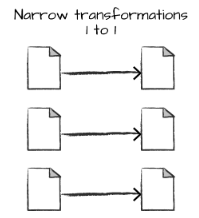
dependencies, and those that specify wide dependencies.

Transformations consisting of narrow dependencies (we’ll call them narrow transfor‐

mations) are those for which each input partition will contribute to only one output

partition. In the preceding code snippet, the where statement specifies a narrow

dependency, where only one partition contributes to at most one output partition.



A wide dependency (or wide transformation) style transformation will have input

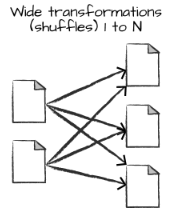
partitions contributing to many output partitions. You will often hear this referred to

as a shuffle whereby Spark will exchange partitions across the cluster. With narrow

transformations, Spark will automatically perform an operation called pipelining,

meaning that if we specify multiple filters on DataFrames, they’ll all be performed in memory. The same cannot be said for shuffles. When we perform a shuffle, Spark

writes the results to disk. Like aggregation,…



# Actions

Transformations allow us to build up our logical transformation plan. To trigger the

computation, we run an action. An action instructs Spark to compute a result from a

series of transformations.

There are three kinds of actions:

• Actions to view data in the console

• Actions to collect data to native objects in the respective language

• Actions to write to output data sources

# Structured Streaming

Structured Streaming is a high-level API for stream processing that became

production-ready in Spark 2.2. With Structured Streaming, you can take the same

operations that you perform in batch mode using Spark’s structured APIs and run

them in a streaming fashion. This can reduce latency and allow for incremental pro‐

cessing. The best thing about Structured Streaming is that it allows you to rapidly and

quickly extract value out of streaming systems with virtually no code changes. It also

makes it easy to conceptualize because you can write your batch job as a way to pro‐

totype it and then you can convert it to a streaming job. The way all of this works is

by incrementally processing that data.

## Structured API Execution

This section will demonstrate how this code is actually executed across a cluster. This will help

you understand (and potentially debug) the process of writing and executing code on clusters, so

let’s walk through the execution of a single structured API query from user code to executed

code. Here’s an overview of the steps:

1. Write DataFrame/Dataset/SQL Code.

2. If valid code, Spark converts this to a Logical Plan.

3. Spark transforms this Logical Plan to a Physical Plan, checking for optimizations along

the way.

4. Spark then executes this Physical Plan (RDD manipulations) on the cluster.

To execute code, we must write code. This code is then submitted to Spark either through the

console or via a submitted job. This code then passes through the Catalyst Optimizer, which

decides how the code should be executed and lays out a plan for doing so before, finally, the

code is run and the result is returned to the user. Like filtering, …

# Machine Learning and Advanced Analytics

Another popular aspect of Spark is its ability to perform large-scale machine learning

with a built-in library of machine learning algorithms called MLlib. Spark provides a sophisticated machine learning API for performing a variety of

machine learning tasks, from classification to regression, and clustering to deep

learning.

105

## spark-submit

spark-submit does one thing: it lets you send your

application code to a cluster and launch it to execute there. Upon submission, the

application will run until it exits (completes the task) or encounters an error.

By default, when we perform a shuffle, Spark outputs 200 shuffle partitions. To change that config :

spark.conf.set("spark.sql.shuffle.partitions", "5")

You can make any DataFrame into a table or view with one simple method call:

flightData2015.createOrReplaceTempView("flight\_data\_2015")

# Get records methods

1. **show(n)**
   * Displays the first n rows in a readable table format.
   * Executes as an action but **does not return data programmatically**—only prints it.
   * Optimized for quick inspection (e.g., verifying transformations)[5](https://www.reddit.com/r/apachespark/comments/1fer87t/display_fast_collect_cache_extremely_slow/).
2. **take(n)**
   * Returns the first n rows as a list to the driver.
   * More efficient than collect() for small n since it avoids loading the full dataset[3](https://sparktpoint.com/pyspark-collect-method-explanation/)[6](https://holdenk.github.io/spark-flowchart/details/best-pratice-collect/).
   * Example: Use df.take(10) to sample data for validation.
3. **collect()**
   * Retrieves **all rows** to the driver as a single list.
   * Risks Out-of-Memory (OOM) errors on large datasets[1](https://stackoverflow.com/questions/44348670/which-is-faster-in-spark-collect-or-tolocaliterator)[6](https://holdenk.github.io/spark-flowchart/details/best-pratice-collect/).
   * Only suitable for small datasets (e.g., results after heavy filtering/aggregation)[3](https://sparktpoint.com/pyspark-collect-method-explanation/).
4. **toLocalIterator()**
   * Returns an iterator that processes data one partition at a time.
   * Reduces driver memory pressure compared to collect() but still requires memory for the largest partition[2](https://sparkbyexamples.com/pyspark/pyspark-loop-iterate-through-rows-in-dataframe/)[7](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.toLocalIterator.html).
   * Slower than collect() due to sequential partition processing[1](https://stackoverflow.com/questions/44348670/which-is-faster-in-spark-collect-or-tolocaliterator)[4](https://www.waitingforcode.com/apache-spark/collecting-part-data-driver-rdd-tolocaiIterator/read).
   * Use case: Iterate through large datasets on the driver without loading all data at once[2](https://sparkbyexamples.com/pyspark/pyspark-loop-iterate-through-rows-in-dataframe/)[6](https://holdenk.github.io/spark-flowchart/details/best-pratice-collect/).