SPARK

Introduction

- Spark is the first fast, general-purpose distributed computing.
  - Rapidly gaining popularity particularly because of its speed and adaptability.
- Spark primarily achieves this speed via a new data model called resilient distributed datasets (RDDs) that are stored in memory while being computed upon,
  - Thus eliminating expensive intermediate disk writes.
- It also takes advantage of a directed acyclic graph (DAG) execution engine that can optimize computation,
  - particularly iterative computation, which is essential for data theoretic tasks such as optimization and machine learning.
- These speed gains allow Spark to be accessed in an interactive fashion
  - Making the user an integral part of computation
  - Allowing for data exploration of big datasets that was not previously possible
  - Bringing the cluster to the data scientist.

Spark Basics

- Apache Spark is a cluster-computing platform that provides an API for distributed programming like the MapReduce model,
- Designed to be fast for interactive queries and iterative algorithms
  - Primarily achieves this by caching data required for computation in the memory of the nodes in the cluster.
- In-memory cluster computation enables Spark to run iterative algorithms
  - as programs can checkpoint data and refer back to it without reloading it from disk;
- it supports interactive querying and streaming data analysis at extremely fast speeds.
- Spark is compatible with YARN
  - run on an existing Hadoop cluster and access any Hadoop data source, including HDFS, S3, HBase, and Cassandra.

Spark Basics

- Spark was designed from the ground up to support big data applications and data science in particular.
- Spark API has many other powerful distributed abstractions similarly related to functional programming, including sample, filter, join, and collect, to name a few.
- Spark is implemented in Scala
  - programming APIs in Scala, Java, R, and Python makes Spark much more accessible to a range of data scientists who can take fast and full advantage of the Spark engine.

The Spark Stack

- Spark is a general-purpose distributed computing abstraction and can run in a standalone mode.
- Spark focuses purely on computation rather than data storage
- Typically run in a cluster that implements data warehousing and cluster management tools.
- When Spark is built with Hadoop, it utilizes YARN to allocate and manage cluster resources like processors and memory via the ResourceManager.
- Spark can then access any Hadoop data source—for example HDFS, HBase, or Hive

The Spark Stack

- Spark exposes its primary programming abstraction to developers through the Spark Core module.
  - Module contains basic and general functionality, including the API that defines resilient distributed datasets (RDDs).
- RDDs, are the essential functionality upon which all Spark computation resides.
- Spark then builds upon this core, implementing special-purpose libraries for a variety of data science tasks that interact with Hadoop,

The Spark Stack

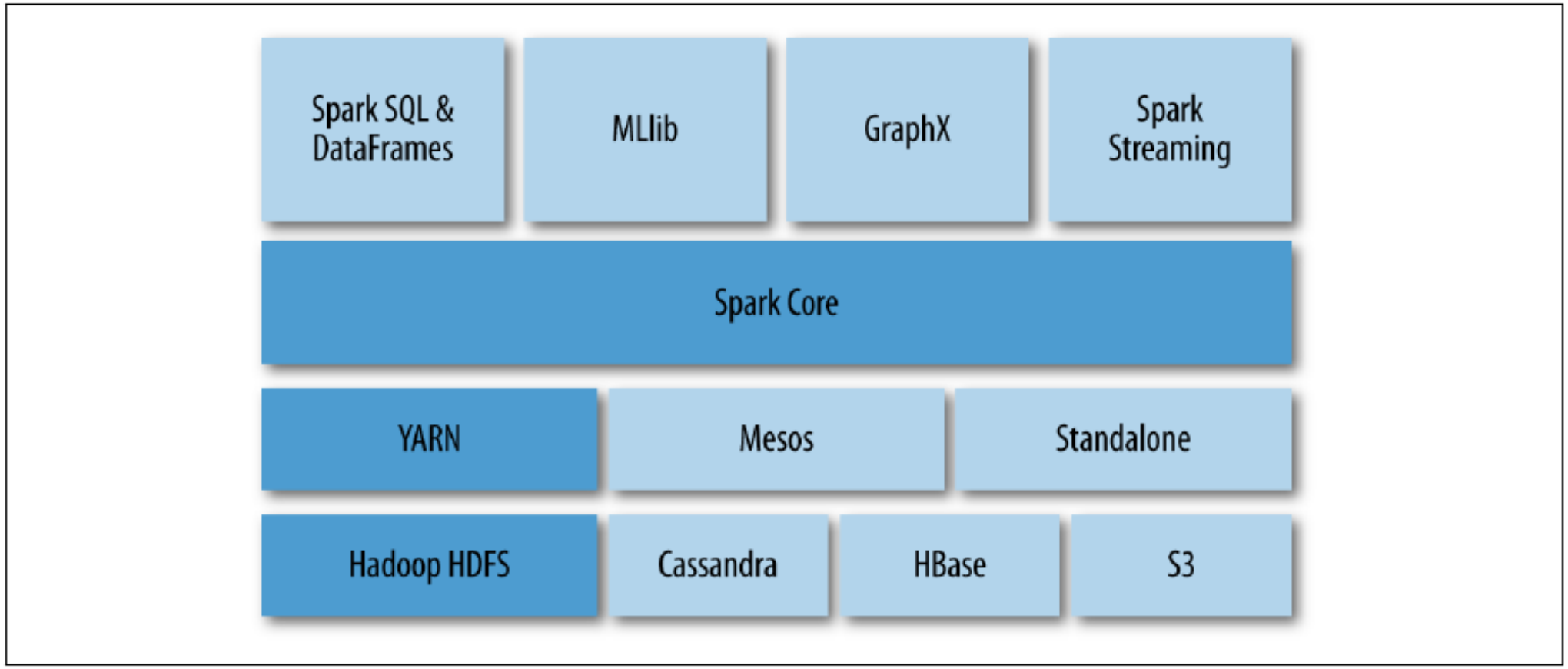


Figure 4-1. Spark is a computational framework designed to take advantage of cluster management platforms like YARN and distributed data storage like HDFS

The Spark Stack

- Component libraries are not integrated into the general-purpose computing Framework
  - making the Spark Core module extremely flexible and allowing developers to easily solve similar use cases with different approaches.
- For example, Hive will be moving to Spark, allowing an easy migration path for existing users;
- GraphX is based on the Pregel model of vertex-centric graph computation, but other graph libraries that leverage gather, apply, scatter (GAS) style computations could easily be implemented with RDDs.
- Flexibility means that specialist tools can still use Spark for development
  - new users can quickly get started with the Spark components that already exist.

primary components

- Spark SQL
  - APIs for interacting with Spark via the Apache Hive variant of SQL called HiveQL;
  - This library is moving toward providing a more general, structure data-processing abstraction, DataFrames.
  - DataFrames are essentially distributed collections of data organized into columns, conceptually similar to tables in relational databases.

The Spark Stack - primary components

Spark Streaming

- Enables the processing and manipulation of unbounded streams of data in real time.
- Many streaming data libraries (such as Apache Storm) exist for handling real-time data.
- Spark Streaming enables programs to leverage this data similar to how you would interact with a normal RDD as data is flowing in.

Mllib

- Library of common machine learning algorithms implemented as Spark operations on RDDs.
- Library contains scalable learning algorithms that require iterative operations across large datasets.
  - e.g., classifications, regressions
- Mahout library, formerly the big data machine learning library of choice, will move to Spark for its implementations in the future

The Spark Stack - primary components

GraphX

- Collection of algorithms and tools for
  - manipulating graphs
  - performing parallel graph operations and computations.
- GraphX extends the RDD API to include operations for
  - manipulating graphs
  - creating subgraphs
  - accessing all vertices in a path.

The Spark Stack

- These components combined with the Spark programming model provide a rich methodology of interacting with cluster resources.
  - Spark has become so immensely popular for distributed analytics.
- Basic API remains the same across components and the components themselves are easily accessed without extra installation.

Resilient Distributed Datasets

- Hadoop as a distributed computing framework dealt with two primary problems:
  - how to distribute data across a cluster,
  - how to distribute computation.
- Distributed data storage problem deals with high availability of data as well as recoverability and durability.
- Distributed computation intends to improve the performance (speed) of a computation by
  - breaking a large computation or task into smaller
  - independent computations that can be run simultaneously
  - aggregated to a final result.
- Each parallel computation is run on an individual node or computer in the cluster, distributed computing framework needs to provide consistency, correctness, and fault-tolerant guarantees for the whole computation.
- Spark does not deal with distributed data storage, relying on Hadoop to provide this functionality,
  - focuses on reliable distributed computation through a framework called resilient distributed datasets.

Resilient Distributed Datasets

- RDDs are essentially a programming abstraction that represents a read-only collection of objects that are partitioned across a set of machines.
- RDDs can be rebuilt from a lineage (so fault tolerant),
  - are accessed via parallel operations,
  - can be read from and written to distributed storages (e.g., HDFS or S3),
  - can be cached in the memory of worker nodes for immediate reuse.
- in-memory caching feature allows for massive speedups and provides for iterative computing required for machine learning and user-centric interactive analyses.

Resilient Distributed Datasets

- RDDs are operated upon with functional programming constructs that include and expand upon map and reduce.
- Programmers create new RDDs
  - by loading data from an input source
  - by transforming an existing collection to generate a new one.
- History of applied transformations is primarily defines the RDD's lineage,
  - because the collection is immutable, transformations can be reapplied to part or all of the collection in order to recover from failure.
- Spark API is therefore essentially a collection of operations that create, transform, and export RDDs.

Resilient Distributed Datasets

- Fundamental programming model is describing how RDDs are created and modified via programmatic operations.
- Two types of operations that can be applied to RDDs:
  - transformations and actions.
- Transformations are operations that are applied to an existing RDD to create a new RDD
  - for example, applying a filter operation on an RDD to generate a smaller RDD of filtered values.
- Actions, are operations that actually return a result back to the Spark driver program
  - resulting in a coordination or aggregation of all partitions in an RDD.

Resilient Distributed Datasets

- In this model,
  - map is a transformation--function is passed to every object stored in the RDD
  - Output of that function maps to a new RDD.
- aggregation like reduce is an action, because reduce requires the RDD to be repartitioned (according to a key)
  - some aggregate value like sum or mean computed and returned.
- Most actions in Spark are designed solely for the purpose of output
  - To return a single value or a small list of values, or to write data back to distributed storage.

Resilient Distributed Datasets

- Additional benefit of Spark is that it applies transformations “lazily”
  - Inspecting a complete sequence of transformations and an action before executing them by submitting a job to the cluster.
- lazy-execution provides significant storage and computation optimizations,
  - as it allows Spark to build up a lineage of the data and evaluate the complete transformation chain in order to compute upon only the data needed for a result;
  - For example, if you run the first() action on an RDD, Spark will avoid reading the entire dataset and return just the first matching line.

Programming with RDDs

- Programming Spark applications is similar to other data flow frameworks.
- Code is written in a driver program that is evaluated lazily on the driver-local machine when submitted,
- Upon an action, the driver code is distributed across the cluster to be executed by workers on their partitions of the RDD.
- Results are then sent back to the driver for aggregation or compilation.
- Driver program
  - creates one or more RDDs by parallelizing a dataset from a Hadoop data source
  - applies operations to transform the RDD
  - invokes some action on the transformed RDD to retrieve output.

What is In-Memory Computing?

- In-memory computing refers to the process of storing intermediate computation results in RAM (main memory) instead of writing them to disk between each operation.
- This dramatically reduces I/O overhead, which is a major bottleneck in traditional disk-based systems.

Apache Spark was designed to overcome the inefficiencies of Hadoop MapReduce, which writes intermediate results to HDFS after every map and reduce step.

In Spark:

- Data can be stored as RDDs (Resilient Distributed Datasets) or DataFrames in memory.
- When transformations (e.g., map(), filter(), join()) are applied, Spark keeps intermediate data in RAM.
- This makes iterative algorithms (like machine learning, graph processing, etc.) 10–100× faster.

RDD (Resilient Distributed Dataset) is a fault-tolerant, distributed collection of data that can be processed in parallel across a cluster.

In simpler terms:

- It's like a big list (or array) of data, split into smaller chunks (called partitions) across many machines (nodes).
- You can perform operations (like map, filter, reduce) on all those chunks in parallel.

How RDDs are Created

There are three main ways to create an RDD:

(a) From an existing collection

```
data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data)
```

(b) From external storage

(e.g., HDFS, S3, local files)

```
rdd = sc.textFile("hdfs:///data/input.txt")
```

(c) From other RDDs (via transformations)

```
rdd2 = rdd.filter(lambda x: x > 3)
```

RDD Operations

- RDDs support two types of operations:

(a) Transformations → Create a new RDD

Transformation	Description	Example
map()	Apply a function to each element	rdd.map(lambda x: x * 2)
filter()	Keep only elements that match a condition	rdd.filter(lambda x: x > 10)
flatMap()	Similar to map(), but flattens results	rdd.flatMap(lambda x: x.split(" "))
union()	Combine two RDDs	rdd1.union(rdd2)
distinct()	Remove duplicates	rdd.distinct()

(b) **Actions** → Trigger computation and return results

Action	Description	Example
collect()	Return all elements to the driver	rdd.collect()
count()	Count elements	rdd.count()
take(n)	Return first *n* elements	rdd.take(5)
reduce()	Aggregate elements using a function	rdd.reduce(lambda a, b: a + b)
saveAsTextFile()	Write RDD to external storage	rdd.saveAsTextFile("output")

Feature	RDD	DataFrame	Dataset
Type Safety	No	Yes (in Scala/Java)	Yes
Performance	Lower (no optimization)	Higher (Catalyst optimizer)	High
API Level	Low-level (functional)	High-level (SQL-like)	High-level (typed)
When to Use	Complex transformations, unstructured data	SQL analytics, structured data	Type-safe, structured data

RDD Lineage is the record of all transformations that were applied to create an RDD from other datasets.

In other words, Spark keeps track of how an RDD was derived — step by step — from its parent RDDs (through operations like map(), filter(), join(), etc.).

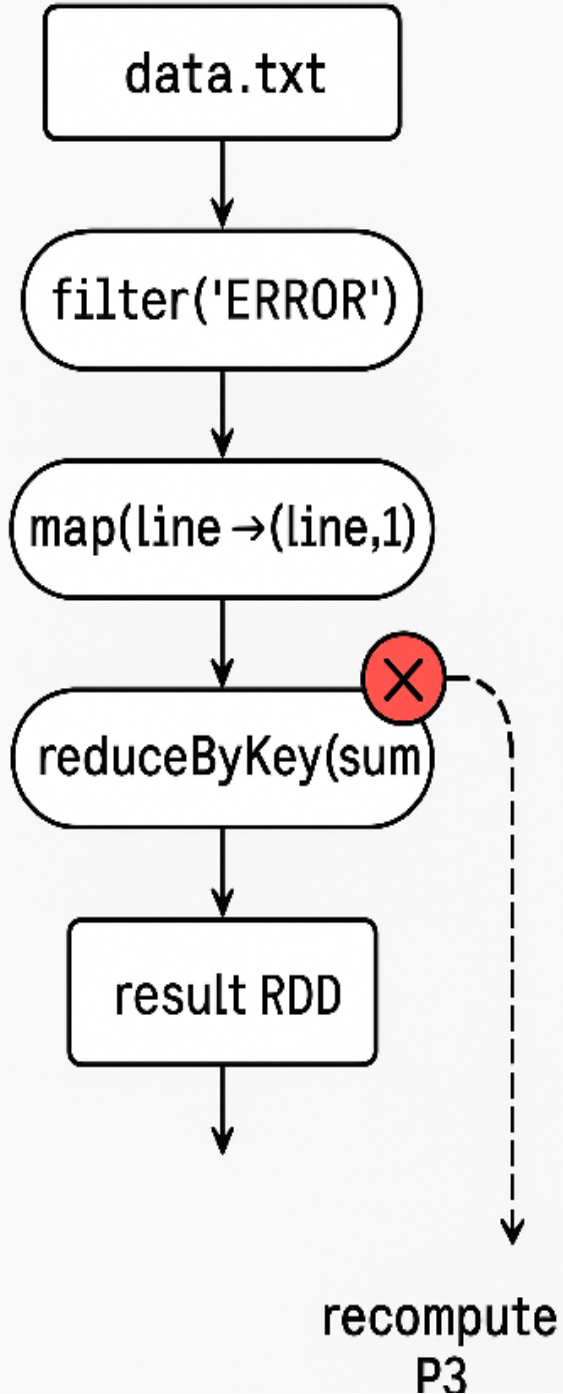
This lineage information is stored as a Directed Acyclic Graph (DAG), also called the lineage graph.

Why Lineage is Important

- When Spark runs a job, it may store RDDs in memory or on disk, but if part of that data is lost (e.g., due to a node failure), Spark can rebuild only the missing partitions — not the entire dataset.
- It does this using the lineage information.

Fault Recovery using Lineage

- Suppose Spark has an RDD divided into 4 partitions (P1, P2, P3, P4).
- Node containing P3 fails and the partition is lost.
- Instead of restarting the job or reloading all data, Spark:
  - Checks the lineage graph.
  - Finds out how P3 was created (e.g., by applying a filter on data from data.txt).
  - Recomputes only P3 by rerunning the required transformations on that portion of the input.



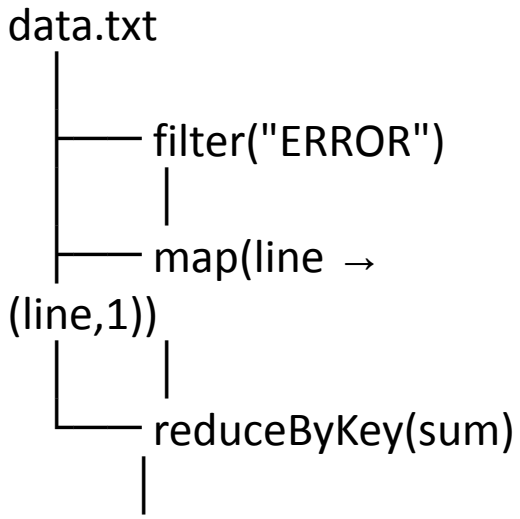
Lineage Graph (DAG)

The lineage of RDDs forms a **Directed Acyclic Graph (DAG)**:

- Nodes = RDDs
- Edges = Transformations between RDDs

Spark uses this DAG to:

- Plan the execution of jobs efficiently.
- Reconstruct lost data in case of failure.



Lineage vs. Checkpointing

- While lineage is great, if a job has too many transformations, recomputation could be expensive.
- So Spark also supports checkpointing:
- Saves an RDD's data to stable storage (like HDFS).
- Truncates its lineage.
- Used when recomputation cost or lineage depth is too high.

Feature	Lineage	Checkpoint
Storage	In-memory metadata	Stable storage (e.g., HDFS)
Recovery	Recompute lost partitions	Load from checkpoint file
Overhead	Minimal	Higher (I/O cost)
Best for	Short or moderate pipelines	Long iterative jobs

Programming with RDDs

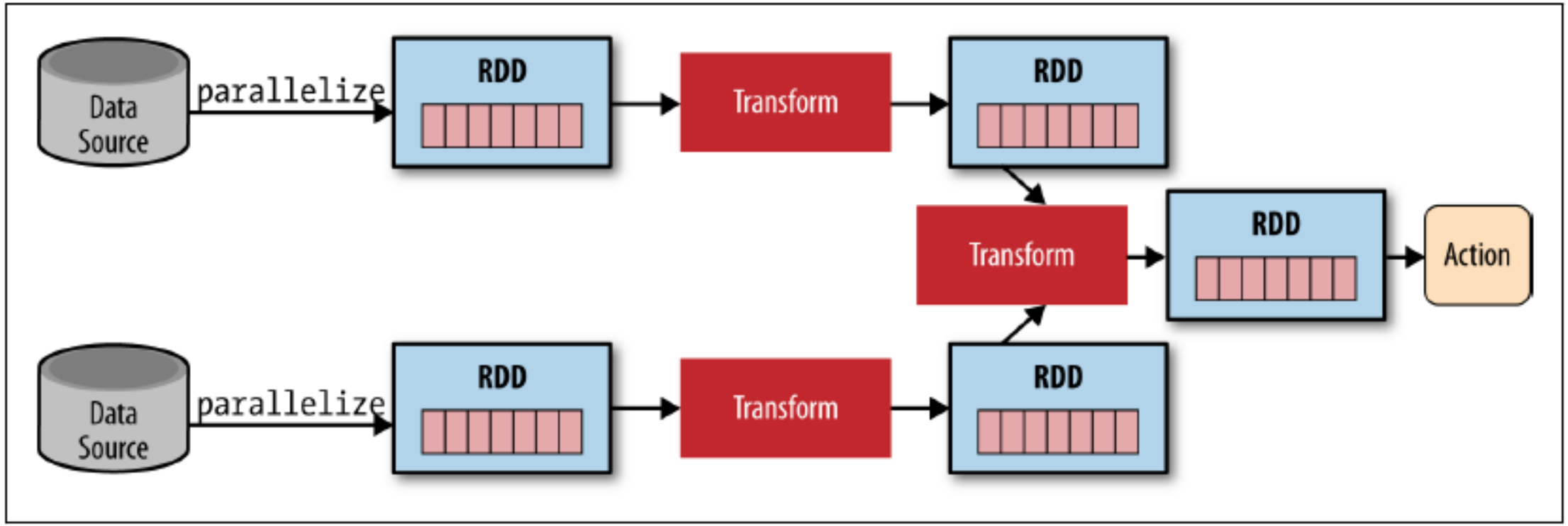


Figure 4-2. A typical Spark application parallelizes (partitions) a dataset across a cluster into RDDs

Programming with RDDs

- A typical data flow sequence for programming Spark is as follows:

I Step

- Define one or more RDDs
  - through accessing data stored on disk (e.g., HDFS, Cassandra, HBase, or S3)
  - through parallelizing some collection
  - through transforming an existing RDD,
  - by caching.
- Caching is one of the fundamental procedures in Spark
  - storing an RDD in the memory of a node for rapid access as the computation progresses.

II Step

Invoke operations on the RDD by passing closures to each element of the RDD. Spark offers many high-level operators beyond map and reduce.

III Step

Use the resulting RDDs with aggregating actions (e.g., count, collect, save, etc.). Actions kick off the computation on the cluster because no progress can be made until the aggregation has been computed.

Advantage of RDD	Description
Fault tolerance	Recovers lost data using lineage
In-memory computation	Reduces disk I/O; faster processing
Immutability	Simplifies recovery and debugging
Parallelism	Automatic data partitioning and parallel execution
Lazy evaluation	Optimized execution via DAG
Rich API	Expressive transformations and actions
Type safety	Compile-time error checking (Scala/Java)
Data flexibility	Works with structured or unstructured data
Reusability	Cached RDDs reused across operations
Integration	Forms foundation for Spark’s entire ecosystem

Creating and Manipulating RDDs in Apache Spark

- RDDs (Resilient Distributed Datasets) are Spark's **core data structure** for parallel computation.
- You can create them, apply **transformations** to derive new datasets, and perform **actions** to trigger computation.

1. Initialize Spark Context

Before working with RDDs, you need a `SparkContext` (usually via `SparkSession`).

```
from pyspark.sql import SparkSession

# Initialize SparkSession
spark = SparkSession.builder.
appName("RDDExample").getOrCreate()

# Access the SparkContext
sc = spark.sparkContext
```

2. Create RDDs

A. From a Python Collection

```
data = [1, 2, 3, 4, 5, 6]
rdd1 = sc.parallelize(data)
```

This distributes the list across multiple nodes as an RDD.

B. From an External File

```
rdd2 = sc.textFile("data/sample.txt")
```

Each line in the text file becomes one element (record) of the RDD

3. Apply Transformations

- Transformations **create new RDDs** from existing ones.
- They are **lazy**, meaning Spark builds a DAG (Directed Acyclic Graph) but does **not** execute immediately.

Let's say we want to square even numbers from our dataset:

```
numbers = sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Filter even numbers and square them
transformed_rdd = numbers.filter(lambda x: x % 2 == 0).map(lambda x: x ** 2)
```

4. Apply Actions

Actions **trigger computation** and return results to the driver or save to storage.

```
# Trigger computation
result = transformed_rdd.collect()
print("Squared Even Numbers:", result)
```

```
# Count elements
count = transformed_rdd.count()
print("Count:", count)
```

```
# Sum of squares
sum_squares = transformed_rdd.
reduce(lambda a, b: a + b)
print("Sum of Squares:", sum_square
```

comparison between Spark's in-memory model and Hadoop MapReduce's disk-based model.

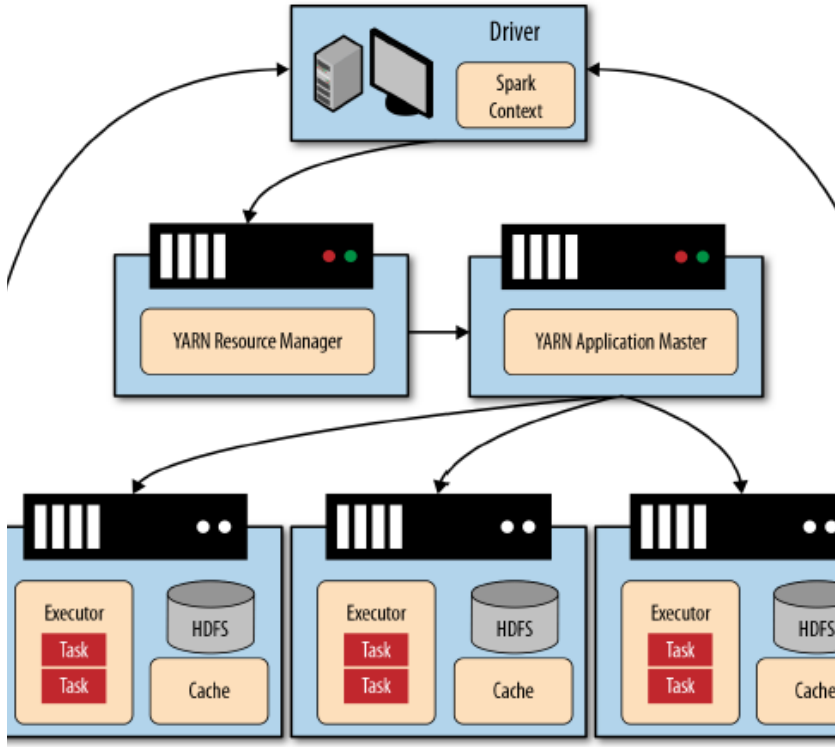
Feature	**Spark (In-Memory)**	**Hadoop MapReduce (Disk-Based)**
Data Persistence	In-memory (RAM)	On-disk (HDFS)
Performance	10–100× faster	Slower due to disk I/O
Fault Tolerance	RDD lineage	HDFS replication
Iterative Processing	Excellent	Poor
Real-Time Processing	Supported	Not supported
Ease of Use	High (Python, Scala, SQL APIs)	Complex (Java Map/Reduce)

Spark's in-memory computation model fundamentally improves performance by minimizing disk I/O and enabling fast iterative and interactive processing.
Hadoop MapReduce's disk-based model, while robust and reliable, incurs high overhead due to frequent writes to and reads from HDFS after every job stage.

- **Spark → Fast, In-Memory, Iterative, Interactive**
- **MapReduce → Slow, Disk-Based, Batch-Oriented**

Spark Execution

- Spark applications are run as independent sets of processes, coordinated by a Spark Context in a driver program.
- Context will connect to some cluster manager (e.g., YARN), which allocates system resources.
- Each worker in the cluster is managed by an executor, which is in turn managed by the Spark Context.
- Executor manages computation as well as storage and caching on each machine.



ure 4-3. In the Spark execution model, the driver program is an essential part o
cessing

Spark Execution

- Application code is sent from the driver to the executors
- Executors specify the context and the various tasks to be run.
- Executors communicate back and forth with the driver for data sharing or for interaction.
- Drivers are key participants in Spark jobs
  - they should be on the same network as the cluster.
- Different from Hadoop code
  - where you might submit a job from anywhere to the `ResourceManager`, which then handles the execution on the cluster.

Spark Execution

- Spark applications can be submitted to the Hadoop cluster in two modes:
  - yarn-client and yarn-cluster.
- In yarn-client mode, the driver is run inside of the client process as described,
  - ApplicationMaster simply manages the progression of the job and requests resources.
- In yarncluster mode, the driver program is run inside of the ApplicationMaster process
  - releasing the client process and proceeding more like traditional MapReduce jobs.
- Programmers would use
  - yarn-client mode to get immediate results or in an interactive mode
  - yarn-cluster for long-running jobs or ones that do not require user intervention.

Apache Spark deployment and performance running Spark **locally** and on a **cluster**.

Component	Local Mode	Cluster Mode
Driver	Runs on the same machine as executors	Runs on a cluster node (can be client or cluster-managed)
Executors	All run as threads within the same JVM	Run as separate JVM processes across multiple cluster nodes
Cluster Manager	Not needed (runs locally)	Uses YARN, Kubernetes, or Spark Standalone
Parallelism	Limited by number of CPU cores on the local machine	Parallelism scales across all cluster nodes
Storage Access	Typically local disk	Distributed storage (HDFS, S3, etc.)

Performance Factor	Local Mode	Cluster Mode
Computation Speed	Limited to one machine's CPU & memory	Scales horizontally across nodes → faster for large datasets
Data Size Handling	Can handle small/medium data	Handles massive datasets (terabytes/petabytes)
Fault Tolerance	If the machine fails, job fails	High fault tolerance (executors can be restarted on other nodes)
I/O Throughput	Local file system or single-node HDFS	Distributed I/O from HDFS, S3, or cluster storage
Network Overhead	None (local)	Some overhead for inter-node communication
Latency	Low for small data (no network shuffle)	Slightly higher due to network communication, but amortized by parallelism
Scalability	Limited to local hardware	Virtually unlimited (add more nodes)
Resource Utilization	Single machine resources only	Uses distributed CPU, memory, and disk of cluster

Spark applications in **local mode** are convenient for small-scale development and debugging, but limited by local hardware resources.

In contrast, **cluster mode** distributes computation and storage across multiple nodes, enabling Spark to handle **massive datasets** with **high throughput, scalability, and fault tolerance** — making it ideal for production environments.

Writing Spark Applications

- Writing Spark applications in Python is similar to working with Spark in the interactive console because the API is the same.
- Instead of typing commands into an interactive shell, you need to create a complete, executable driver program to submit to the cluster.
- Involves a few housekeeping tasks that were automatically taken care of in pyspark
  - Getting access to the SparkContext, which was automatically loaded by the shell.

Writing Spark Applications

- Spark programs are simple Python scripts that contain some data (shared variables)
  - Define closures for transforming RDDs,
  - Describe a step-by step execution plan of RDD transformation and aggregation.

Basic template for writing a Spark application in Python

- Template exposes the top-down structure of a Python Spark application:
- Imports allow various Python libraries to be used for analysis as well as Spark components such as GraphX or SparkSQL.
- Shared data and variables are specified as module constants, including an identifying application name that is used in web UIs, for debugging, and in logging.
- Job-specific closures or custom operators are included with the driver program for easy debugging or to be imported in other Spark jobs
- Main method defines the analytical methodology that transforms and aggregates RDDs, which is run as the driver program.

Basic template for writing a Spark application in Python

- Python programmers should note the use of the if __name__ == '__main__' statement
  - Spark configuration and SparkContext are defined and passed to the main function.
- Use of the ifmain allows us to easily import driver code into other Spark contexts, without creating a new context or configuration and executing a job
- Spark programmers will routinely import code from applications into an iPython/Jupyter notebook or the pyspark interactive shell to explore the analysis before running a job on a larger dataset.

Basic template for writing a Spark application in Python

- Driver program defines the entirety of the Spark execution;
- for example, to stop or exit the program in code,
  - programmers can use sc.stop() or sys.exit(0).
- in this template, a Spark cluster configuration, local[*] is hardcoded into the SparkConf via the setMaster method.
  - This tells Spark to run on the local machine using as many processes as available
- You can specify where Spark executes on the command line using spark-submit,
  - Driver programs often select this based on an environment variable using os.environ.
- While developing Spark jobs the job can be run locally
- In production run across the cluster on a larger data set.

Writing Spark Applications

- Writing Spark applications is certainly different than writing MapReduce applications
  - flexibility provided by the many transformations and actions
  - flexible programming environment.

Step	Description
1	Set up environment (Spark, Python/Scala, Java)
2	Import necessary Spark libraries
3	Create `SparkSession` (entry point)
4	Load or create data (RDD/DataFrame)
5	Apply transformations (lazy operations)
6	Apply actions (trigger execution)
7	Optimize (cache, persist, partition)
8	Configure and monitor job
9	Stop the session and release resources

Interactive Data Analysis

Interactive data analysis means running **queries or computations on large datasets and getting results quickly**, allowing analysts or data scientists to:

- Explore data in **real time**
- Run **ad-hoc queries**
- Perform **iterative analysis** (test → tweak → rerun)

- Traditional systems like Hadoop are **batch-oriented** — each job takes minutes to hours — making interactive work impractical.
- Spark was designed to fix this.

Spark Enables Interactive Analysis

Apache Spark enables interactive analysis primarily through its

- in-memory computing,
- lazy evaluation, and
- high-level APIs.

In-Memory Computing

- Spark stores data in **RAM** (not disk) using **RDDs** or **DataFrames**.
- This avoids repeated disk I/O between stages, which drastically reduces latency.
- As a result, queries that took **minutes** in Hadoop can take **seconds** in Spark

Lazy Evaluation and DAG Optimization

- Spark doesn’t execute transformations immediately.
- It builds a Directed Acyclic Graph (DAG) of operations.
- When an action (like show() or count()) is triggered, Spark optimizes the plan and executes it efficiently.

Spark SQL and DataFrames

- Spark provides a SQL interface (Spark SQL) and DataFrame API for interactive queries.
- Users can use familiar SQL commands directly on large distributed datasets.

REPL Support (Interactive Shells)

- Spark provides interactive shells:
- PySpark Shell (Python)
- Spark Shell (Scala)
- SparkR Shell (R)

Feature	Hadoop MapReduce	Apache Spark
Execution Model	Batch (disk-based)	In-memory
Response Time	Minutes	Seconds or less
Data Reuse	No (writes after each stage)	Yes (cache/persist)
Query Interface	Java, Pig, Hive (slow)	SQL, Python, Scala (fast)
Use Case	Batch processing	Interactive & iterative analysis

Interactive Spark Using PySpark

- For datasets that fit into the memory of a cluster
  - Spark is fast enough to allow data scientists to interact and explore big data from an interactive shell that implements a Python REPL (read-evaluate-print loop)
  - called pyspark.
- Similar to
  - interaction with native Python code in the Python interpreter
  - writing commands on the command line
  - receiving output to stdout
- Scala and R interactive shells
- This type of interactivity also allows the use of interactive notebooks and setting up an iPython or Jupyter notebook with a Spark environment is very easy.

Interactive Spark Using PySpark

- How to use RDDs with pyspark?.
- In order to run the interactive shell, you will need to locate the pyspark command
  - Bin directory of the Spark library.
- Similar to $HADOOP_HOME, you should also have a $SPARK_HOME.
- Spark requires no configuration to run right off the bat, so simply downloading the Spark build for your system is enough.
- Replacing $SPARK_HOME with the download path, you can run the interactive shell as follows:

```
hostname $ $SPARK_HOME/bin/pyspark
[… snip …]
>>>
```

Interactive Spark Using PySpark

- PySpark automatically creates a SparkContext for you to work with, using the local Spark configuration.
- It is exposed to the terminal via the sc variable. Let's create ourfirst RDD:

```
>>> text = sc.textFile("shakespeare.txt")
>>> print text
shakespeare.txt MappedRDD[1] at textFile at  NativeMethodAccessorImpl.java:-2
```

- `textFile` method loads the complete works of William Shakespeare from the local disk into an RDD named text.
- If you inspect the RDD, you can see that it is a MappedRDD and that the path to the file is a relative path from the current working directory

Interactive Spark Using PySpark

- Transform this RDD in order to implement word count application using Spark:

```
>>> from operator import add
>>> def tokenize(text):
...     return text.split()
...
>>> words = text.flatMap(tokenize)
```

Interactive Spark Using PySpark

- We imported the operator **`add`**
  - which is a named function that can be used as a closure for addition.
- First thing we have to do is split our text into words.
  - We created a function called **`tokenize`** whose argument is some piece of text
    - returns a list of the tokens (words) in that text by simply splitting on whitespace.
- We then created a new RDD called **`words`** by transforming the `text` RDD through the application of the **`flatMap`** operator, and passed it the closure `tokenize`.

Interactive Spark Using PySpark

- At this point, we have an RDD of type PythonRDD called words;
- Entering these commands has been instantaneous,
  - You might have expected a slight processing delay as the entirety of Shakespeare was split into words.
- Spark performs lazy evaluation, the execution of the processing has not occurred yet.
  - read the dataset,
  - partition across processes,
  - map the tokenize function to the collection
- PythonRDD
  - describes what needs to take place to create this RDD
  - maintains a lineage of how the data got to the words form.

Interactive Spark Using PySpark

- We can therefore continue to apply transformations to this RDD without waiting for a long, possibly erroneous or non-optimal distributed execution to take place.
- Next steps are to map each word to a key/value pair, where the key is the word and the value is a 1
- then use a reducer to sum the 1s for each key

Interactive Spark Using PySpark

- Map

```
>>> wc = words.map(lambda x: (x,1))
>>> print wc.toDebugString()
(2) PythonRDD[3] at RDD at PythonRDD.scala:43
|  shakespeare.txt MappedRDD[1] at textFile at NativeMethodAccessorImpl.java:-2
|  shakespeare.txt HadoopRDD[0] at textFile at
NativeMethodAccessorImpl.java:-2
```

Interactive Spark Using PySpark

- Instead of using a named function, we will use an anonymous function
  - with the lambda keyword in Python
  - This line of code will map the lambda to each element of words.
- Each x is a word, and the word will be transformed into a tuple (word, 1) by the anonymous closure.
- To inspect the lineage so far, we can use the toDebugString method to see how our PipelinedRDD is being transformed.

Interactive Spark Using PySpark

- We can then apply the reduceByKey action to get our word counts and then write those word counts to disk:

```
>>> counts = wc.reduceByKey(add)
>>> counts.saveAsTextFile("wc")
```

- we finally invoke the action saveAsTextFile
  - the distributed job kicks off and you should see a lot of INFO statements as the job runs "across the cluster"
  - or simply as multiple processes on your local machine).
- If you exit the interpreter, you should see a directory called wc in your current working directory:

Interactive Spark Using PySpark

```
hostname $ ls wc/
_SUCCESS   part-00000 part-00001
```

- Each part file represents a partition of the final RDD that was computed by various processes on your computer and saved to disk.
- If you use the head command on one of the part files, you should see tuples of word count pairs:

```
hostname $ head wc/part-00000
(u'fawn', 14)
(u'Fame.', 1)
(u'Fame,', 2)
(u'kinghenryviii@7731', 1)
(u'othello@36737', 1)
(u'loveslabourslost@51678', 1)
(u'1kinghenryiv@54228', 1)
(u'troilusandcressida@83747', 1)
(u'fleeces', 1)
(u'midsummersnightsdream@71681', 1)
```

Interactive Spark Using PySpark

- In a MapReduce job, the keys would be sorted due to the mandatory intermediate shuffle and sort phase between map and reduce.
- Spark's repartitioning for reduction does not necessarily utilize a sort
  - because all executors can communicate with each other
- Without the sort, you are guaranteed that each key appears only once across all part files
  - reduceByKey operator was used to aggregate the counts RDD.
- If sorting is necessary, you could use the sort operator to ensure that all the keys are sorted before writing them to disk.

Visualizing Airline Delays with Spark

visualizing airline delays with Apache Spark is a classic big data use case that demonstrates data ingestion, transformation, and visualization at scale. Let's go step by step on how you can **analyze and visualize airline delays using Spark (PySpark)**, including sample code and ideas for visualizations.

- Compute the average flight delay per airport using the US Department of Transportation's on-time flight dataset.
- This kind of computation—parsing a CSV file and performing an aggregate computation—is an extremely common use case of Hadoop, particularly as CSV data is easily exported from relational databases.
- This dataset, which records all US domestic flight departure and arrival times along with their delays, is also interesting because while a single month is easily computed upon, the entire dataset would benefit from distributed computation due to its size.

1. Load airline delay data (commonly from the **U.S. Bureau of Transportation**).
2. Clean and transform it using **PySpark**.
3. Aggregate statistics (average delay by airline, airport, month, etc.).
4. Export data to Pandas or visualization tools (Matplotlib, Seaborn, Plotly).

Step 1: Load and Inspect Data

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

spark = SparkSession.builder.appName("AirlineDelayAnalysis").getOrCreate()

# Load CSV data
df = spark.read.csv("airline_delay_data.csv", header=True, inferSchema=True)

# Inspect
df.printSchema()
df.show(5)
```

Step 2: Clean Data

Filter out missing or cancelled flights:

```
clean_df = df.filter((col("Cancelled") == 0) & (col("DepDelay").isNotNull()) &
(col("ArrDelay").isNotNull()))
```

Step 3: Compute Aggregates

- Average delay by airline

```
avg_delay_by_airline = clean_df.groupBy("Carrier").avg("DepDelay", "ArrDelay")
avg_delay_by_airline = avg_delay_by_airline.withColumnRenamed("avg(DepDelay)",
"AvgDepDelay") \
                                .withColumnRenamed("avg(ArrDelay)", "AvgArrDelay")


avg_delay_by_airline.show()
```

Average delay by month

```
from pyspark.sql.functions import round

monthly_delay = clean_df.groupBy("Month").avg("ArrDelay")
monthly_delay = monthly_delay.withColumnRenamed("avg(ArrDelay)",
"AvgArrDelay")
monthly_delay = monthly_delay.orderBy("Month")

monthly_delay.show()
```

Convert to Pandas for Visualization

Spark isn't a visualization tool — but you can **convert small aggregates to Pandas**:

```
pdf_airline = avg_delay_by_airline.toPandas()
pdf_month = monthly_delay.toPandas()
```

Airline Delay Comparison

```
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(10,6))
sns.barplot(x="Carrier",
y="AvgArrDelay", data=pdf_airline)
plt.title("Average Arrival Delay by
Airline")
plt.xlabel("Airline")
plt.ylabel("Average Arrival Delay
(minutes)")
plt.show()
```

Monthly Trend

```
plt.figure(figsize=(10,6))
sns.lineplot(x="Month", y="AvgArrDelay",
data=pdf_month, marker='o')
plt.title("Monthly Average Arrival
Delay")
plt.xlabel("Month")
plt.ylabel("Average Delay (minutes)")
plt.show()
```

potential bottlenecks in Spark applications handling large airline datasets

Bottleneck	Cause	Symptom	Solution
Data Skew	Uneven key distribution	Long tail tasks	Key salting, repartition
Shuffle Overhead	GroupBy/Join	Slow stage completion	Broadcast, reduce shuffles
Memory Pressure	Oversized datasets	GC, OOM	Cache selectively, tune memory
Partition Imbalance	Wrong partition count	Unused CPU cores	Repartition or coalesce
Inefficient Joins	Large shuffle joins	Long stages	Broadcast joins
I/O Overhead	Large reads/writes	Disk waits	Use Parquet, compression
Network Latency	High data transfer	Slow shuffles	Optimize locality
Driver Overload	Large collect	Driver OOM	Aggregate before collect
Config Issues	Poor tuning	Underperformance	Adjust executors & cores

In Spark applications processing large airline datasets, **data skew**, **shuffle operations**, **memory mismanagement**, and **inefficient joins** are the most common bottlenecks.

Proper **data partitioning**, **broadcast joins**, **serialization optimization**, and **resource tuning** are essential to achieve **high performance and scalability**.

RDD-based programming with DataFrame-based programming in Spark

When to Use Each

Scenario	Recommended API
Low-level control of transformations	**RDD**
Unstructured data (logs, binary data)	**RDD**
Structured/tabular data	**DataFrame**
SQL-style operations or aggregation	**DataFrame**
Performance-critical pipelines	**DataFrame**
Type-safe, complex algorithms (Scala)	**Dataset (typed DataFrame)**

Feature	**RDD (Resilient Distributed Dataset)**	**DataFrame**
Definition	Low-level distributed collection of Java/ Python/Scala objects.	High-level distributed table with named columns (like a SQL table).
API Type	Functional API (map, filter, reduce).	Declarative API (select, filter, groupBy, SQL).
Abstraction Level	Low-level abstraction.	High-level abstraction built on top of RDDs.
Introduced In	Spark 1.0	Spark 1.3 (and matured in 2.x with Catalyst optimizer).

- **RDDs** provide fine-grained control but require manual optimization and schema management.
- **DataFrames**, built on top of RDDs, offer a high-level, optimized, and declarative API that's **faster, more efficient, and easier to use** — making them ideal for most modern Spark applications, including large-scale airline delay analysis.

Feature	**RDD**	**DataFrame**
Abstraction Level	Low	High
Schema	No	Yes
Optimization	Manual	Automatic (Catalyst)
Performance	Slower	Faster
Ease of Use	Verbose	Simple
SQL Integration	No	Yes
Type Safety	Yes (Scala)	No (Python)
Ideal For	Complex transformations	Analytical and SQL workloads

Spark’s DAG (Directed Acyclic Graph) Scheduler

DAG (Directed Acyclic Graph):

- Directed → Each operation flows in one direction (from input → output).
- Acyclic → No loops; once an operation finishes, Spark moves forward.
- Graph → Vertices (stages) and edges (data dependencies between stages)

The **DAG Scheduler** is at the heart of Spark’s execution engine.

It transforms user operations (like map(), filter(), join(), groupBy(), etc.) into a logical execution plan — a Directed Acyclic Graph (DAG) — and then optimizes and executes it efficiently across cluster nodes.

Stages of Execution in Spark

Stage	Description
1. Logical Plan	Spark analyzes your code (transformations + actions) and builds a logical DAG of RDD/DataFrame operations.
2. Physical Plan	Catalyst optimizer (for DataFrames) or the RDD DAG scheduler generates a physical execution plan.
3. Task Scheduling	The DAG scheduler divides the physical plan into stages and tasks and submits them to the Task Scheduler.

How the DAG Scheduler Works

Example Workflow

Consider:

```
df = spark.read.csv("airline_data.csv", header=True, inferSchema=True)
result = df.filter(df["ArrDelay"] > 0).groupBy("Carrier").avg("ArrDelay")
result.show()
```

Step-by-step:

1. **User Code → Logical Plan**
    1. Spark constructs a logical DAG:
    2. `Read → Filter → GroupBy → Aggregate → Show`
2. **Optimization**
    1. The **Catalyst optimizer** (for DataFrames) or DAG Scheduler (for RDDs) reorders and combines operations for efficiency.
    2. Example: **Predicate pushdown** — filter is applied before reading all data.
3. **Stage Division**
    1. The DAG is divided into **stages** based on **shuffle boundaries**.
    2. Example:
        1. Stage 1: `Read → Filter`
        2. Stage 2: `GroupBy → Aggregate`

4. Task Generation

1. Each stage is split into multiple **tasks**, one per data partition.
2. If you have 200 partitions, Stage 1 has 200 tasks.

5. Cluster Execution

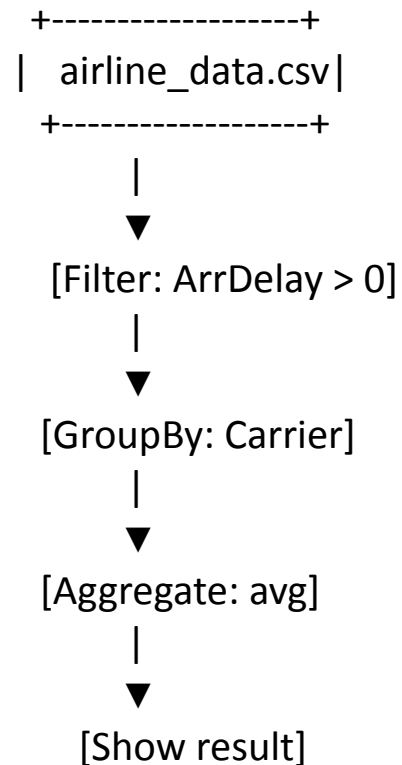
3. Tasks are distributed across **executors** on cluster nodes.
4. The **Task Scheduler** assigns tasks close to data (data locality optimization).

6. Result Assembly

5. Intermediate data (e.g., grouped results) are shuffled and aggregated.
6. Final output is sent to the driver (for `show()` or `collect()`).

DAG Example Visualization

Stage 1 → Read + Filter
Stage 2 → GroupBy + Aggregate + Show



Spark's **DAG Scheduler** is the core of its high performance and scalability.
By transforming a user's program into an optimized **DAG of stages and tasks**, it minimizes data movement, maximizes parallelism, and efficiently utilizes cluster resources — achieving **faster, fault-tolerant, and optimized execution** compared to traditional systems like Hadoop MapReduce.