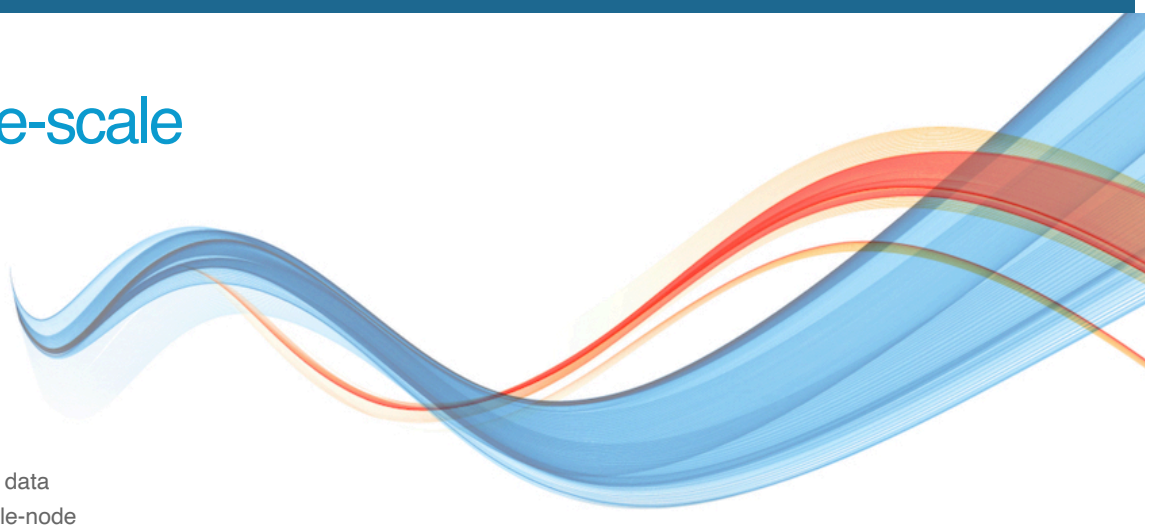
Unified engine for large-scale data analytics

GET STARTED



What is Apache Spark™?

Apache Spark™ is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters.

Simple.
Fast.
Scalable.
Unified.

Key features

Batch/streaming data

Unify the processing of your data in batches and real-time streaming, using your preferred language: Python, SQL, Scala, Java or R.

SQL analytics

Execute fast, distributed ANSI SQL queries for dashboarding and ad-hoc reporting. Runs faster than most data warehouses.

Data science at scale

Perform Exploratory Data Analysis (EDA) on petabyte-scale data without having to resort to downsampling

Machine learning

Train machine learning algorithms on a laptop and use the same code to scale to fault-tolerant clusters of thousands of machines.

Apache Spark

Unified engine for large-scale data analytics

What is Apache Spark?

Apache Spark™ is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters.

Key features

• Batch/streaming data

Unify the processing of your data in batches and real-time streaming, using your preferred language: Python, SQL, Scala, Java or R.

• SQL analytics

Execute fast, distributed ANSI SQL queries for dashboarding and ad-hoc reporting. Runs faster than most data warehouses.

• Data science at scale

Perform Exploratory Data Analysis (EDA) on petabyte-scale data without having to resort to downsampling.

• Machine learning

Train machine learning algorithms on a laptop and use the same code to scale to fault-tolerant clusters of thousands of machines.

Apache Spark™ FAQ

How does Spark relate to Apache Hadoop?

Spark is a fast and general processing engine compatible with Hadoop data. It can run in Hadoop clusters through YARN or Spark's standalone mode, and it can process data in HDFS, HBase, Cassandra, Hive, and any Hadoop InputFormat. It is designed to perform both batch processing (similar to MapReduce) and new workloads like streaming, interactive queries, and machine learning.

How large a cluster can Spark scale to?

Many organizations run Spark on clusters of thousands of nodes. The largest cluster we know has 8000 of them. In terms of data size, Spark has been shown to work well up to petabytes. It has been used to sort 100 TB of data 3X faster than Hadoop MapReduce on 1/10th of the machines, winning the 2014 Daytona GraySort Benchmark, as well as to sort 1 PB. Several production workloads use Spark to do ETL and data analysis on PBs of data.

Does my data need to fit in memory to use Spark?

No. Spark's operators spill data to disk if it does not fit in memory, allowing it to run well on any sized data. Likewise, cached datasets that do not fit in memory are either spilled to disk or recomputed on the fly when needed, as determined by the RDD's storage level.

How can I run Spark on a cluster?

You can use either the standalone deploy mode, which only needs Java to be installed on each node, or the Mesos and YARN cluster managers. If you'd like to run on Amazon EC2, AMPLab provides EC2 scripts to automatically launch a cluster.

Note that you can also run Spark locally (possibly on multiple cores) without any special setup by just passing local[N] as the master URL, where N is the number of parallel threads you want.

Do I need Hadoop to run Spark?

No, but if you run on a cluster, you will need some form of shared file system (for example, NFS mounted at the same path on each node). If you have this type of filesystem, you can just deploy Spark in standalone mode.

Does Spark require modified versions of Scala or Python?

No. Spark requires no changes to Scala or compiler plugins. The Python API uses the standard CPython implementation, and can call into existing C libraries for Python such as NumPy.

What's the difference between Spark Streaming and Spark Structured Streaming? What should I use?

Spark Streaming is the previous generation of Spark's streaming engine. There are no longer updates to Spark Streaming and it's a legacy project. Spark Streaming provides a high-level abstraction called discretized stream or DStream, which represents a continuous stream of data. Internally, a DStream is represented as a sequence of RDDs.

Spark Structured Streaming is the current generation of Spark's streaming engine, which is richer in functionality, easier to use, and more scalable. Spark Structured Streaming is built on top of the Spark SQL engine and enables you to express streaming computation the same way you express a batch computation on static data.

You should use Spark Structured Streaming for building streaming applications and pipelines with Spark. If you have legacy applications and pipelines built on Spark Streaming, you should migrate them to Spark Structured Streaming.

Apache Spark™ examples

This page shows you how to use different Apache Spark APIs with simple examples.

Spark is a great engine for small and large datasets. It can be used with single-node/localhost environments, or distributed clusters. Spark's expansive API, excellent performance, and flexibility make it a good option for many analyses. This guide shows examples with the following Spark APIs:

- DataFrames
- SQL
- Structured Streaming
- RDDs

The examples use small datasets so the they are easy to follow.

Spark DataFrame example

This section shows you how to create a Spark DataFrame and run simple operations. The examples are on a small DataFrame, so you can easily see the functionality.

Let's start by creating a Spark Session:

```
from pyspark.sql import SparkSession


spark = SparkSession.builder.appName("demo").getOrCreate()
```

Some Spark runtime environments come with pre-instantiated Spark Sessions.
The getOrCreate() method will use an existing Spark Session or create a new Spark Session if one does not already exist.

Create a Spark DataFrame

Start by creating a DataFrame with first_name and age columns and four rows of data:

```
df = spark.createDataFrame(
    [
        ("sue", 32),
        ("li", 3),
        ("bob", 75),
        ("heo", 13),
    ],
```

```
    ["first_name", "age"],
)
```

Use the show() method to view the contents of the DataFrame:

```
df.show()


+----------+---+
|first_name|age|
+----------+---+
|      sue| 32|
|       li|  3|
|      bob| 75|
|      heo| 13|
+----------+---+
```

Now, let's perform some data processing operations on the DataFrame.

Add a column to a Spark DataFrame

Let's add a life_stage column to the DataFrame that returns "child" if the age is 12 or under, "teenager" if the age is between 13 and 19, and "adult" if the age is 20 or older.

```
from pyspark.sql.functions import col, when


df1 = df.withColumn(
    "life_stage",
    when(col("age") < 13, "child")
    .when(col("age").between(13, 19), "teenager")
    .otherwise("adult"),
)
```

It's easy to add columns to a Spark DataFrame. Let's view the contents of df1.

```
df1.show()
```

```
+----------+---+----------+
|first_name|age|life_stage|
+----------+---+----------+
|      sue| 32|     adult|
|       li|  3|     child|
|      bob| 75|     adult|
|      heo| 13|  teenager|
+----------+---+----------+
```

Notice how the original DataFrame is unchanged:

```
df.show()


+----------+---+
|first_name|age|
+----------+---+
|      sue| 32|
|       li|  3|
|      bob| 75|
|      heo| 13|
+----------+---+
```

Spark operations don't mutate the DataFrame. You must assign the result to a new variable to access the DataFrame changes for subsequent operations.

Filter a Spark DataFrame

Now, filter the DataFrame so it only includes teenagers and adults.

```
df1.where(col("life_stage").isin(["teenager", "adult"])).show()


+----------+---+----------+
|first_name|age|life_stage|
```

```
+----------+---+----------+
|      sue| 32|     adult|
|      bob| 75|     adult|
|      heo| 13|  teenager|
+----------+---+----------+
```

Group by aggregation on Spark DataFrame

Now, let's compute the average age for everyone in the dataset:

```
from pyspark.sql.functions import avg


df1.select(avg("age")).show()


+--------+
|avg(age)|
+--------+
|   30.75|
+--------+
```

You can also compute the average age for each life_stage:

```
df1.groupBy("life_stage").avg().show()


+----------+--------+
|life_stage|avg(age)|
+----------+--------+
|     adult|    53.5|
|     child|     3.0|
|  teenager|    13.0|
+----------+--------+
```

Spark lets you run queries on DataFrames with SQL if you don't want to use the programmatic APIs.

Query the DataFrame with SQL

Here's how you can compute the average age for everyone with SQL:

```
spark.sql("select avg(age) from {df1}", df1=df1).show()


+--------+
|avg(age)|
+--------+
|   30.75|
+--------+
```

And here's how to compute the average age by life_stage with SQL:

```
spark.sql("select life_stage, avg(age) from {df1} group by life_stage", df1=df1).show()


+----------+--------+
|life_stage|avg(age)|
+----------+--------+
|     adult|    53.5|
|     child|     3.0|
|  teenager|    13.0|
+----------+--------+
```

Spark lets you use the programmatic API, the SQL API, or a combination of both. This flexibility makes Spark accessible to a variety of users and powerfully expressive.

Spark SQL Example

Let's persist the DataFrame in a named Parquet table that is easily accessible via the SQL API.

```
df1.write.saveAsTable("some_people")
```

Make sure that the table is accessible via the table name:

```
spark.sql("select * from some_people").show()
```

```
+----------+---+----------+
|first_name|age|life_stage|
+----------+---+----------+
|      heo| 13|  teenager|
|      sue| 32|     adult|
|      bob| 75|     adult|
|       li|  3|     child|
+----------+---+----------+
```

Now, let's use SQL to insert a few more rows of data into the table:

```
spark.sql("INSERT INTO some_people VALUES ('frank', 4, 'child')")
```

Inspect the table contents to confirm the row was inserted:

```
spark.sql("select * from some_people").show()


+----------+---+----------+
|first_name|age|life_stage|
+----------+---+----------+
|      heo| 13|  teenager|
|      sue| 32|     adult|
|      bob| 75|     adult|
|       li|  3|     child|
|    frank|  4|     child|
+----------+---+----------+
```

Run a query that returns the teenagers:

```
spark.sql("select * from some_people where life_stage='teenager'").show()


+----------+---+----------+
|first_name|age|life_stage|
```

```
+----------+---+----------+
|      heo| 13|  teenager|
+----------+---+----------+
```

Spark makes it easy to register tables and query them with pure SQL.

Spark Structured Streaming Example

Spark also has Structured Streaming APIs that allow you to create batch or real-time streaming applications.

Let's see how to use Spark Structured Streaming to read data from Kafka and write it to a Parquet table hourly.

Suppose you have a Kafka stream that's continuously populated with the following data:

```
{"student_name":"someXXperson", "graduation_year":"2023", "major":"math"}
{"student_name":"liXXyao", "graduation_year":"2025", "major":"physics"}
```

Here's how to read the Kafka source into a Spark DataFrame:

```
df = (
   spark.readStream.format("kafka")
   .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
   .option("subscribe", subscribeTopic)
   .load()
)
```

Create a function that cleans the input data.

```
schema = StructType([
 StructField("student_name", StringType()),
 StructField("graduation_year", StringType()),
 StructField("major", StringType()),
])


def with_normalized_names(df, schema):
```

```
    parsed_df = (
        df.withColumn("json_data", from_json(col("value").cast("string"), schema))
        .withColumn("student_name", col("json_data.student_name"))
        .withColumn("graduation_year", col("json_data.graduation_year"))
        .withColumn("major", col("json_data.major"))
        .drop(col("json_data"))
        .drop(col("value"))
    )
    split_col = split(parsed_df["student_name"], "XX")
    return (
        parsed_df.withColumn("first_name", split_col.getItem(0))
        .withColumn("last_name", split_col.getItem(1))
        .drop("student_name")
    )
```

Now, create a function that will read all of the new data in Kafka whenever it's run.

```
def perform_available_now_update():
    checkpointPath = "data/tmp_students_checkpoint/"
    path = "data/tmp_students"
    return df.transform(lambda df: with_normalized_names(df)).writeStream.trigger(
        availableNow=True
    ).format("parquet").option("checkpointLocation", checkpointPath).start(path)
```

Invoke the perform_available_now_update() function and see the contents of the Parquet table.

You can set up a cron job to run the perform_available_now_update() function every hour so your Parquet table is regularly updated.

Spark RDD Example

The Spark RDD APIs are suitable for unstructured data.

The Spark DataFrame API is easier and more performant for structured data.

Suppose you have a text file called some_text.txt with the following three lines of data:

```
these are words
these are more words
words in english
```

You would like to compute the count of each word in the text file. Here is how to perform this computation with Spark RDDs:

```
text_file = spark.sparkContext.textFile("some_words.txt")


counts = (
   text_file.flatMap(lambda line: line.split(" "))
   .map(lambda word: (word, 1))
   .reduceByKey(lambda a, b: a + b)
)
```

Let's take a look at the result:

```
counts.collect()


[('these', 2),
 ('are', 2),
 ('more', 1),
 ('in', 1),
 ('words', 3),
 ('english', 1)]
```

Spark allows for efficient execution of the query because it parallelizes this computation. Many other query engines aren't capable of parallelizing computations.

Conclusion

These examples have shown how Spark provides nice user APIs for computations on small datasets. Spark can scale these same code examples to large datasets on distributed clusters. It's fantastic how Spark can handle both large and small datasets.

Spark also has an expansive API compared with other query engines. Spark allows you to perform DataFrame operations with programmatic APIs, write SQL, perform streaming analyses, and do machine learning. Spark saves you from learning multiple frameworks and patching together various libraries to perform an analysis.