

ACCELERATING APACHE SPARK 3

Leveraging NVIDIA GPUs
to Power the Next Era
of Analytics and AI

Carol McDonald with contributions from NVIDIA



Table of Contents

Preface: Why are GPUs Driving the Next Wave of Data Science?	4
The Evolution of Data Analytics	4
The Beginning of Big Data Processing	4
Apache Spark Makes Big Data Processing Faster and Easier	6
GPUs Accelerate Computer Processing	7
GPU-Accelerated Data Science Powered by RAPIDS	8
Boosting Data Science Frameworks with GPUs and the RAPIDS Library	10
NVIDIA GPUs in Action	11
Chapter 1: Introduction to Spark Processing	12
How Spark Executes on a Cluster	13
Creating a DataFrame from a File	13
DataFrame Transformations and Actions	16
DataFrame Transformation Narrow and Wide Dependencies	18
How a Spark Application Executes	20
Viewing the Physical Plan	21
Executing the Tasks on a Cluster	28
Summary	30
Chapter 2: Spark SQL and DataFrame Programming	31
DataFrames and Spark SQL Advantages	31
Optimized Memory Usage	31
Query Optimization	32
Exploring the Taxi Dataset with Spark SQL	32
Load the Data from a File into a DataFrame and Cache	32
Using Spark SQL	33
Using Spark Web UI to Monitor Spark SQL	36
SQL Tab	36
Jobs Tab	37
Stages Tab	37
Storage Tab	37
Executors Tab	38
Partitioning and Bucketing	38
Summary	39
Chapter 3: GPU-Accelerated Apache Spark 3.x	40
Spark 3.x and GPUs	40
Accelerated ETL and AI in Spark	40
New GPU-Accelerated Libraries on NVIDIA CUDA	41

RAPIDS GPU-Accelerated Spark DataFrames	41
Spark GPU-Accelerated DataFrame and SQL	42
GPU-Aware Scheduling in Spark	44
Stage Level Scheduling in Spark	44
Using the Spark Web UI to Monitor GPU Resources	46
XGBoost, RAPIDS, and Spark	47
Other Spark 3.x Features	48
Spark 3.x CPU vs. GPU Performance Comparisons	50
Task Level Parallelism vs Data Level Parallelism	52
Summary	53
Chapter 4: Getting Started with GPU-Accelerated Apache Spark 3	54
Accelerated Spark Platforms	55
Databricks	55
Google	56
AWS	58
Cloudera	59
CDP Powered by NVIDIA Accelerated Data Science	60
Microsoft	61
Alluxio	62
Configuration	65
GPU Scheduling	65
Advanced Configuration	66
Tuning General Recommendations	66
Number of Executors	66
Number of Tasks per Executor	66
Input Files	67
Input Partition Size	67
Monitoring Using the Physical Plan	67
Monitoring Using the Spark Web UI	68
SQL Tab	68
Stages Tab	70
Environment Tab	71
Executors Tab	71
Summary	71

Chapter 5: Predicting Housing Prices Using Apache Spark Machine Learning	72
Classification and Regression	72
Regression	73
Decision Trees	74
Random Forests	75
Machine Learning Workflows	76
Using Spark ML Pipelines	76
Example Use Case Dataset	78
Load the Data from a File into a DataFrame	78
Summary Statistics	80
Feature Extraction and Pipelining	81
Train the Model	83
Predictions and Model Evaluation	86
Save the Model	89
Summary	89
Chapter 6: Predicting Taxi Fares Using GPU-Accelerated XGBoost	90
XGBoost	90
GPU-Accelerated XGBoost	91
Example Use Case Dataset	91
Load the Data from a File into a DataFrame	92
Define Features Array	94
Save the Model	98
Summary	98
Chapter 7: Real-world Examples of Accelerating End-to-End Machine Learning Pipelines	99
Adobe: Accelerated End-to-End Customer AI	99
GPU-Accelerated, End-to-End ML	100
AWS: Accelerating Deep Learning on the JVM with Apache Spark and NVIDIA GPUs	102
Uber: Accelerated End-to-End ETL and DL	103
Summary	104
Appendix: Code	105
Code	105
Additional Resources	105
About the Author	105
NVIDIA Contributors	106

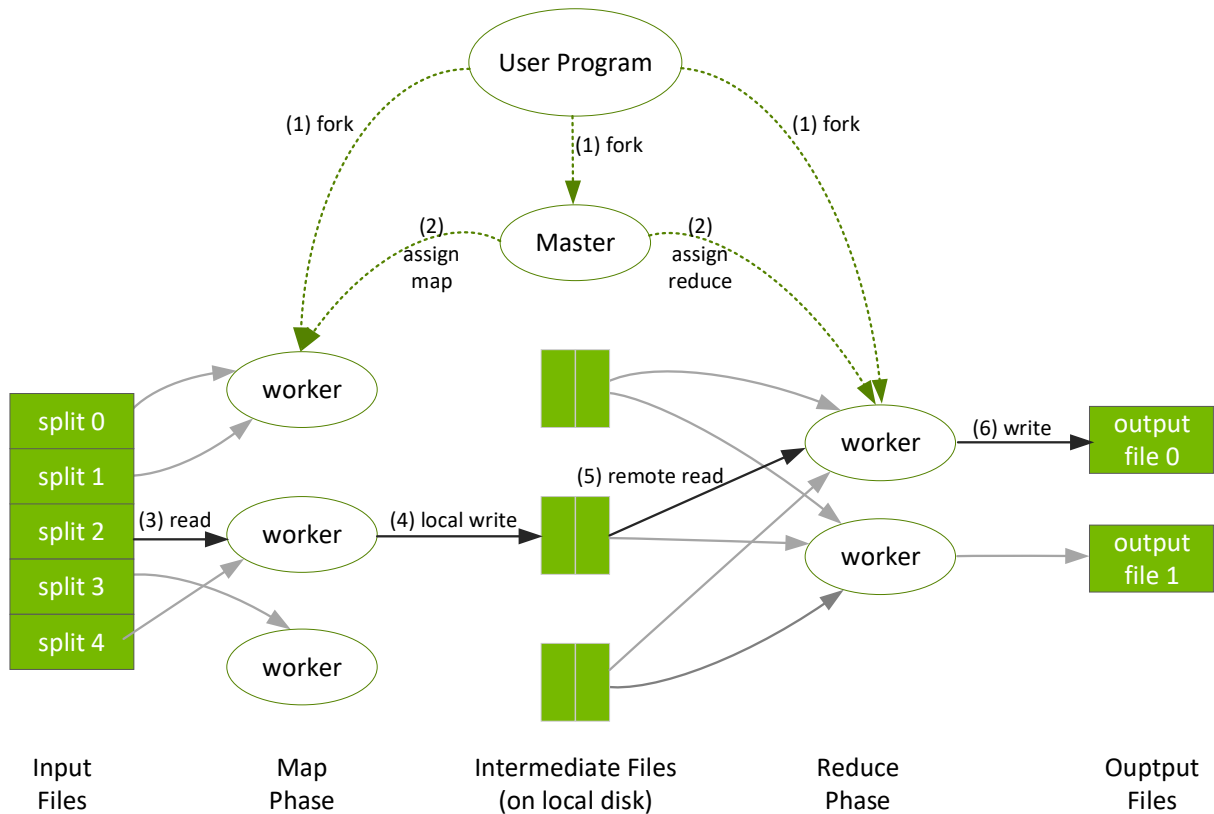
Preface: Why are GPUs Driving the Next Wave of Data Science?

The Evolution of Data Analytics

According to Thomas Davenport in the updated version of *Competing on Analytics*, analytical technology has changed dramatically over the last decade, with more powerful and less expensive distributed computing across commodity servers, and improved machine learning (ML) technologies, enabling companies to store and analyze many different types of data and far more of it.

The Beginning of Big Data Processing

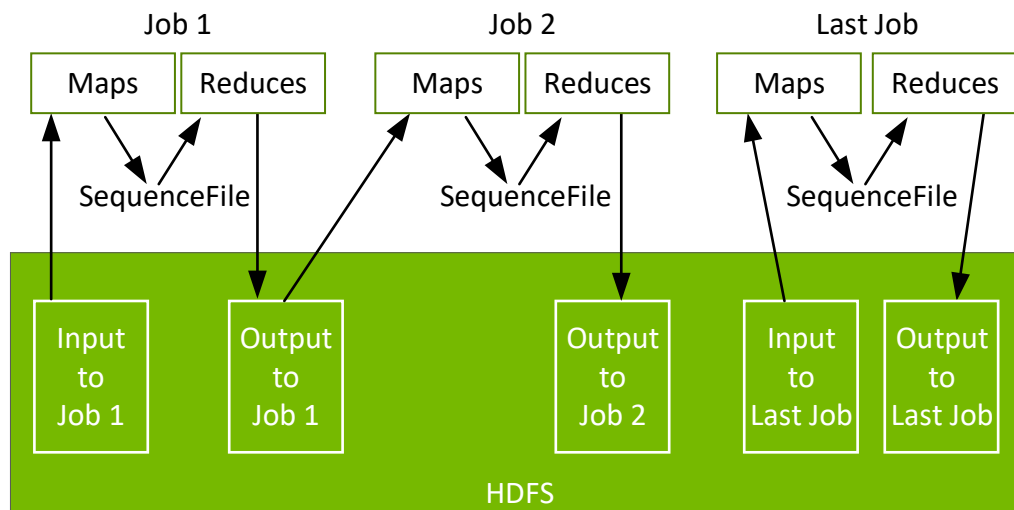
Google invented a distributed file system and MapReduce, a resilient distributed processing framework, in order to index the exploding volume of content on the web, across large clusters of commodity servers. The Google file system (GFS) partitioned, distributed, and replicated file data across the data nodes in a cluster. MapReduce distributed computation across the data nodes in a cluster: users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs and a reduce function that merges all intermediate values associated with the same intermediate key. Both the GFS and MapReduce were designed for fault tolerance by failing over to another node for data or processing.



Reference: [MapReduce Google White Paper](#)

A year after Google published a [white paper describing the MapReduce](#) framework, Doug Cutting and Mike Cafarella created [Apache Hadoop™](#).

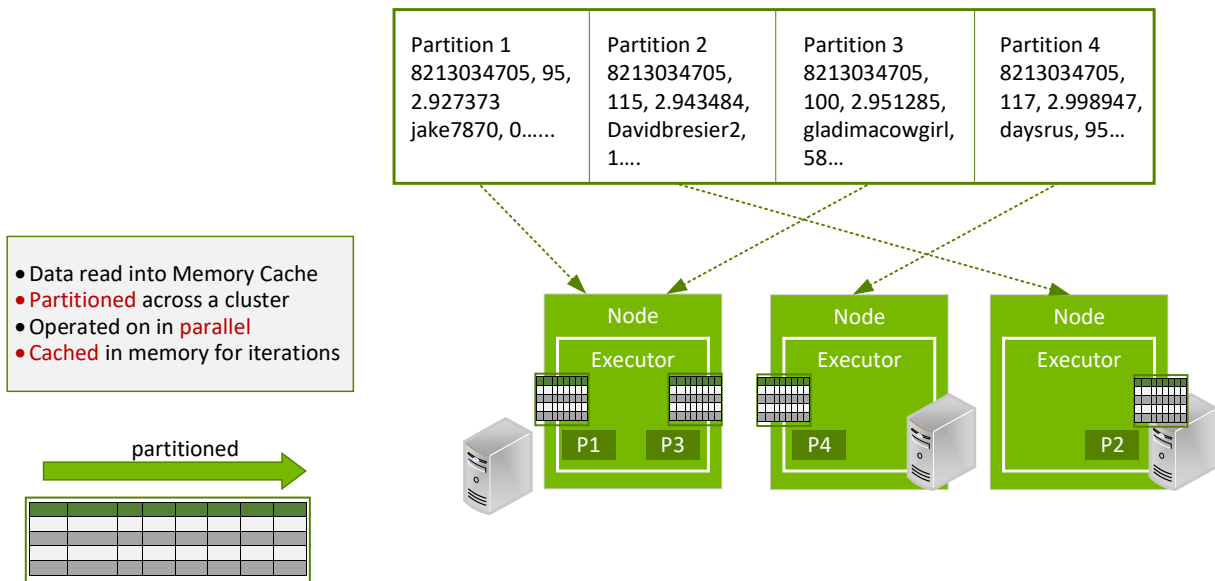
However, Hadoop performance is bottlenecked by its model of checkpointing results to disk. At the same time, Hadoop adoption has been hindered by the low-level programming model of MapReduce. Data pipelines and iterative algorithms require chaining multiple MapReduce jobs together, which can be difficult to program and cause a lot of reading and writing to disk.



Apache Spark Makes Big Data Processing Faster and Easier

Apache Spark started as a [research project](#) at UC Berkeley in the AMPLab, became a top level [Apache Software Foundation](#) project in 2014, and is now maintained by a community of hundreds of developers from hundreds of organizations. Spark was developed with the goal of keeping the benefits of MapReduce's scalable, distributed, fault-tolerant processing framework, while making it more efficient and easier to use. Spark is more efficient than MapReduce for data pipelines and iterative algorithms because it caches data in memory across iterations and uses lighter weight threads. Spark also provides a richer functional programming model than MapReduce.

Distributed Datasheet

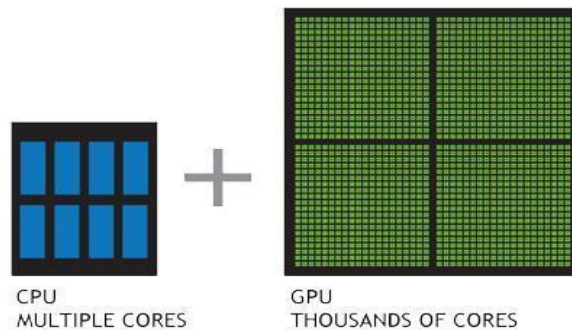


Spark mitigated the I/O problems found in Hadoop, but now the bottleneck has shifted from I/O to compute for a growing number of applications. This performance bottleneck has been thwarted with the advent of GPU-accelerated computation.

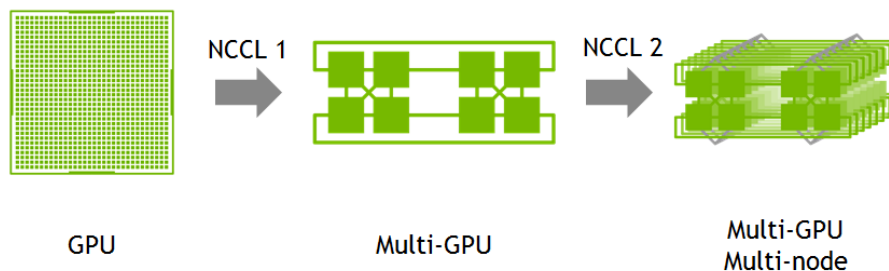
GPUs Accelerate Computer Processing

Graphics Processing Units (GPUs) are popular for their extraordinarily low price per flop (performance) and are addressing the compute performance bottleneck today by speeding up multi-core servers for parallel processing.

A CPU consists of a few cores, optimized for sequential serial processing. Whereas, a GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously. GPUs are capable of processing data much faster than configurations containing CPUs alone.



Once large amounts of data need to be broadcasted, aggregated, or collected across nodes in a cluster, the network can become a bottleneck. [GPUDirect Remote](#) direct memory access with the [NVIDIA Collective Communications Library](#) can solve this bottleneck by allowing GPUs to communicate directly with each other, across nodes, for faster multi-GPU and multi-node reduction operations.



The benefits of GPUDirect RDMA are also critical for large, complex extract, transform, load (ETL) workloads, allowing them to operate as if they were on one massive server.

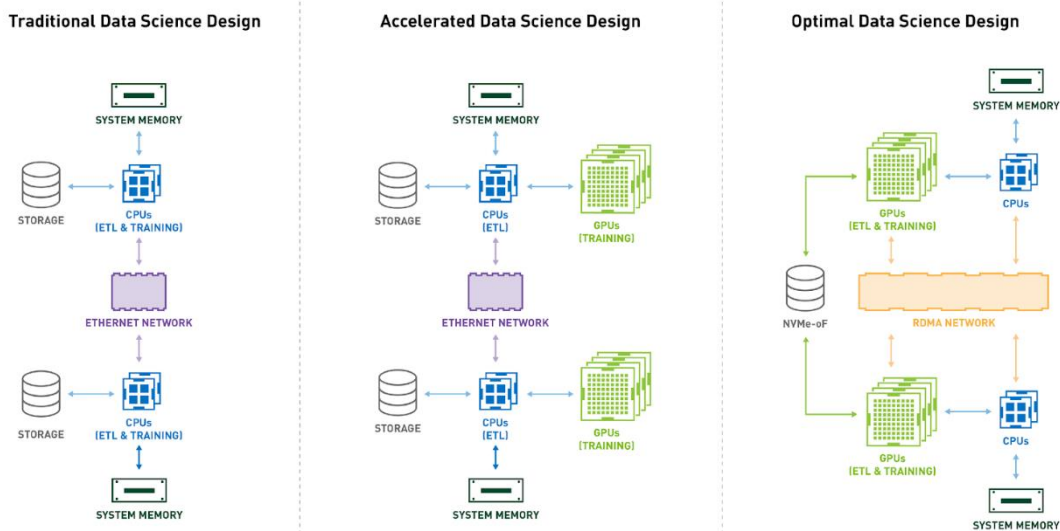


FIGURE 2: MOVING TO A GPU-CENTRIC DATA SCIENCE DESIGN

GPU-Accelerated Data Science Powered by RAPIDS

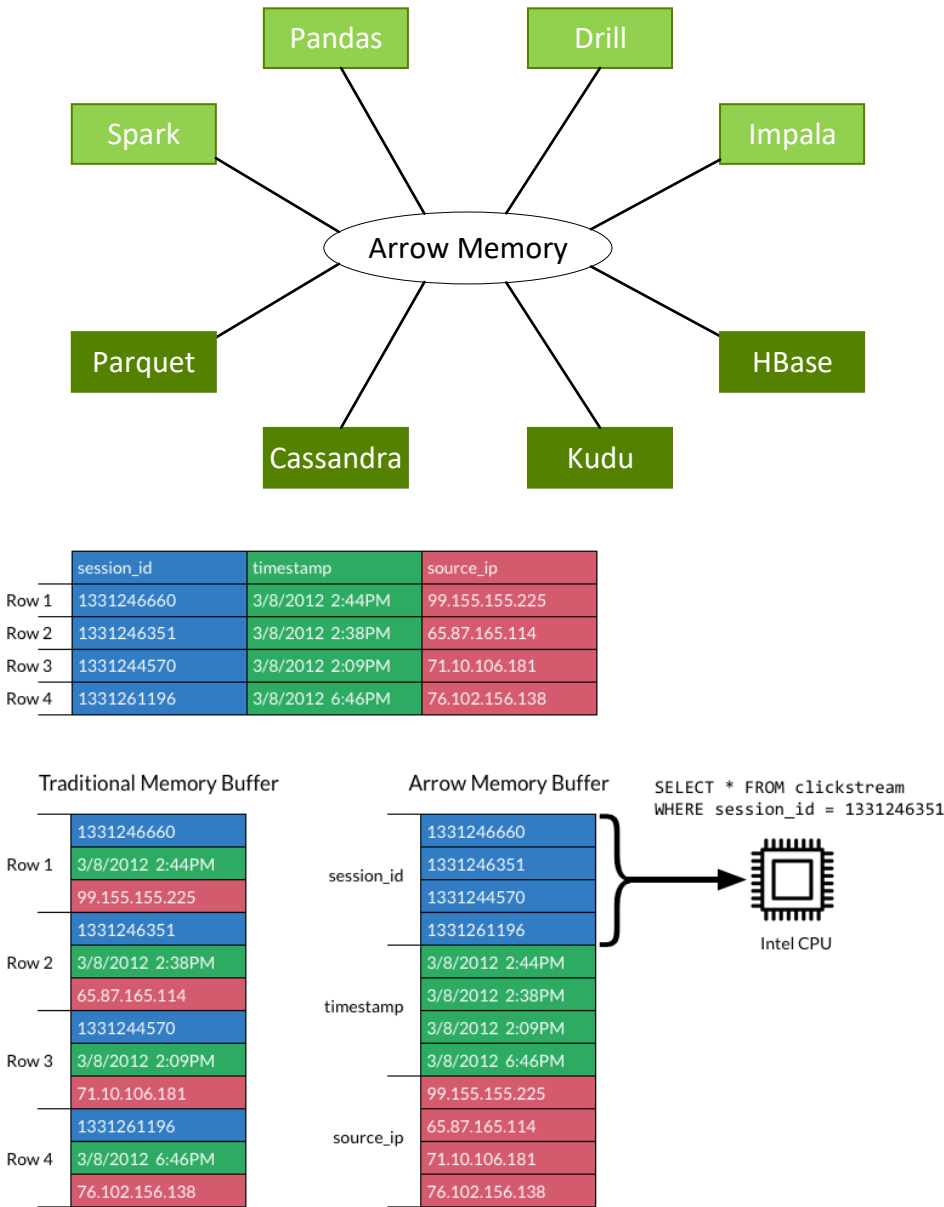
A key component of data science is data exploration. Preparing a data set for ML requires understanding the data set, cleaning and manipulating data types and formats, and extracting features for the learning algorithm. These tasks are grouped under the term ETL. ETL is often an iterative, exploratory process. As data sets grow, the interactivity of this process suffers when running on CPUs.

GPUs have been responsible for the advancement of deep learning (DL) in the past several years, while ETL and traditional ML workloads continued to be written in Python, often with single-threaded tools like Scikit-Learn or large, multi-CPU distributed solutions like Spark.

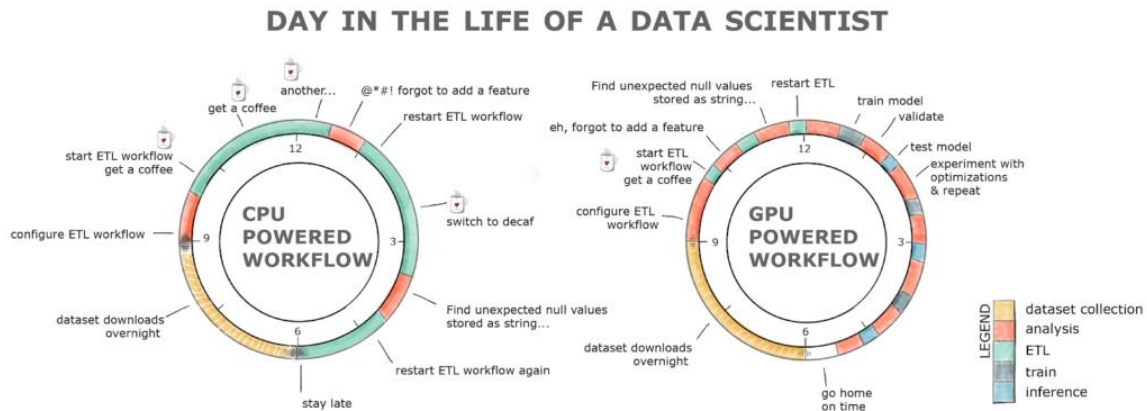
[RAPIDS](#) is a suite of open-source software libraries and APIs for executing end-to-end data science and analytics pipelines entirely on GPUs, achieving speedup factors of 50X or more on typical end-to-end data science workflows. RAPIDS accelerates the entire data science pipeline, including data loading, enabling more productive, interactive, and exploratory workflows.

Built on top of [NVIDIA® CUDA®](#), an architecture and software platform for GPU computing, RAPIDS exposes GPU parallelism and high-bandwidth memory speed through user-friendly APIs. RAPIDS focuses on common data preparation tasks for analytics and data science, offering a powerful GPU DataFrame that is compatible with [ApacheArrow](#) data structures with a familiar DataFrame API.

Apache Arrow specifies a standardized language-independent columnar memory format, optimized for data locality, to accelerate analytical processing performance on modern CPUs or GPUs, and provides zero-copy streaming messaging and interprocess communication without serialization overhead.



The DataFrame API integrates with a variety of ML algorithms without incurring typical serialization and deserialization costs, enabling end-to-end pipeline accelerations.



By hiding the complexities of low-level CUDA programming, RAPIDS creates a simple way to execute data science tasks. As more data scientists use Python and other high-level languages, providing acceleration with minimal to no code change is essential to rapidly improving development time.

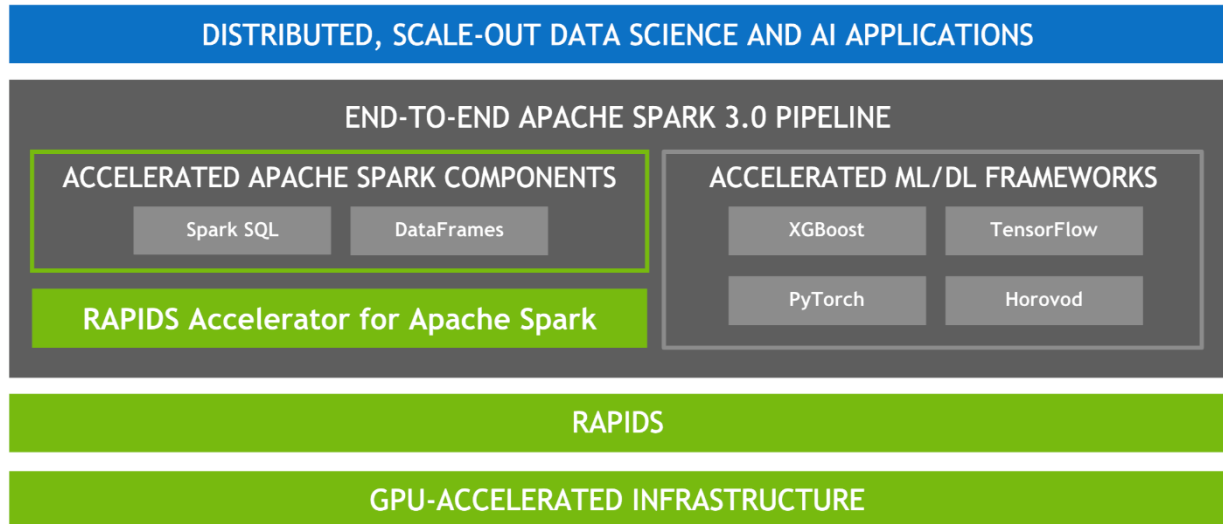
Boosting Data Science Frameworks with GPUs and the RAPIDS Library

Another way RAPIDS accelerates development is with integration to leading data science frameworks, such as [PyTorch](#), [Chainer](#), and [ApacheMxNet](#) for DL and distributed computing frameworks like [Apache Spark](#) and [Dask](#) for seamless scaling from GPU workstations to multi-GPU servers and multi-node clusters. Also, products such as [BlazingSQL](#), an open source SQL engine, are being built on top of RAPIDS, adding even more accelerated capabilities for users.

Apache Spark 3.x empowers GPU applications by providing user APIs and configurations to easily request and utilize GPUs and is now extensible to allow columnar processing on the GPU; all of which wasn't supported prior to Spark 3.x. Internally, Spark added GPU scheduling, further integration with the cluster managers (YARN, Kubernetes, etc.) to request GPUs, and plugin points to allow it to be extended to run operations on the GPU. This makes GPUs easier to request and use for Spark application developers, allows for closer integration with DL and AI frameworks, such as Horovod and TensorFlow on Spark, and allows for better utilization of GPUs. This extensibility also allows columnar processing, which opens up the possibility for users to add plugins that accelerate queries using the GPU.

Later in this eBook, we explore how the Apache Spark 3.x stack shown below accelerates Spark 3.x applications.

APACHE SPARK 3.x GPU-ACCELERATED SOFTWARE STACK



NVIDIA GPUs in Action

Regardless of industry or use case, when putting ML into action, many data science problems break down into similar steps: iteratively preprocessing data to build features, training models with different parameters, and evaluating the model to ensure performance translates into valuable results.

RAPIDS helps accelerate all of these steps while maximizing the user's hardware investments. Early customers have taken full data pipelines that took days, if not weeks, and ran them in minutes. They've simultaneously reduced costs and improved the accuracy of their models since more iterations allow data scientists to explore more models and parameter combinations, as well as train on larger datasets.

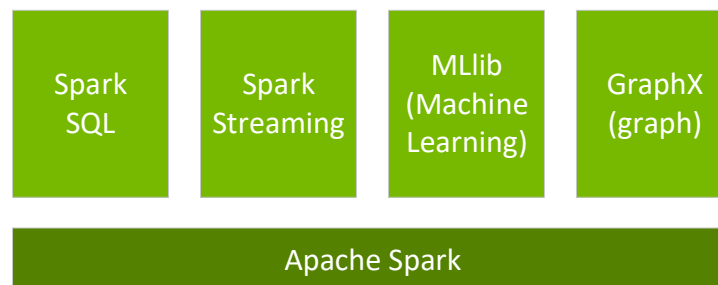
Retailers are improving their forecasting. Finance companies are getting better at assessing credit risk. And adtech firms are enhancing their ability to predict click-through rates. Data scientists often achieve improvements of 1-2 percent. This can translate to tens or hundreds of millions of dollars of revenue and profitability.

Chapter 1: Introduction to Spark Processing

[IDC predicts](#) that data generated in data centers, as well as from edge computing and IOT, will quintuple in the next seven years to 175 ZB. In tandem with the monumental growth of data, Apache Spark from Apache Software Foundation has become one of the most popular frameworks for distributed scale-out data processing, running on millions of servers—both on premises and in the cloud. This chapter provides an introduction to the Spark framework and explains how it executes applications.

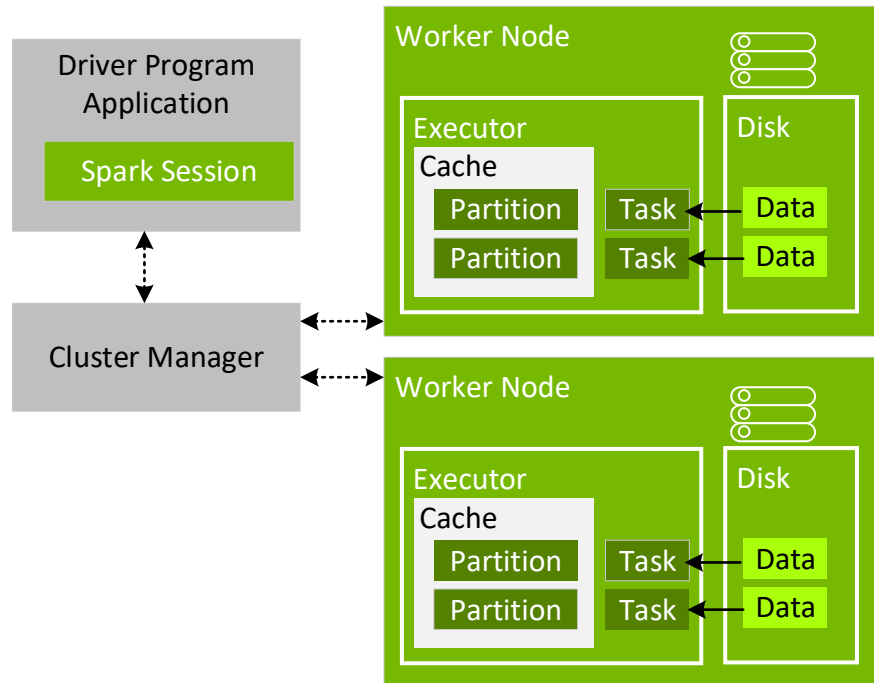
Apache Spark is a fast and general purpose analytics engine for large-scale data processing, that runs on Hadoop, Apache Mesos, Kubernetes, standalone, or in the cloud. Spark offers high-level operators that make it easy to build parallel applications in Scala, Python, R, or SQL, using an interactive shell, notebooks, or packaged applications.

On top of the Spark core data processing engine, there exist libraries for SQL and DataFrames, machine learning, GraphX, graph computation, and stream processing. These libraries can be used together on massive datasets from a variety of data sources, such as HDFS, Alluxio, Apache Cassandra, Apache HBase, or Apache Hive.



How Spark Executes on a Cluster

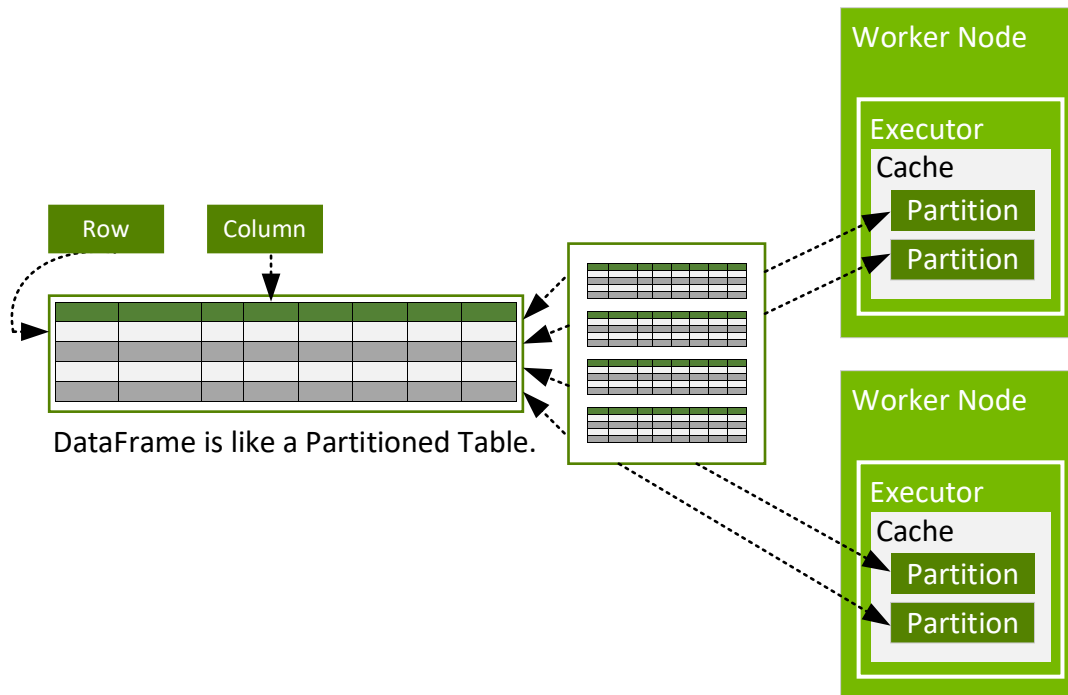
A Spark application runs as parallel tasks inside of executor processes on cluster nodes, with execution coordinated between the `SparkSession` object in the driver program and the Resource or Cluster manager (either Standalone, Mesos, YARN, or Kubernetes) on the cluster.



Spark can also run on a single machine, called local mode. In local mode, the driver program and the tasks run in threads in the same Java Virtual Machine. Local mode is useful for prototyping, development, debugging and testing. However local mode is not meant for running production applications.

Creating a DataFrame from a File

A Spark `DataFrame` is a distributed [Dataset](#) of `org.apache.spark.sql.Row` objects, that are partitioned across multiple nodes in a cluster and can be operated on in parallel. A `DataFrame` represents a table of data with rows and columns, similar to a `DataFrame` in R or Python, but with Spark optimizations. A `DataFrame` consists of partitions, each of which is a range of rows in cache on a data node.



DataFrames can be constructed from data sources, such as csv, parquet, JSON files, Hive tables, or external databases. A DataFrame can be operated on using relational transformations and Spark SQL queries.

The Spark shell or Spark notebooks provide a simple way to use Spark interactively. You can start the shell in local mode with the following command:

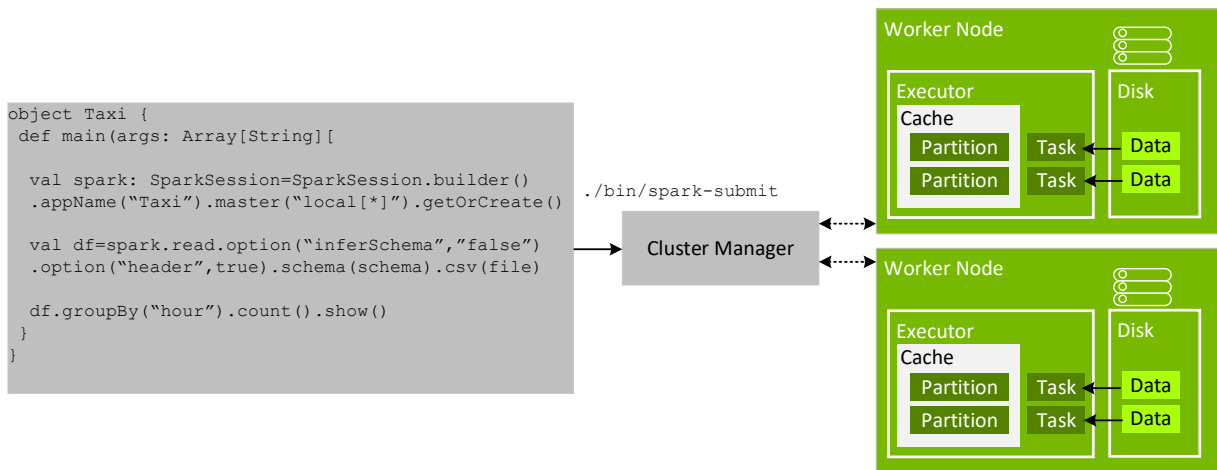
```
$ /[installation path]/bin/spark-shell --master local[2]
```

You can then enter the code from the rest of this chapter into the shell to see the results interactively. In the code examples, the outputs from the shell are prefaced with the result.

For execution coordination between your application driver and the Cluster manager, you create a `SparkSession` object in your program, as shown in the following code example:

```
val spark = SparkSession.builder.appName("Simple
Application").master("local[2]").getOrCreate()
```

When a Spark application starts, it connects to the cluster manager via the master URL. The master URL can be set to the cluster manager or local[N] to run locally with N threads, when creating the SparkSession object or when submitting the Spark application. When using the spark-shell or notebook, the SparkSession object is already created and available as the variable spark. Once connected, the cluster manager allocates resources and launches executor processes, as configured for the nodes in your cluster. When a Spark application executes, the SparkSession sends tasks to the executors to run.



With the SparkSession read method, you can read data from a file into a DataFrame, specifying the file type, file path, and input options for the schema.

```
import org.apache.spark.sql.types._
import org.apache.spark.sql._
import org.apache.spark.sql.functions._

val schema =
  StructType(Array(
    StructField("vendor_id", DoubleType),
    StructField("passenger_count", DoubleType),
    StructField("trip_distance", DoubleType),
    StructField("pickup_longitude", DoubleType),
    StructField("pickup_latitude", DoubleType),
    StructField("rate_code", DoubleType),
    StructField("store_and_fwd", DoubleType),
    StructField("dropoff_longitude", DoubleType),
    StructField("dropoff_latitude", DoubleType),
    StructField("fare_amount", DoubleType),
    StructField("hour", DoubleType),
    StructField("year", IntegerType),
    StructField("month", IntegerType),
    StructField("day", DoubleType),
    StructField("day_of_week", DoubleType),
```



```
    StructField("is_weekend", DoubleType)
  ))

val file = "/data/taxi_small.csv"

val df = spark.read.option("inferSchema", "false")
  .option("header", true).schema(schema).csv(file)

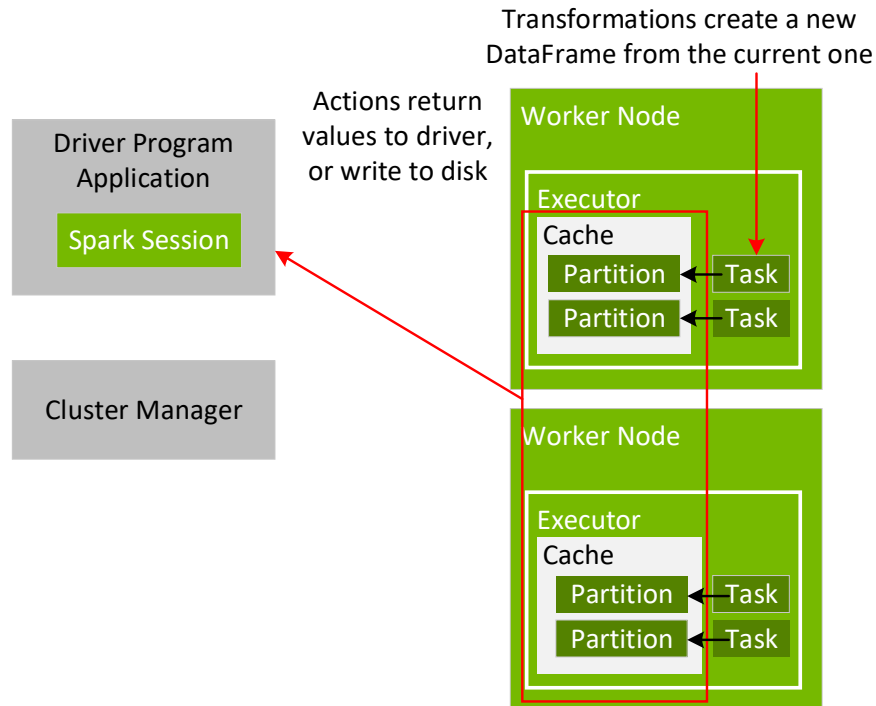
result:
df: org.apache.spark.sql.DataFrame = [vendor_id: double, passenger_count:
double ... 14 more fields]
```

The take method returns an array with objects from this DataFrame, which we see is of the `org.apache.spark.sql.Row` type.

```
df.take(1)
result:
Array[org.apache.spark.sql.Row] = Array([4.52563162E8,5.0,2.72,-
73.948132,40.829826999999995,-6.77418915E8,-1.0,-
73.969648,40.7974720000000006,11.5,10.0,2012,11,13.0,6.0,1.0])
```

DataFrame Transformations and Actions

DataFrames provide a domain-specific language API for structured data processing, known as transformations. Transformations create a new transformed DataFrame from the current DataFrame and are lazily evaluated. Transformations are executed when triggered by an action, which returns a result to the driver program or writes to disk. Once an action has run and the value is returned, the DataFrame is no longer in memory, unless it is cached. Spark can cache DataFrames using an in-memory columnar format by calling [DataFrame.cache\(\)](#).



Here is a list of some commonly used DataFrame transformations.

- `select` Selects a set of columns
- `join` Join with another DataFrame, using the given join expression
- `groupBy` Groups the DataFrame, using the specified columns

This `groupBy` transformation example groups the taxi DataFrame by hour of the day, then the `count` action totals the number of taxi trips for each hour. The `show` action prints out the resulting DataFrame rows in a tabular format.

```
df.groupBy("hour").count().show(4)
```

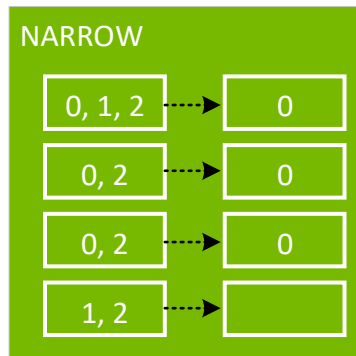
```
result:
+----+-----+
|hour|count|
+----+-----+
| 0.0|   12|
| 1.0|   49|
| 2.0|  658|
| 3.0|  742|
+----+-----+
```

Following is a list of some commonly used DataFrame actions.

- `show(n)` Displays the first `n` rows in a tabular format
- `take(n)` Returns the first `n` rows in the DataFrame in an array
- `count` Returns the number of rows in the DataFrame

DataFrame Transformation Narrow and Wide Dependencies

There are two types of DataFrame transformations, those with narrow dependencies and those with wide dependencies. Transformations with narrow dependencies do not have to move data between partitions when creating a new DataFrame from an existing one. An example narrow transformation is `filter()` which is used to filter the rows from a DataFrame based on the given SQL expression. The following example filters for the hour value = 0.

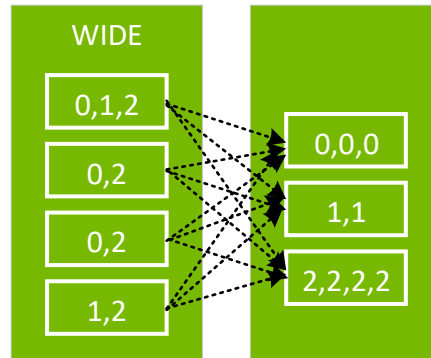


Multiple narrow transformations can be performed on a DataFrame in memory, using a process called pipelining, making narrow transformations very efficient. Narrow transformations like `filter` and `select` are used in the example below to retrieve taxi fare_amounts for the 0 hour of the day.

```
// select and filter are narrow transformations
df.select($"hour", $"fare_amount").filter($"hour" === "0.0" ).show(2)

result:
+----+-----+
|hour|fare_amount|
+----+-----+
| 0.0|      10.5|
| 0.0|      12.5|
+----+-----+
```

Transformations with wide dependencies have to move data between partitions, when creating a new DataFrame from an existing one, in a process called a shuffle. Shuffles send data across the network to other nodes and write to disk, causing network and disk I/O. Example wide transformations are `groupBy`, `agg`, `sortBy`, and `orderBy`. The wide transformation shows groups by the hour value.



Following is a wide transformation to group by the hour value and count the number of taxi trips by hour.

```
df.groupBy("hour").count().show(4)
```

```
result:
+----+-----+
|hour|count|
+----+-----+
| 0.0|   12|
| 1.0|   49|
| 2.0|  658|
| 3.0|  742|
+----+-----+
```

How a Spark Application Executes

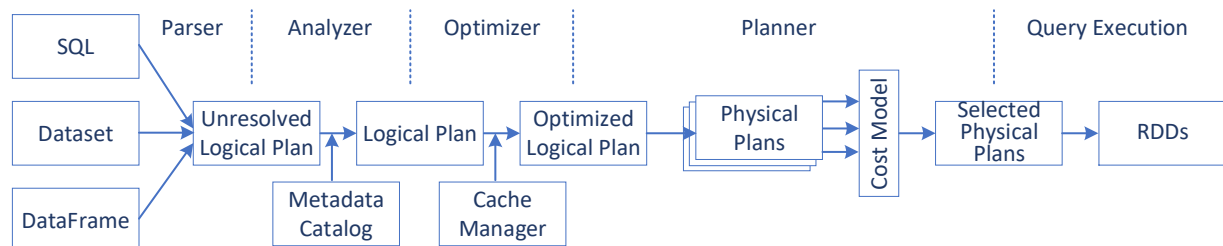
When a Spark query executes, it goes through the following steps:

- ▶ Creates a logical plan
- ▶ Transforms the logical plan to a physical plan
- ▶ Generates code
- ▶ Executes the tasks on a cluster

Apache Spark provides a web UI that you can use to see a visual representation of these plans in the form of Directed Acyclic Graphs (DAGs). With the web UI, you can also see how the plan executes and monitor the status and resource consumption on your Spark cluster. You can view the web UI in real time with this URL: `http://<driver-node>:4040`. You can view the web UI after execution through Spark's history server at `http://<server-url>:18080`, provided that the application's event logs exist.

In the first step, the logical plan is created for the submitted SQL or DataFrame. The logical plan shows the set of abstract transformations that will be executed. The Spark Analyzer uses the Metadata Catalog to resolve tables and columns, then passes the plan to the [Catalyst](#) Optimizer, which uses rules like **filter push down**, to optimize the plan.

Actions trigger the translation of the logical DAG into a physical execution plan. The physical plan identifies resources that will execute the plan, using a cost model for different execution strategies. An example of this would be a **broadcast join** versus a **hash join**.



Reference: Databricks

Viewing the Physical Plan

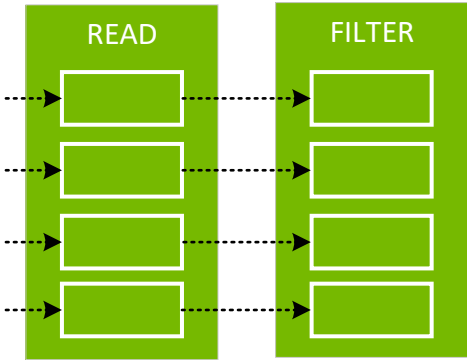
You can see the formatted physical plan for a DataFrame by calling the `explain("formatted")` method. In the physical plan below, the DAG for `df2` consists of a `Scan csv` file, a `Filter` on `day_of_week`, and a `Project` (selecting columns) on `hour`, `fare_amount`, and `day_of_week`.

```
val df = spark.read.option("inferSchema", "false") .option("header",
true).schema(schema).csv(file)
val df2 = df.select($"hour", $"fare_amount",
$"day_of_week").filter($"day_of_week" === "6.0" )
df2.show(3)
result:
+----+-----+-----+
|hour|fare_amount|day_of_week|
+----+-----+-----+
|10.0|      11.5|        6.0|
|10.0|       5.5|        6.0|
|10.0|      13.0|        6.0|
+----+-----+-----+
df2.explain("formatted")
result:
== Physical Plan ==
* Project (3)
+- * Filter (2)
   +- Scan csv   (1)

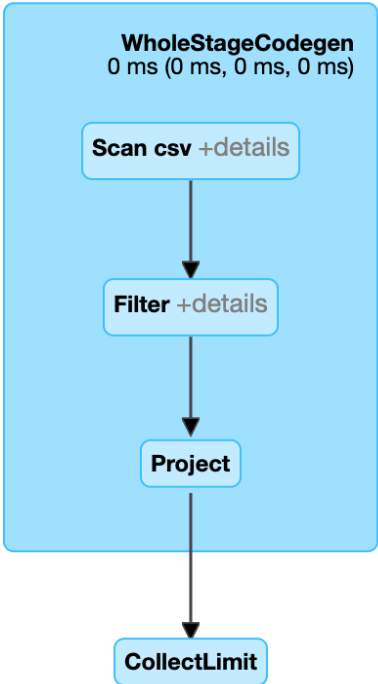
(1) Scan csv
Location: [dbfs:/FileStore/tables/taxi_tsmall.csv]
Output [3]: [fare_amount#143, hour#144, day_of_week#148]
PushedFilters: [IsNotNull(day_of_week), EqualTo(day_of_week,6.0)]

(2) Filter [codegen id : 1]
Input [3]: [fare_amount#143, hour#144, day_of_week#148]
Condition : (isnotnull(day_of_week#148) AND (day_of_week#148 = 6.0))

(3) Project [codegen id : 1]
Output [3]: [hour#144, fare_amount#143, day_of_week#148]
Input [3]: [fare_amount#143, hour#144, day_of_week#148]
```



You can see more details about the plan produced by Catalyst on the web UI SQL tab. Clicking on the query description link displays the DAG and details for the query.

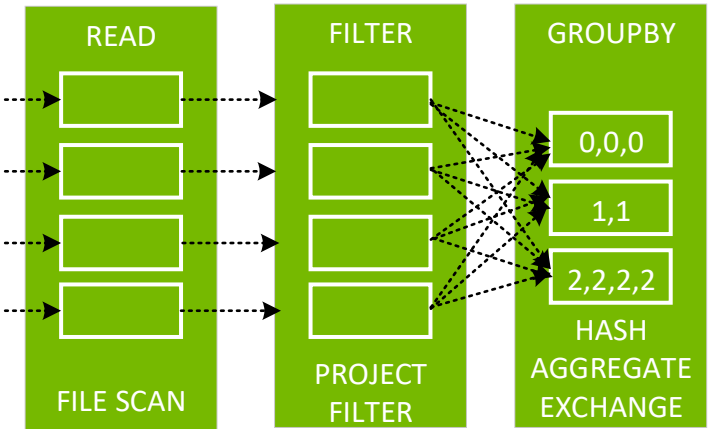


In the following code, after the `explain`, we see that the physical plan for `df3` consists of a `Scan`, `Filter`, `Project`, `HashAggregate`, `Exchange`, and `HashAggregate`. The `Exchange` is the shuffle caused by the `groupBy` transformation. Spark performs a hash aggregation for each partition before shuffling the data in the `Exchange`. After the exchange, there is a hash aggregation of the previous sub-aggregations. Note that we would have an in-memory scan instead of a file scan in this DAG, if `df2` were cached.

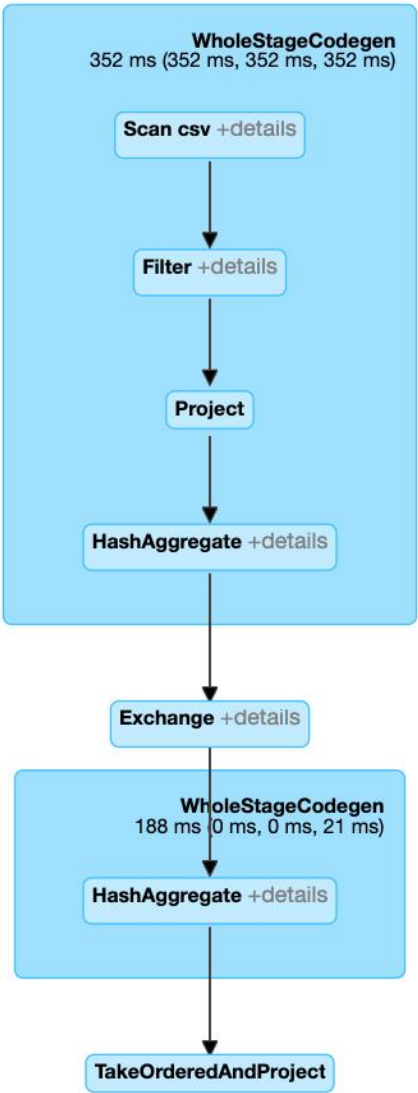
```
val df3 = df2.groupBy("hour").count
df3.orderBy(asc("hour")).show(5)
result:
+-----+-----+
|hour|count|
+-----+-----+
| 0.0|   12|
| 1.0|   47|
| 2.0|  658|
| 3.0|  742|
| 4.0|  812|
+-----+-----+

df3.explain
result:
== Physical Plan ==
* HashAggregate (6)
+- Exchange (5)
   +- * HashAggregate (4)
      +- * Project (3)
         +- * Filter (2)
            +- Scan csv (1)
(1) Scan csv
Output [2]: [hour, day_of_week]
(2) Filter [codegen id : 1]
Input [2]: [hour, day_of_week]
Condition : (isnotnull(day_of_week) AND (day_of_week = 6.0))
(3) Project [codegen id : 1]
Output [1]: [hour]
Input [2]: [hour, day_of_week]
(4) HashAggregate [codegen id : 1]
Input [1]: [hour]
Functions [1]: [partial_count(1) AS count]
Aggregate Attributes [1]: [count]
Results [2]: [hour, count]
(5) Exchange
Input [2]: [hour, count]
Arguments: hashpartitioning(hour, 200), true, [id=]
(6) HashAggregate [codegen id : 2]
```

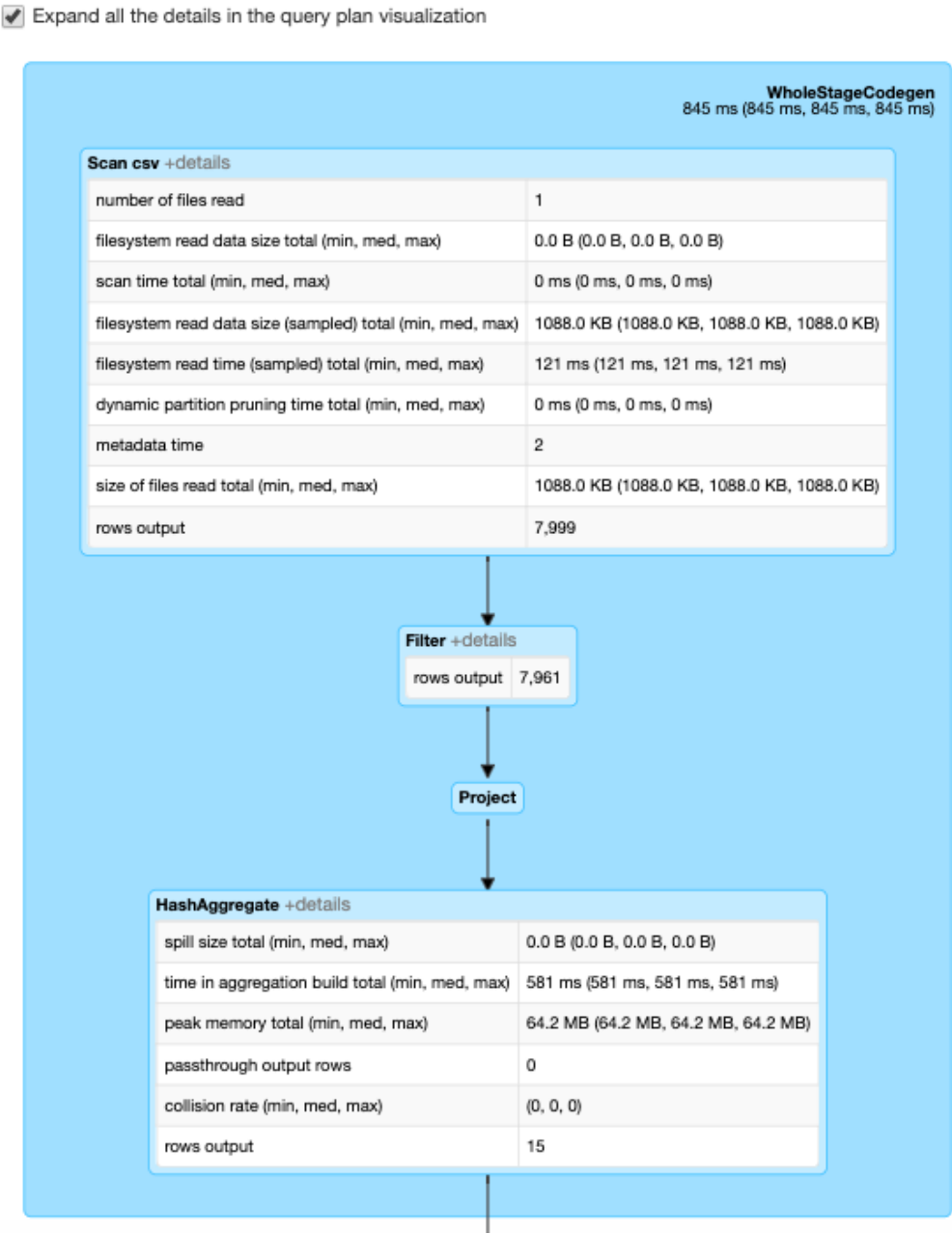
```
Input [2]: [hour, count]
Keys [1]: [hour]
Functions [1]: [finalmerge_count(merge count) AS count(1)]
Aggregate Attributes [1]: [count(1)]
Results [2]: [hour, count(1) AS count]
```



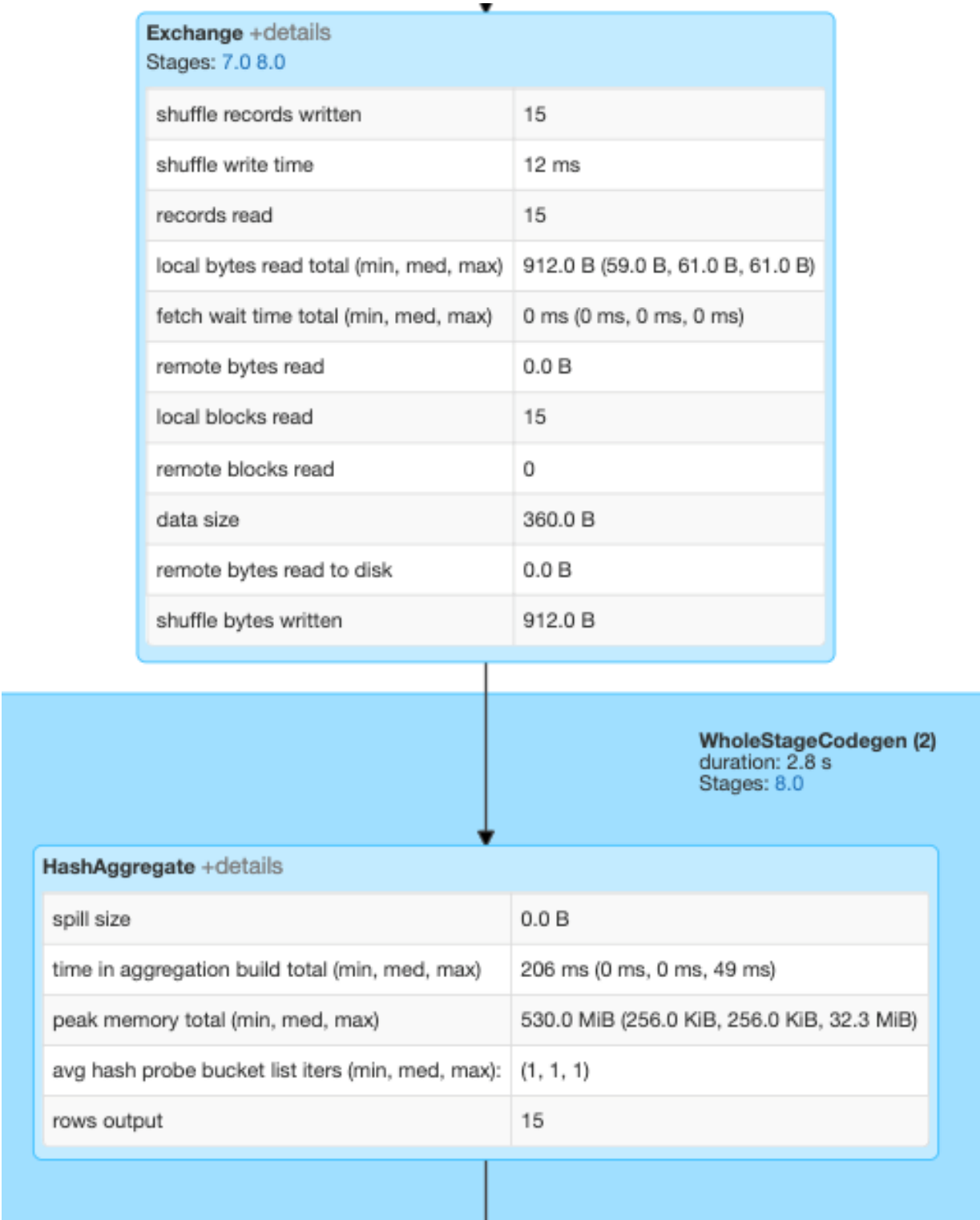
Clicking on the **SQL** tab link for this query display the DAG of the job.



Selecting the **Expand details** checkbox shows detailed information for each stage. The first block **WholeStageCodegen** compiles multiple operators (scan csv, filter, project, and HashAggregate) together into a single Java function to improve performance. Metrics such as number of rows and spill size are shown in the following screen.

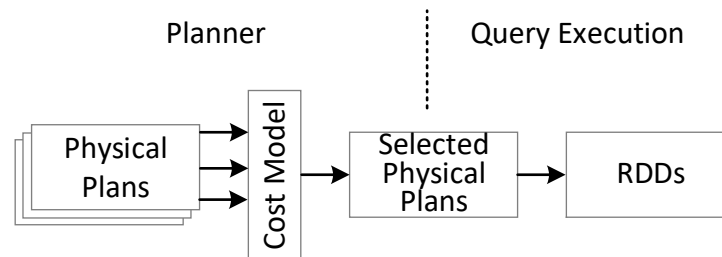


The second block entitled **Exchange** shows the metrics on the shuffle exchange, including the number of written shuffle records and the data size total.

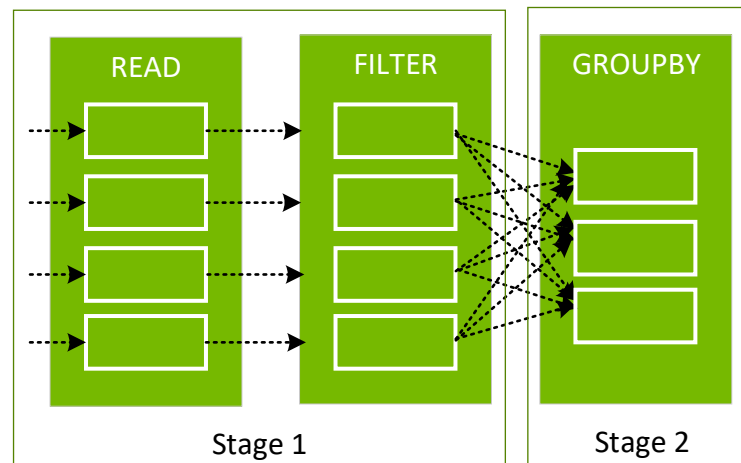


Executing the Tasks on a Cluster

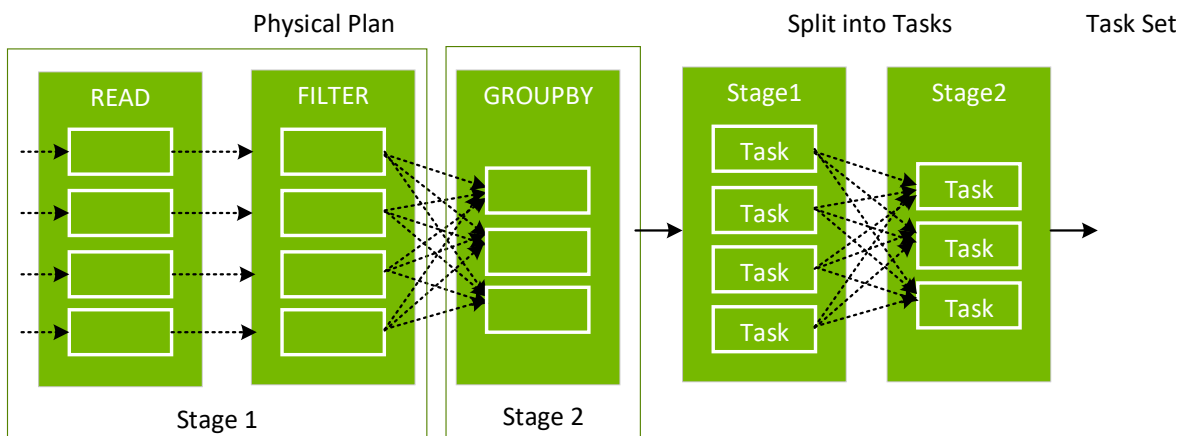
In the third step, the tasks are scheduled and executed on the cluster.



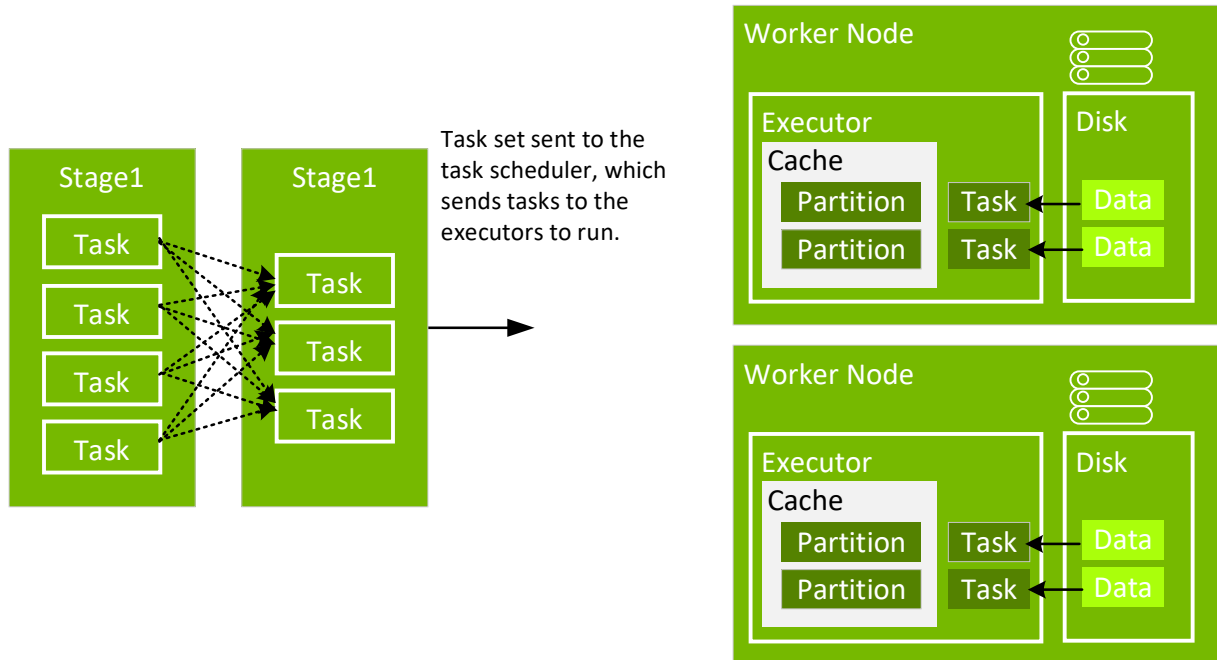
The scheduler splits the graph into stages, based on the transformations. The narrow transformations (transformations without data movement) will be grouped (pipelined) together into a single stage. The physical plan for this example has two stages, with everything before the exchange in the first stage. Spark performs further optimizations at runtime, including **Whole-Stage Java Code Generation**. This optimizes CPU usage by generating a single optimized Java function in bytecode for the set of operators in a SQL query (when possible), instead of generating iterator code for each operator.



Each stage is composed of tasks, based on partitions of the DataFrame, which performs the same computation in parallel.



Next the scheduler submits the stage task set to the task scheduler, which sends tasks to the executors to run.



When the job completes, the action value is returned to the driver, or written to disk, depending on the action.

Clicking the **web UI Jobs** tab gives you details on the progress of the job, including stages and tasks. In the following example, the job consists of two stages, with two tasks in the stage before the shuffle and 200 in the stage after the shuffle. The number of tasks correspond to the partitions. After reading the file in the first stage, there are two partitions.

After a shuffle, the default number of partitions is 200. (You can configure the number of partitions to use when shuffling data with the `spark.sql.shuffle.partitions` property).



Summary

In this chapter, we introduced you to Spark, demonstrated how it executes your code on a cluster, and showed you how to monitor this using the Spark Web UI. Knowing how Spark runs your applications is important when debugging, analyzing, and tuning the performance of your applications.

Chapter 2: Spark SQL and DataFrame Programming

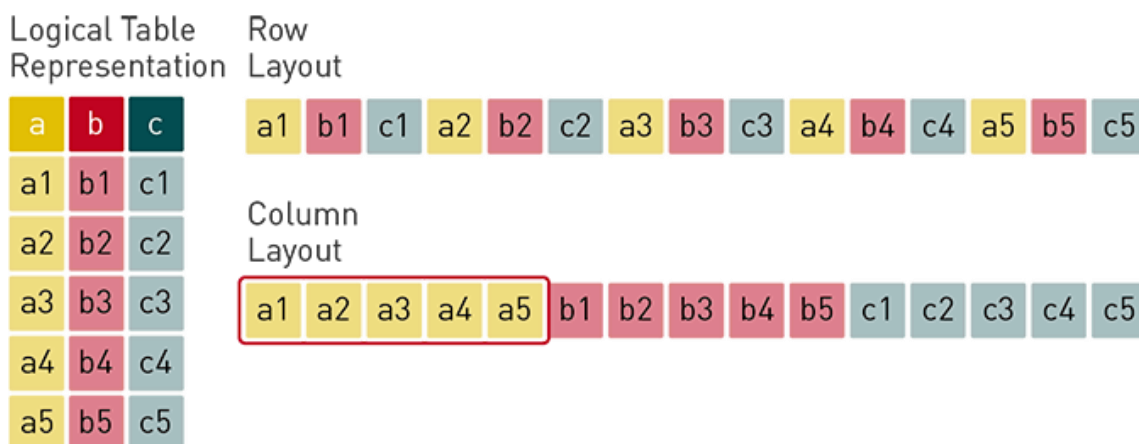
In Chapter 1, we explored how Spark DataFrames execute on a cluster. In this chapter, we'll provide you with an overview of DataFrames and Spark SQL programming, starting with the advantages.

DataFrames and Spark SQL Advantages

The Spark SQL and the DataFrame APIs provide ease of use, space efficiency, and performance gains with Spark SQL's optimized execution engine.

Optimized Memory Usage

Spark SQL caches DataFrames (when you call [DataFrame.cache\(\)](#)) using an in-memory columnar format which is optimized to: scan only required columns, automatically tune compression, minimize memory usage and minimize JVM Garbage Collection.



Spark SQL Vectorized Parquet and ORC readers decompress and decode in column batches, which [is roughly nine times faster for reading](#).

Query Optimization

Spark SQL's Catalyst Optimizer handles logical optimization and physical planning, supporting both rule-based and cost-based optimization. When possible, Spark SQL Whole-Stage Java Code Generation optimizes CPU usage by generating a single optimized function in bytecode for the set of operators in an SQL query.

Exploring the Taxi Dataset with Spark SQL

Data preparation and exploration takes 60 to 80 percent of the analytical pipeline in a typical machine learning (ML) or deep learning (DL) project. In order to build an ML model, you have to clean, extract, explore, and test your dataset in order to find the features of interest that most contribute to the model's accurate predictions. For illustrative purposes, we'll use Spark SQL to explore the Taxi dataset to analyze which features might help predict taxi fare amounts.

Load the Data from a File into a DataFrame and Cache

The following code shows how we loaded the data from a CSV file into a Spark `DataFrame`, specifying the `datasource` and `schema` to load into a `DataFrame`, as discussed in Chapter 1. After we register the `DataFrame` as an SQL temporary view, we can use SQL functions on the `SparkSession` to run SQL queries, which will return the results as a `DataFrame`. We cache the `DataFrame` so that Spark does not have to reload it for each query. Also, Spark can cache `DataFrames` or `Tables` in columnar format in memory, which can improve memory usage and performance.

Load Data → DataFrame

```
// load the data as in Chapter 1
val file = "/data/taxi_small.csv"

val df = spark.read.option("inferSchema", "false")
    .option("header", true).schema(schema).csv(file)

// cache DataFrame in columnar format in memory
df.cache

// create Table view of DataFrame for Spark SQL
df.createOrReplaceTempView("taxi")

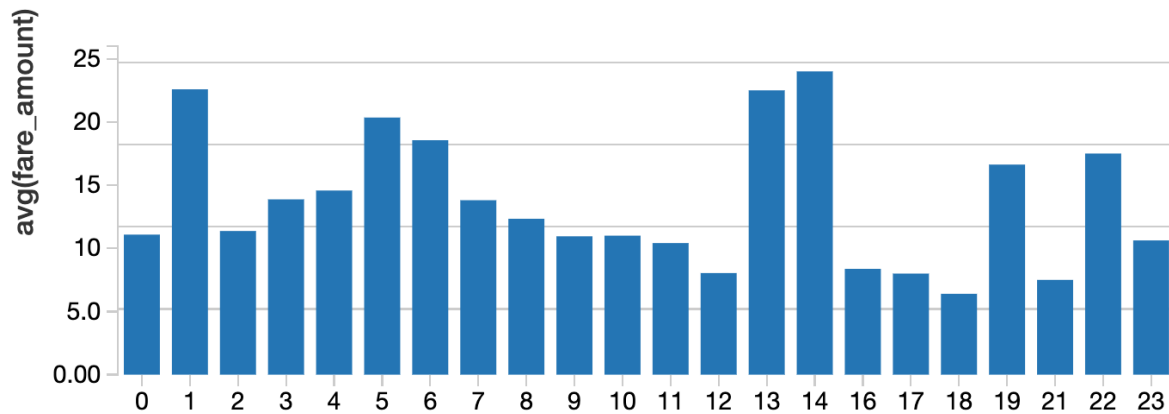
// cache taxi table in columnar format in memory
spark.catalog.cacheTable("taxi")
```


Using Spark SQL

Now we can use Spark SQL to explore what might affect the taxi fare amount, with questions like: What is the average fare amount by hour of the day?

```
%sql
select hour, avg(fare_amount)
from taxi
group by hour order by hour
```

With a notebook like Zeppelin or Jupyter, we can display the SQL results in graph formats.



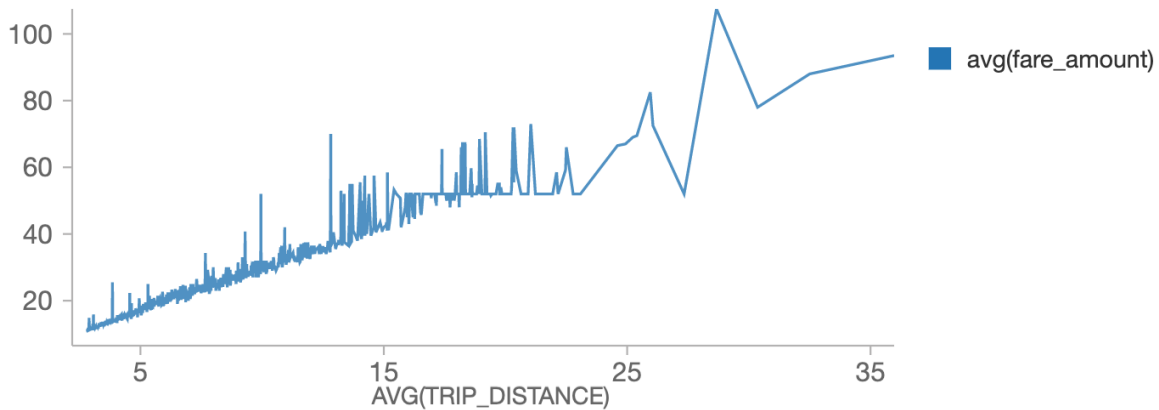
Following is the same query with the DataFrame API:

```
df.groupBy("hour").avg("fare_amount")
.orderBy("hour").show(5)
```

```
result:
+----+-----+
|hour| avg(fare_amount) |
+----+-----+
| 0.0|11.083333333333334|
| 1.0|22.581632653061224|
| 2.0|11.370820668693009|
| 3.0|13.873989218328841|
| 4.0| 14.57204433497537|
+----+-----+
```

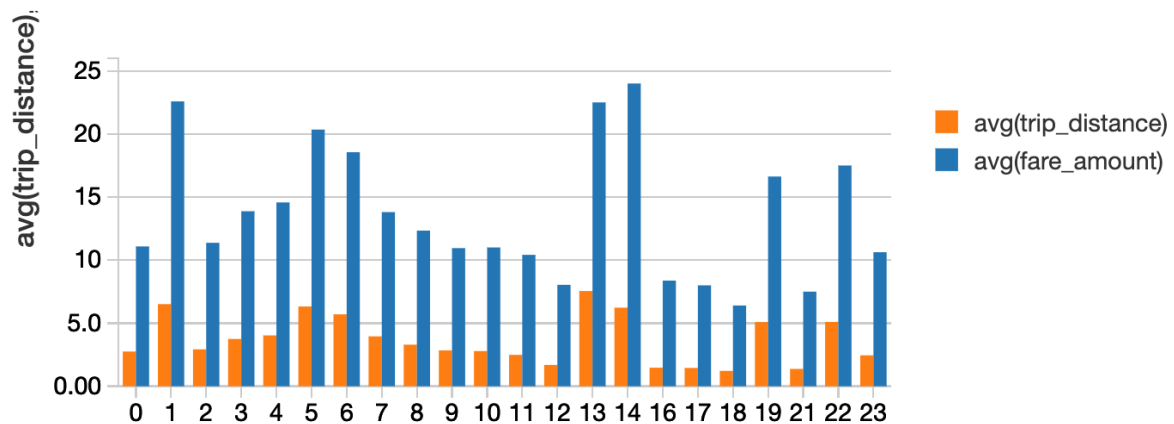
What is the average fare amount compared to the average trip distance?

```
%sql
select trip_distance, avg(trip_distance), avg(fare_amount)
from taxi
group by trip_distance order by avg(trip_distance) desc
```



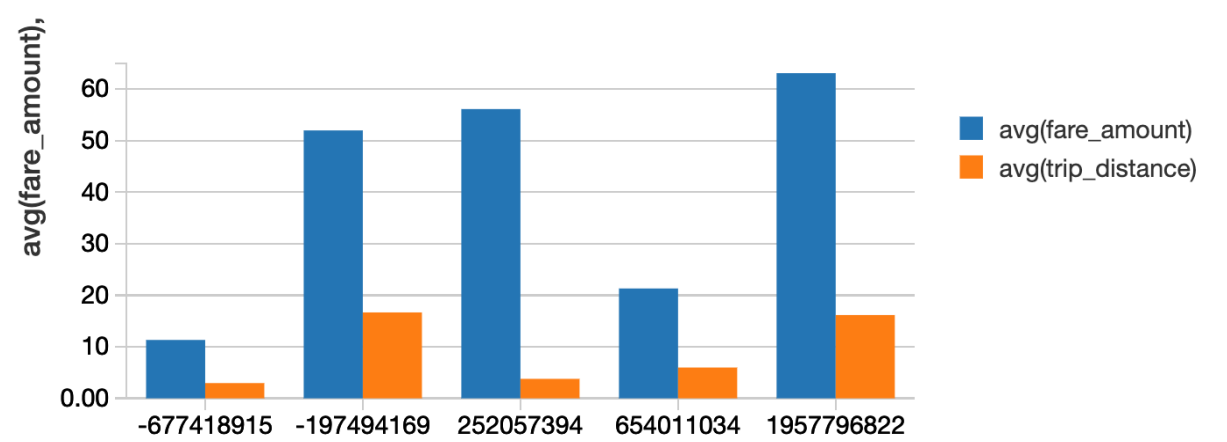
What is the average fare amount and average trip distance by hour of the day?

```
%sql
select hour, avg(fare_amount), avg(trip_distance)
from taxi
group by hour order by hour
```



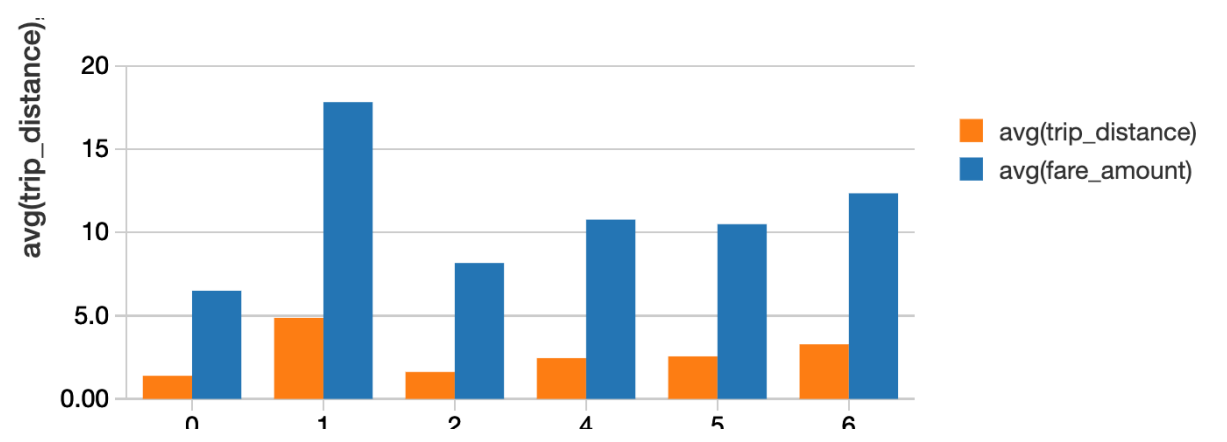
What is the average fare amount and average trip distance by rate code?

```
%sql
select hour, avg(fare_amount), avg(trip_distance)
from taxi
group by rate_code order by rate_code
```



What is the average fare amount and average trip distance by day of the week?

```
%sql
select day_of_week, avg(fare_amount), avg(trip_distance)
from taxi
group by day_of_week order by day_of_week
```



Using Spark Web UI to Monitor Spark SQL

SQL Tab

You can use the Spark SQL tab to view Query execution information, such as the query plan details and SQL metrics. Clicking on the query link displays the DAG of the job.

Jobs

Stages

Storage

Environment

Executors

SQL

JDBC/ODBC Server

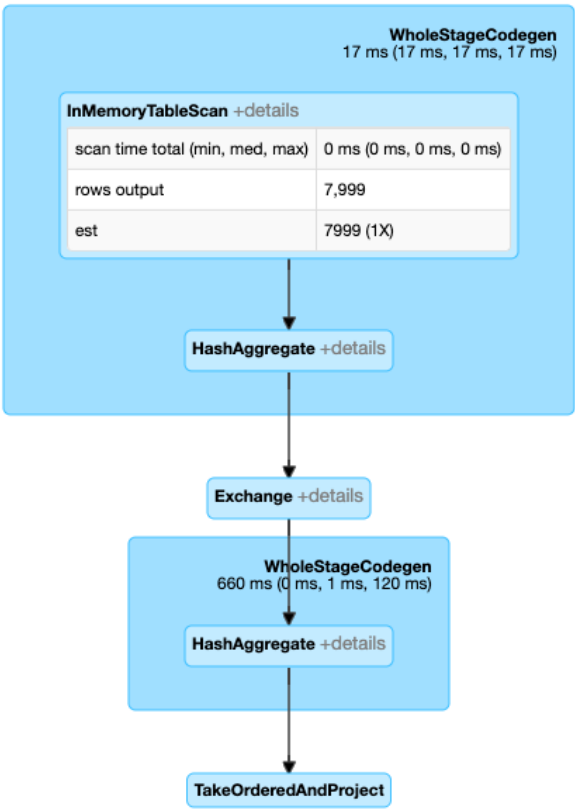
SQL

Completed Queries: 29

Completed Queries (29)

ID	Description	Submitted	Duration	Job IDs
28	<code>tdf.groupBy("hour").avg("fare_amount").orderBy(...)</code> +details	2020/03/27 20:18:15	3 s	[9]

Clicking on the +details in the DAG displays details for that stage



Clicking the **Details** link on the bottom displays the logical plans and the physical plan in text format.

In the query plan details, you can see:

- ▶ The amount of time for each stage.
- ▶ If partition filters, projection, and filter pushdown are occurring.
- ▶ Shuffles between stages (Exchange) and the amount of data shuffled. If joins or aggregations are shuffling a lot of data, consider bucketing.
- ▶ You can set the number of partitions to use when shuffling with the `spark.sql.shuffle.partitions` option.
- ▶ The join algorithm being used. **Broadcast join** should be used when one table is small and **sort-merge join** should be used for large tables. You can use **broadcast** hint to guide Spark to broadcast a table in a join. For faster joins with large tables using the **sort-merge join** algorithm, you can use bucketing to pre-sort and group tables. This will avoid shuffling in the sort merge.

Use the Spark SQL `ANALYZE TABLE tablename COMPUTE STATISTICS` to take advantage of cost-based optimization in the Catalyst Planner.

Jobs Tab

The Jobs tab summary page shows high-level job information, such as the status, duration, and progress of all jobs and the overall event timeline. Here are some metrics to check:

- ▶ **Duration:** Check the amount of time for the job.
- ▶ **Stages succeeded/total tasks, succeeded/total:** Check if there was stage/task failure.

Stages Tab

The stage tab displays summary metrics for all tasks. You can use the metrics to identify problems with an executor or task distribution. Here are some things to look for:

- ▶ **Duration:** Are there tasks that are taking longer? If your task process time is not balanced, resources could be wasted.
- ▶ **Status:** Are there failed tasks?
- ▶ **Read Size, Write Size:** is there skew in data size?
- ▶ If your partitions/tasks are not balanced, then consider repartitioning.

Storage Tab

The Storage tab displays DataFrames that are cached or persisted to disk with size in memory and size on disk information. You can use the storage tab to see if cached DataFrames are fitting into memory. If a DataFrame will be reused, and if it fits into memory, caching it will make execution faster.

Executors Tab

The Executors tab displays summary memory, disk, and task usage information by the executors that were created for the application. You can use this tab to confirm that your application has the amount of resources needed, using the following:

- ▶ **Shuffle Read Write Columns:** Shows size of data transferred between stages.
- ▶ **Storage Memory Column:** Features the current used/available memory.
- ▶ **Task Time Column:** Displays task time/garbage collection time.

Partitioning and Bucketing

File partitioning and Bucketing are common optimization techniques in Spark SQL. They can be helpful for reducing data skew and data shuffling by pre-aggregating data in files or directories. DataFrames can be sorted, partitioned, and/or bucketed when saved as persistent tables. Partitioning optimizes reads by storing files in a hierarchy of directories based on the given columns. For example, when we partition a DataFrame by year:

```
df.write.format("parquet")
  .partitionBy("year")
  .option("path", "/data ")
  .saveAsTable("taxi")
```

The directory would have the following structure:

```
path
├── to
│   └── table
│       ├── year = 2019
│       │   ├── part01.parquet
│       │   └── part02.parquet
│       ├── year = 2018
│       │   └── part01.parquet
│       └── ⋮
│       └── ⋮
```

After partitioning the data, when queries are made with filter operators on the partition column, the Spark SQL catalyst optimizer pushes down the partition filter to the datasource. The scan reads only the directories that match the partition filters, reducing disk I/O and data loaded into memory. For example, the following query reads only the files in the year = '2019' directory.

```
df.filter("year = '2019'")
  .groupBy("year").avg("fareamount")
```


When visualizing the physical plan for this query, you will see `Scan PrunedInMemoryFileIndex[/data/year=2019], PartitionFilters: [(year = 2019)]`.

Similar to partitioning, bucketing splits data by a value. However, bucketing distributes data across a fixed number of buckets by a hash on the bucket value, whereas partitioning creates a directory for each partition column value. Tables can be bucketed on more than one value and bucketing can be used with or without partitioning. If we add bucketing to the previous example, the directory structure is the same as before, with data files in the year directories grouped across four buckets by hour.

```
df.write.format("parquet")
  .partitionBy("year")
  .bucketBy(4, "hour")
  .option("path", "/data ")
  .saveAsTable("taxi")
```

After bucketing the data, aggregations and joins (wide transformations) on the bucketed value do not have to shuffle data between partitions, reducing network and disk I/O. Also, bucket filter pruning will be pushed to the datasource reducing disk I/O and data loaded into memory. The following query pushes down the partition filter on year to the datasource and avoids the shuffle to aggregate on hour.

```
df.filter("year = '2019'")
  .groupBy("hour")
  .avg("hour")
```

Partitioning should only be used with columns used frequently in queries for filtering and that have a limited number of column values with enough corresponding data to distribute the files in the directories. Small files are less efficient with excessive parallelism and too few large files can hurt parallelism. Bucketing works well when the number of unique bucketing column values is large and the bucketing column is used often in queries.

Summary

In this chapter, we explored how to use tabular data with Spark SQL. These code examples can be reused as the foundation for processing data with Spark SQL. In another chapter, we use the same data with DataFrames for predicting taxi fares.

Chapter 3: GPU-Accelerated Apache Spark 3.x

Spark 3.x and GPUs

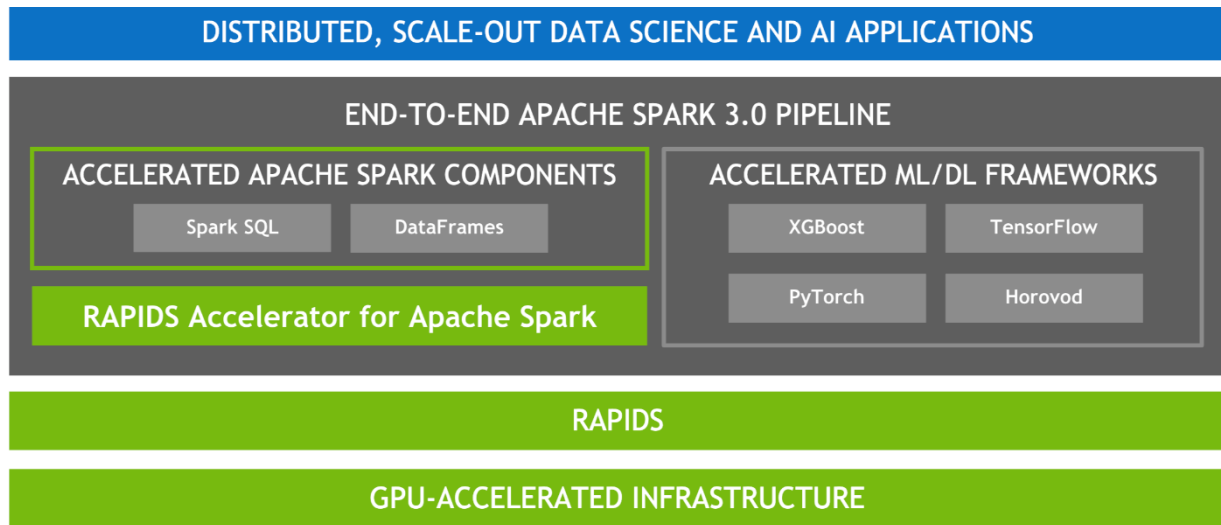
Given the parallel nature of many data processing tasks, it's only natural that the massively parallel architecture of a GPU should be able to parallelize and accelerate Spark data processing queries, in the same way that a GPU accelerates deep learning (DL) in artificial intelligence (AI). Therefore, NVIDIA® has worked with the Spark community to implement GPU acceleration as part of Spark 3.x.

While Spark distributes computation across nodes in the form of partitions, within a partition, computation has historically been performed on CPU cores. However, the benefits of GPU acceleration in Spark are many. For one, fewer servers are required, reducing infrastructure cost. And, because queries are completed faster, you expect a reduction in time to results. Also, since GPU acceleration is transparent, applications built to run on Spark require no changes in order to reap the benefits of GPU acceleration.

Accelerated ETL and AI in Spark

As machine learning (ML) and DL are increasingly applied to larger datasets, Spark has become a commonly used vehicle for the data pre-processing and feature engineering needed to prepare raw input data for the learning phase. The Spark community has been focused on bringing both phases of this end-to-end pipeline together, so that data scientists can work with a single Spark cluster and avoid the penalty of moving data between phases via an external data lake. [Horovod](#) (by Uber) and [TensorFlowOnSpark](#) (by Yahoo) are examples of this approach.

Spark 3.x represents a key milestone, as Spark can now schedule GPU-accelerated ML and DL applications on Spark clusters with GPUs. The complete Spark 3 software stack that includes the RAPIDS Accelerator for Apache Spark is shown in the following figure.

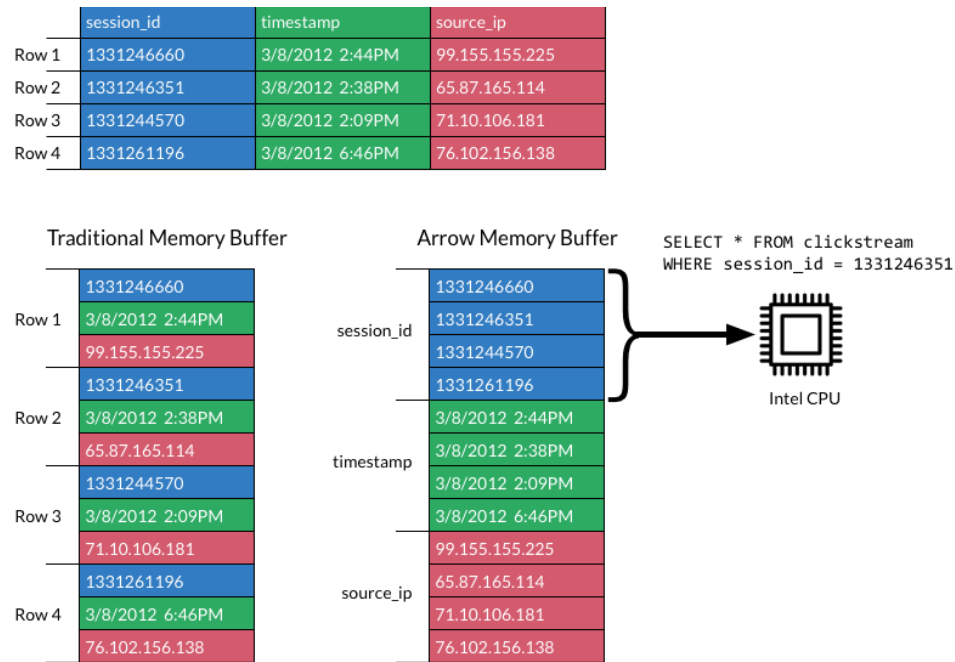


New GPU-Accelerated Libraries on NVIDIA CUDA

As discussed previously, [NVIDIA® CUDA®](#) is a programming model and a set of APIs for accelerating operations on the NVIDIA GPU architecture. Layered on top of CUDA, RAPIDS is a suite of open-source software libraries and APIs that provide GPU parallelism and high-bandwidth memory speed through DataFrame and graph operations.

RAPIDS GPU-Accelerated Spark DataFrames

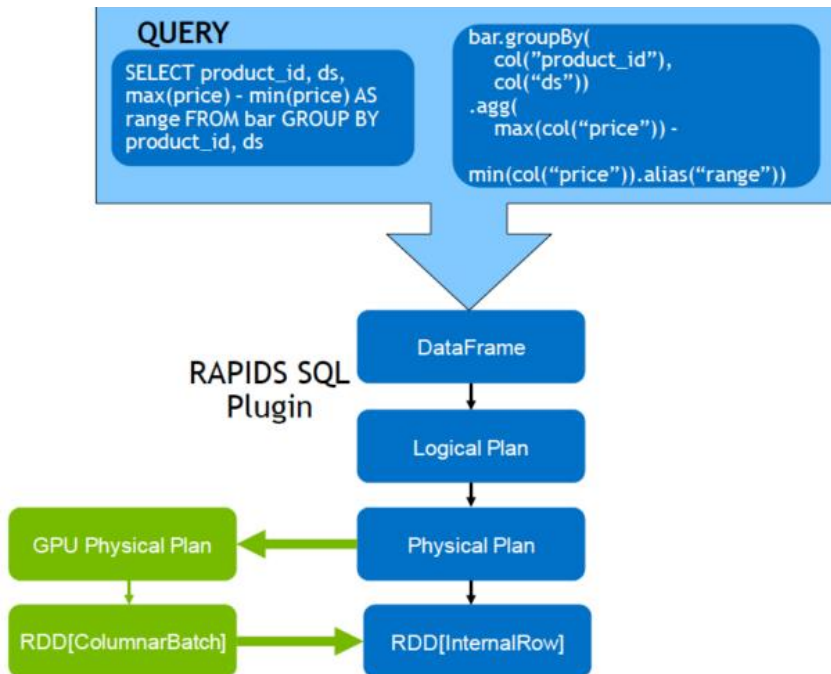
RAPIDS offers a powerful GPU DataFrame based on [Apache Arrow](#) data structures. Arrow specifies a standardized, language-independent, columnar memory format, optimized for data locality, to accelerate analytical processing performance on modern CPUs or GPUs. With the GPU DataFrame, batches of column values from multiple records take advantage of modern GPU designs and accelerate reading, queries, and writing.



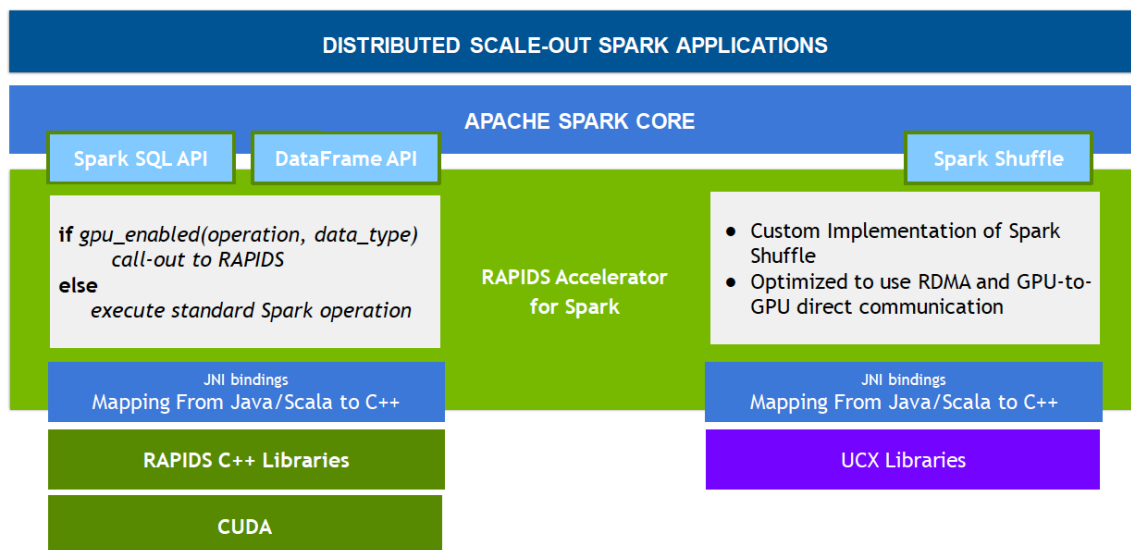
Spark GPU-Accelerated DataFrame and SQL

For Apache Spark 3.0, new RAPIDS APIs are used by Spark SQL and DataFrames for GPU-accelerated memory-efficient columnar data processing and query plans. With the RAPIDS accelerator, the Catalyst query optimizer plugin interface has been extended to identify operators within a query plan that can be accelerated with the RAPIDS API, mostly a one-to-one mapping, and to schedule those operators on GPUs within the Spark cluster when executing the query plan.

With a physical plan for CPUs, the DataFrame data is transformed into RDD row format and usually processed one row at a time. Spark supports columnar batch, but in Spark 2.x only the Vectorized Parquet and ORC readers use it. The RAPIDS plugin extends columnar batch processing on GPUs to most Spark operations. Processing columnar data is much more GPU friendly than row-by-row processing.



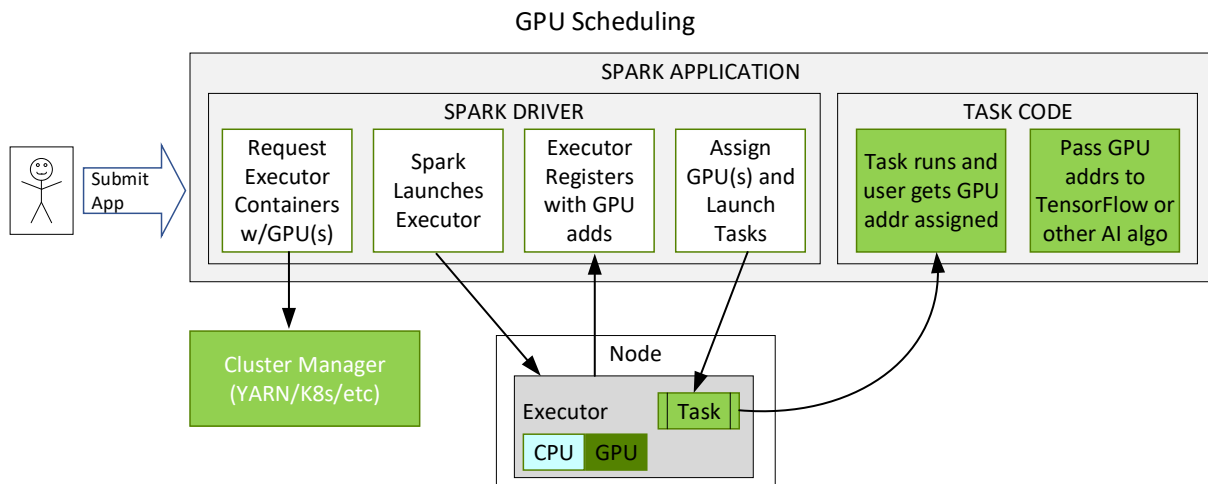
A new Spark shuffle implementation built upon OpenUCX communication libraries leverage NVLink, RDMA and InfiniBand (if available) to dramatically reduce data transfer among Spark processes by: keeping as much data on the GPU as possible, finding the fastest path to move data between nodes, using the best of available hardware resources, including bypassing the CPU to do GPU to GPU memory intra and inter node transfers. RDMA allows GPUs to transfer data directly across nodes at up to PCIe speeds, operating as if on one massive server. NVLink allows GPUs to initiate peer to peer communication at up to 300GB/s.



GPU-Aware Scheduling in Spark

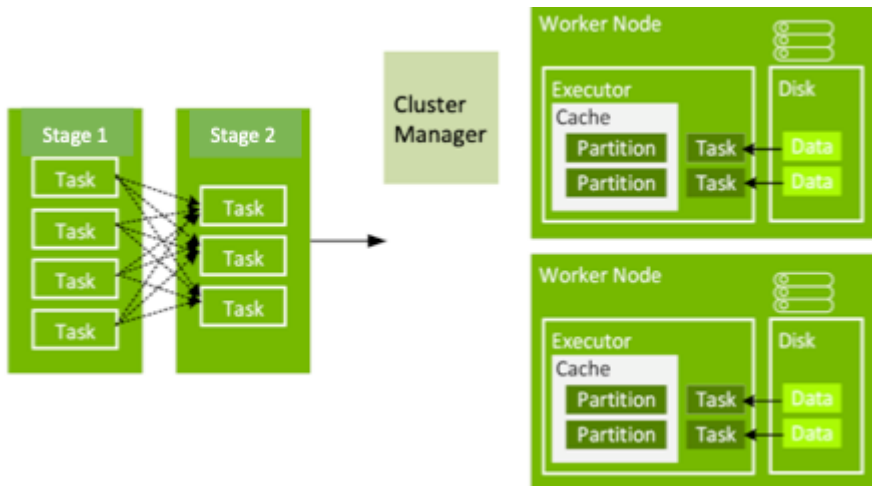
GPUs are now a schedulable resource in Apache Spark 3.0. This allows Spark to schedule executors with a specified number of GPUs, and you can specify how many GPUs each task requires. Spark conveys these resource requests to the underlying cluster manager, Kubernetes, YARN, or standalone. You can also configure a discovery script to detect which GPUs were assigned by the cluster manager. This makes GPUs easier to request and use for Spark application developers, allows for closer integration with DL and AI frameworks like Horovod and TensorFlow on Spark, and allows for better utilization of GPUs. This greatly simplifies running ML or DL applications that need GPUs, as you previously had to work around the lack of GPU scheduling in Spark applications.

An example of a flow for GPU scheduling is shown in the diagram below. The user submits an application with a GPU resource configuration discovery script. Spark starts the driver, which uses the configuration to pass on to the cluster manager, to request a container with a specified amount of resources and GPUs. The cluster manager returns the container. Spark launches the container. When the executor starts, it will run the discovery script. Spark sends that information back to the driver and the driver can then use that information to schedule tasks to GPUs.

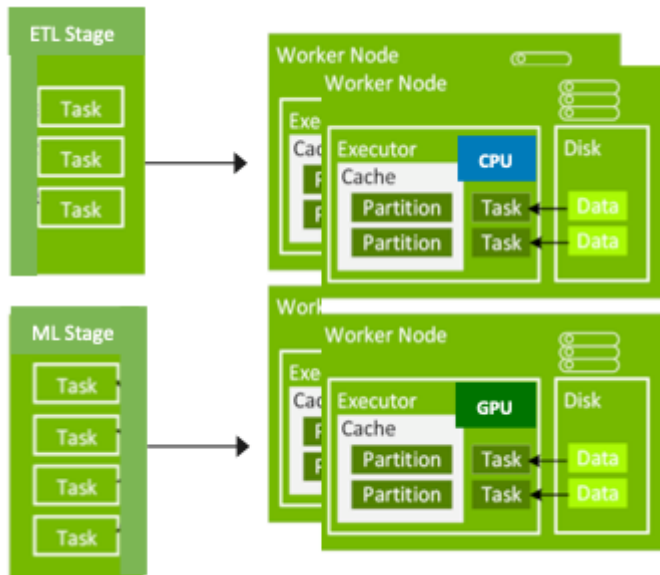


Stage Level Scheduling in Spark

Recall from the overview of *How a Spark Application Executes*, page 20, the scheduler splits the physical plan DAG into stages, based on what operations can be performed serially or in parallel, then the scheduler submits the stage task set to the task scheduler, which sends tasks to the executors to run.



Spark 3.1 and newer stage level resource scheduling allows you to use different executor resources for different stages of your job, adapted to the workload type. For example, CPUs and memory-intensive executors for first stage data processing, and GPUs in a second phase for ML.



Stage level scheduling extends upon accelerator aware scheduling by improving big data ETL and DL integration and also enables multiple other use cases. It is beneficial any time the user wants to change container resources between stages in a single Apache Spark application, whether those resources are CPU, Memory or GPUs. One of the most popular use cases is enabling end-to-end scalable Deep Learning to efficiently use GPU resources. In this type of use case, users read from a distributed file system, do data transformations to get the data into a format that the Deep Learning algorithm needs for training or inference and then send the data into a Deep Learning algorithm. Using stage level scheduling combined with accelerator aware scheduling enables users to seamlessly go from ETL to Deep Learning

running on the GPU by adjusting the container resource requirements for different stages in Spark within the same application.

This makes writing these applications easier and can help with hardware utilization and costs. There are other ETL use cases where users want to change CPU and memory resources between stages, for instance when there is data skew or when the data size is much larger in certain stages of the application.

Using the Spark Web UI to Monitor GPU Resources

Recall from the overview *Using Spark Web UI to Monitor Spark SQL*, page 36, the Executors tab displays summary information about the executors that were created for the application. You can use the Resources checkbox in this tab to see which resources have been allocated. In this example, two GPUs have been allocated.

The screenshot shows the Spark Web UI Executors tab. The 'Resources' checkbox is selected, indicated by a green arrow. The summary table shows the following data:

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(2)	0	0.0 B / 8.7 GiB	0.0 B	2	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	0
Total(2)	0	0.0 B / 8.7 GiB	0.0 B	2	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	0

The Executors table shows the following data:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Resources	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
driver	10.28.9.112:42305	Active	0	0.0 B / 8.4 GiB	0.0 B	0		0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	Logs	Thread Dump
1	tomg-x299-37047	Active	0	0.0 B / 366.3 MiB	0.0 B	2	gpu: [0, 1]	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump

Showing 1 to 2 of 2 entries

For executors configured with stage level scheduling, the Resource Profile Id checkbox displays the profile id number of each executor.

The screenshot shows the Spark Web UI Executors tab. The 'Resource Profile Id' checkbox is selected, indicated by a green arrow. The summary table shows the following data:

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Comple
Active(6)	0	21.5 KiB / 43.7 GiB	0.0 B	18	0	0	4
Dead(1)	0	0.0 B / 21.2 GiB	0.0 B	2	0	0	0
Total(7)	0	21.5 KiB / 64.9 GiB	0.0 B	20	0	0	4

The Executors table shows the following data:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Resources	Resource Profile Id
driver	[REDACTED]	Active	0	7.2 KiB / 10.5 GiB	0.0 B	0		0
1	[REDACTED]	Dead	0	0.0 B / 21.2 GiB	0.0 B	2	gpu: [0]	0

By selecting the **Environment** tab, you can see the executor and task requirements associated with each resource profile id.

Resource Profile Id	Resource Profile Contents
0	Executor Reqs: cores: [amount: 2] memory: [amount: 40960] offHeap: [amount: 0] gpu: [amount: 1, discovery: ./getGpus] Task Reqs: cpus: [amount: 1.0] gpu: [amount: 0.5]
1	Executor Reqs: memoryOverhead: [amount: 2048] cores: [amount: 4] memory: [amount: 6144] gpu: [amount: 2, discovery: ./getGpus] Task Reqs: cpus: [amount: 4.0] gpu: [amount: 2.0]

XGBoost, RAPIDS, and Spark

XGBoost is a scalable, distributed gradient-boosted decision tree (GBDT) ML library. XGBoost provides parallel tree boosting and is the leading ML library for regression, classification, and ranking problems. The RAPIDS team works closely with the Distributed Machine Learning Common (DMLC) XGBoost organization, and XGBoost now includes seamless, drop-in GPU acceleration, significantly speeding up model training and improving accuracy for better predictions.

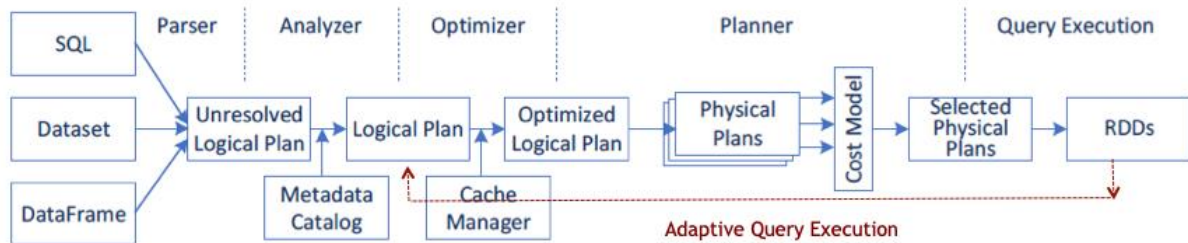
RAPIDS, XGBOOST, and SPARK has three features that help with speed-up and cost:

- ▶ **GPU-accelerated DataFrame:** Reads any number/size of supported input file formats directly into GPU memory and divides up evenly among the different training nodes.
- ▶ **GPU-accelerated training:** XGBoost training time has been improved with a dynamic in-memory representation of the training data that optimally stores features based on the sparsity of a dataset. This replaces a fixed in-memory representation based on the largest number of features amongst different training instances.
- ▶ **Efficient GPU memory utilization:** XGBoost requires that data fit into memory which creates a restriction on data size using either a single GPU or distributed multi-GPU multi-node training. The latest release has improved GPU memory utilization by 5X. Now users can train with data that is five times the size as compared to the first version. This improves total cost of training without impacting performance.

Later in this eBook, we explore and discuss an example using the upgraded XGBoost library to load/transform data and conduct distributed training using GPUs.

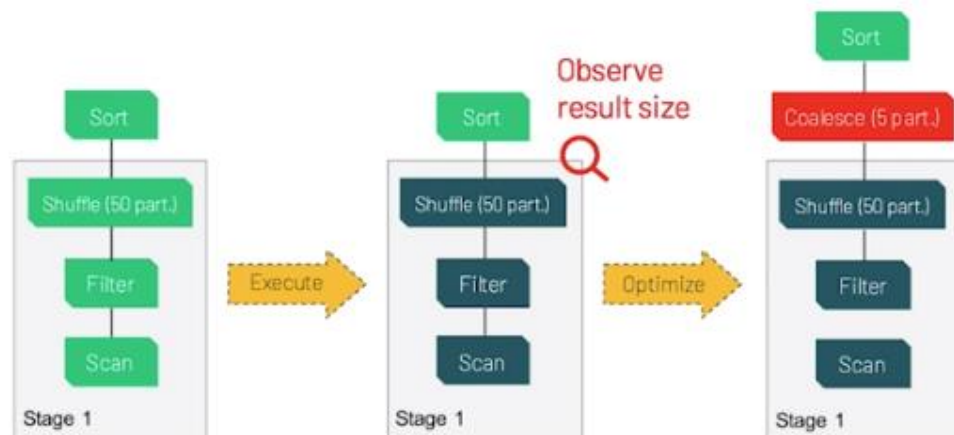
Other Spark 3.x Features

- **Adaptive Query execution:** Spark 2.2 added cost-based optimization to the existing rule based SQL Optimizer. Spark 3.0 now has runtime adaptive query execution(AQE). With AQE, runtime statistics retrieved from completed stages of the query plan are used to re-optimize the execution plan of the remaining query stages. Databricks benchmarks yielded speed-ups ranging from 1.1x to 8x when using AQE.

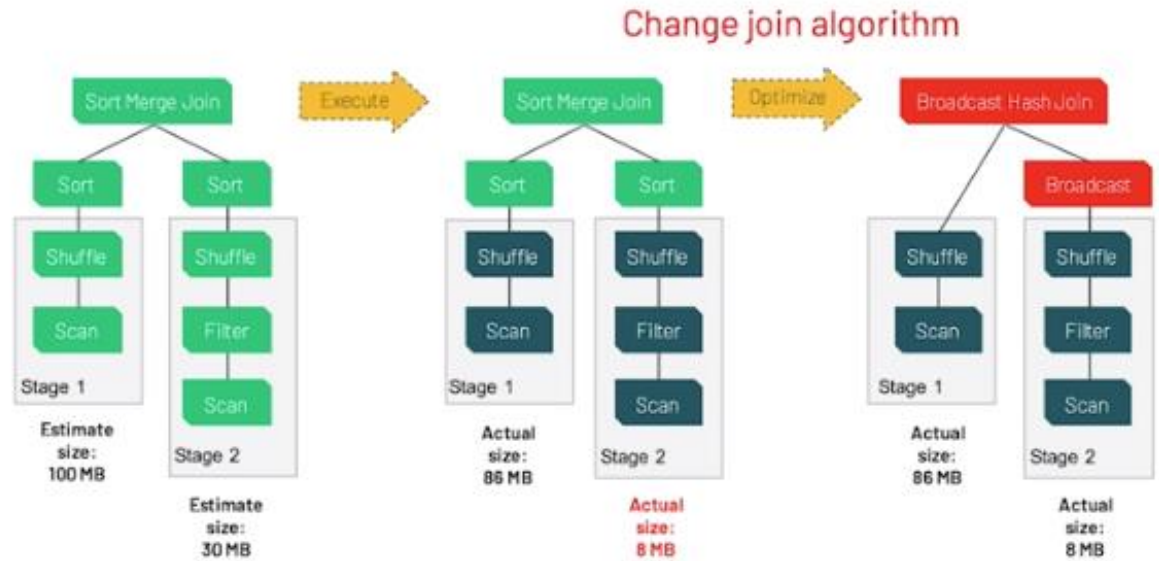


Spark 3.0 AQE optimization features include:

- **Dynamically coalesce shuffle partitions:** AQE can combine adjacent small partitions into bigger partitions in the shuffle stage by looking at the shuffle file statistics, reducing the number of tasks for query aggregations.

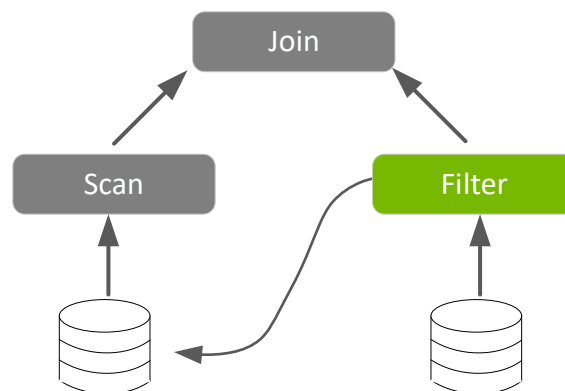


- **Dynamically switch join strategies:** AQE can optimize the join strategy at runtime based on the join relation size. For example, converting a sort merge join to a broadcast hash join which performs better if one side of the join is small enough to fit in memory.



- Dynamically optimize skew joins: AQE can detect data skew in sort-merge join partition sizes using runtime statistics and split skew partitions into smaller sub-partitions.
- **Dynamic Partition Pruning:** Partition pruning is a performance optimization that limits the number of files and partitions that Spark reads when querying. After partitioning the data, queries that match certain partition filter criteria improve performance by allowing Spark to only read a subset of the directories and files. Spark 3.0 dynamic partition pruning allows the Spark engine to dynamically infer, at runtime, the specific partitions within a table that need to be read and processed for a specific query, by identifying the partition column values that result from filtering another table in a join. For example, the following query involves two tables: the **flight_sales** table that contains all of the total sales for all flights, partitioned by originating airport, and the **flight_airports** table that contains a mapping of airports for each region. Here we are querying for sales in the North-East America region.

```
select fs.airport, fs.total_sales
from flight_sales fs, flight_airports fa
where fs.airport = fa.airport and fa.region = 'NEUSA'
```

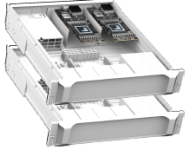
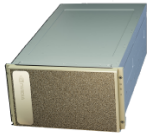




With dynamic partition pruning, this query will scan and process only the partitions for the airports returned by the where filter on the region. Reducing the amount of data read and processed results in a significant time savings.

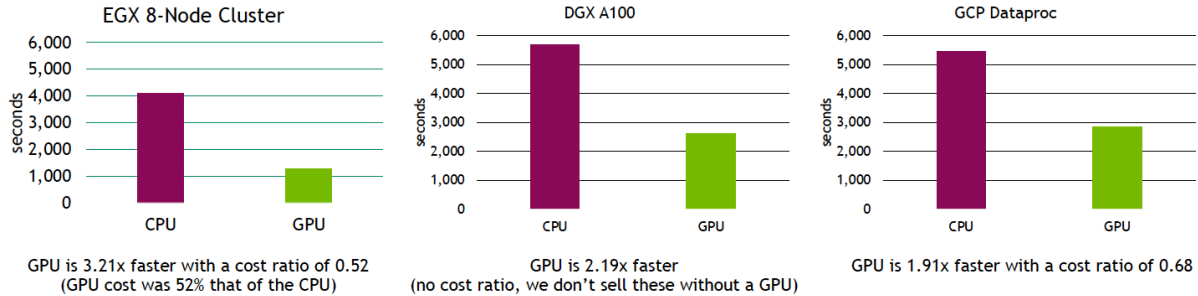
- ▶ Join strategy hints instruct the optimizer to use the hinted plan for join strategies. MERGE, SHUFFLE_HASH and SHUFFLE_REPLICATE_NL hints were added to the existing BROADCAST hint.
- ▶ DataSource API Improvements:
 - Pluggable catalog integration.
 - Improved predicate push down for faster queries via reduced data loading.

Spark 3.x CPU vs. GPU Performance Comparisons

For Spark 3.x CPU vs. GPU performance comparisons, the NVIDIA RAPIDS Accelerator team uses NVIDIA Decision Support (NDS), a NVIDIA adaptation of an industry-standard data science benchmark often used by Apache Spark customers and providers. NDS consists of the same 100+ SQL queries as the industry standard benchmark, designed to mimic large-scale ETL from a retail or company, but parts have been modified for dataset generation and execution. This performance comparison was performed on cluster configurations shown in the table below, with approximately 3 TB of raw data compressed and partitioned using Parquet to 1 TB stored in the native file system for the cluster (HDFS, GCFS, or the local file system):

<div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;">  <p>EGX / NVIDIA Certified OEM servers</p> </div> <div style="text-align: center;">  <p>DGX A100</p> </div> <div style="text-align: center;">  <p>Google Cloud</p>  <p>GCP Dataproc</p> </div> </div>			
Nodes	8	1	1 driver (CPU only), 8 workers
CPU	2 x AMD EPYC 7452 (64 cores/128 threads)	2 x AMD Rome 7742 (128 cores/256 threads)	n1-standard-4 (driver) 8 x n1-highmem-16 (workers)
GPU	2 x NVIDIA Ampere A100, PCIe, 250W, 40GB	8 x NVIDIA Ampere A100 40GB	1 x 16GB T4 per executor
RAM	0.5 TB	2 TB	104 GB
Storage	4 x 7.68 TB Gen4 U.2 NVMe	8 x 3.84 TB Gen4 U.2 NVMe	Google Cloud Storage
Networking	1 x Mellanox CX-6 Single Port HDR100 QSFP56	8 x Mellanox CX-6 Single Port HDR 200Gb/s InfiniBand	32 Gbps
Cost w/o GPU	~\$42,000 per w/ bulk discount	N/A	\$9.08/hour incl GCE + Dataproc
Cost w/ GPU	~\$71,000 per w/ bulk discount	Approximately \$239,000 retail	\$11.88/hour incl GCE + Dataproc
Software	HDFS (Hadoop 3.2.1) Spark 3.0.2 (stand alone)	Spark 3.0.2 (stand alone)	Dataproc Spark 3.0.1 + YARN

The following graphs shows the results of this performance comparison. For the EGX 8-Node Cluster, the GPU is 3.21x faster, and the GPU cost is 52% that of the CPU. For the DGX A100 server, the GPU is 2.19x faster, there is no cost ratio, since these are not sold without a GPU. On GCP Dataproc, the GPU is 1.91x faster with a cost ratio of 0.68.



GPUs are not a silver bullet, they are not going to solve all problems for all queries, the following Spark workloads don't accelerate well:

- ▶ Small Data Sizes (spark.sql.adaptive.advisoryPartitionSizeInBytes=1G)
- ▶ Very Small Aggregate Results
- ▶ Complex user-defined functions
- ▶ Row-wise processing (Lack of GPU support)
- ▶ Host/Device Memory Transfers

Whereas the following Spark workloads accelerate really well:

- ▶ Group by (aggregate) operations with high cardinality
- ▶ Joins with a high cardinality
- ▶ Sorts with a high cardinality
- ▶ Window operations, especially for large windows
- ▶ Column-wise operations
- ▶ Complicated processing
- ▶ Writing Parquet/ORC
- ▶ Reading CSV
- ▶ Transcoding (reading an input file and doing minimal processing before writing it out again, possibly in a different format, like CSV to Parquet)

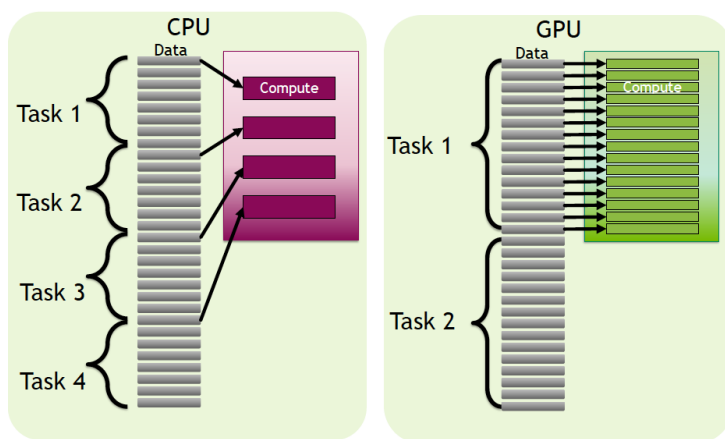


GPUs can be truly amazing in a lot of cases. If you have high cardinality data; that is, if you're doing very big Joins, very big Aggregates, crazy Sorts, the GPU can go at insane speeds compared to what the CPU can do. If the window that you're operating on is bigger than the CPU cache, for say a Sort or Join, you're definitely going to be able to beat it on the GPU.

As for Window operations, if you're doing processing on large windows, the GPU can completely destroy the CPU in many cases. GPUs are able to process those much faster Aggregations with lots of distinct operations than the CPU can.

Complicated processing is another thing that GPUs do really really well. For example, writing out to Parquet or ORC, is not only complex, it's incredibly expensive because the system is doing all kinds of compression. There are risks of deduplication and other statistics involved in getting the size of those files as small as possible. In some cases, there is a 20X speed up by writing to Parquet or ORC on a GPU compared with a CPU, which is going to save both time and money. Parsing CSV can also be very expensive. Some really smart people have spent a lot of time figuring out how to do that in a really parallel way, and the GPU is able to beat the CPU in most cases.

Why is the GPU really fast? The image below shows why the GPU can be amazingly fast in a lot of cases. Spark on the CPU relies on task level parallelism, where all the data is split up between tasks, and each core on the CPU processes only one task until it is done, then it goes to another task not assigned to a core. The GPU does task and data level parallelism, it processes all the data for a task at once completely, not one piece at a time. Data parallelism is cooperative and uses a more consistent amount of memory. For operations that benefit from cooperation such as groupby or sort, being able to process all the data at once in parallel is faster. Because there are fewer tasks, it also reduces the total data in memory, which results in less spilling. Data parallelism also helps with skewed data sets, where you have one task that needs to be larger than other tasks.



Task Level Parallelism vs Data Level Parallelism

For more information see [Running Large-Scale ETL Benchmarks with GPU-Accelerated Apache Spark Video](#).

Summary

In this chapter, we covered the main improvements in Spark 3.x that are proving instrumental in accelerating time to insights, especially when executed on NVIDIA GPUs.

Chapter 4: Getting Started with GPU-Accelerated Apache Spark 3

In the previous chapter, we discussed the features of GPU-Acceleration in Spark 3.x. In this chapter, we go over the basics of getting started using the new RAPIDS Accelerator for Apache Spark 3.x that leverages GPUs to accelerate processing via the [RAPIDS](#) libraries (For details refer to the [Getting Started with the RAPIDS Accelerator for Apache Spark](#)).

The RAPIDS Accelerator for Apache Spark has the following features and limitations:

- ▶ Allows running Spark SQL on a GPU with Columnar processing
- ▶ Requires no API changes from the user
- ▶ Handles transitioning from Row to Columnar and back
- ▶ Runs supported SQL operations on the GPU, If an operation is not implemented or not compatible with GPU, it will fall back to using the Spark CPU version.
- ▶ The accelerator library also provides an implementation of Spark's shuffle that can leverage UCX to optimize GPU data transfers, keeping as much data on the GPU as possible and bypassing the CPU to do GPU to GPU transfers.
- ▶ The RAPIDS Accelerator cannot accelerate operations that manipulate RDDs directly.
- ▶ The RAPIDS Accelerator supports the DataFrame API which is implemented in Spark as Dataset[Row]. When using custom classes or types with Dataset, it is likely most query operations will not be [performed on the GPU](#).

Accelerated Spark Platforms

NVIDIA worked with the Apache Spark Community, Databricks, Google, AWS, Cloudera, and Microsoft to offer [GPU acceleration](#) on all leading Spark platforms, making it easy and cost-effective to launch scalable Apache Spark clusters with NVIDIA GPU acceleration. GPU accelerated Spark offers the following benefits:

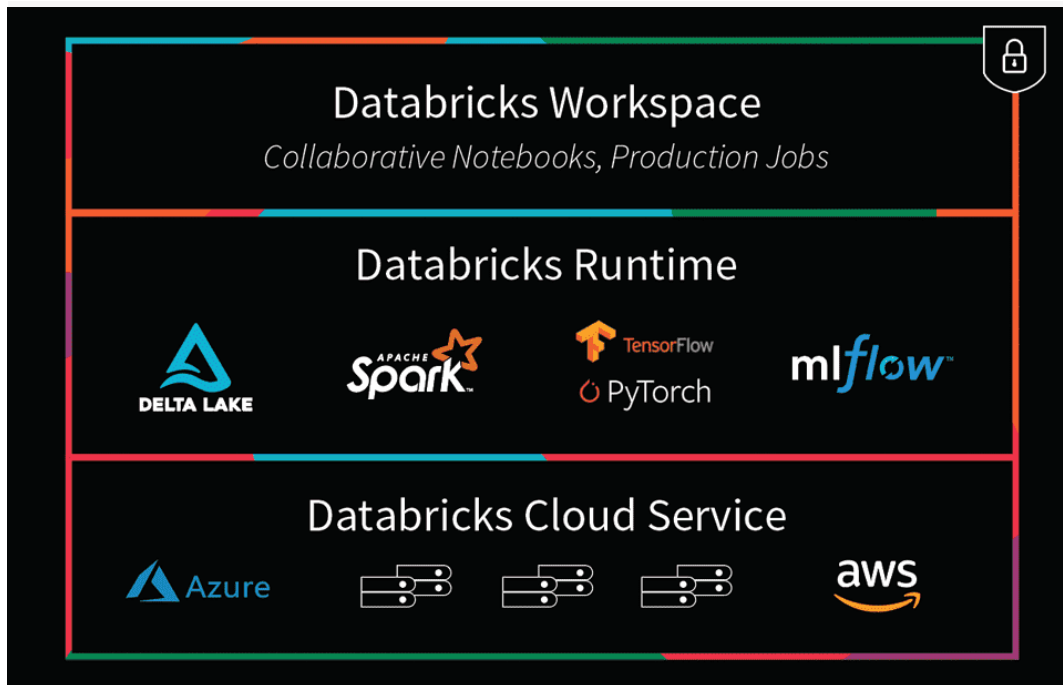
- ▶ Data processing, queries and model training are completed faster, allowing accelerated time to insight.
- ▶ The same GPU-accelerated infrastructure can be used for both Spark and ML/DL frameworks, eliminating the need for complex decision making and tuning.
- ▶ Fewer compute nodes are required; reducing infrastructure cost and potentially helping avoid scale-related problems.

Next, we give an overview of the leading Spark platforms that support NVIDIA GPU acceleration. (Spark can also be run on-premises with the RAPIDS software and RAPIDS Accelerator for Spark being available directly from NVIDIA.)

Databricks

From the original creators of Apache Spark, the Databricks Lakehouse Platform is a fully managed cloud platform for massive scale data engineering and collaborative data science. The Databricks Lakehouse Platform offers:

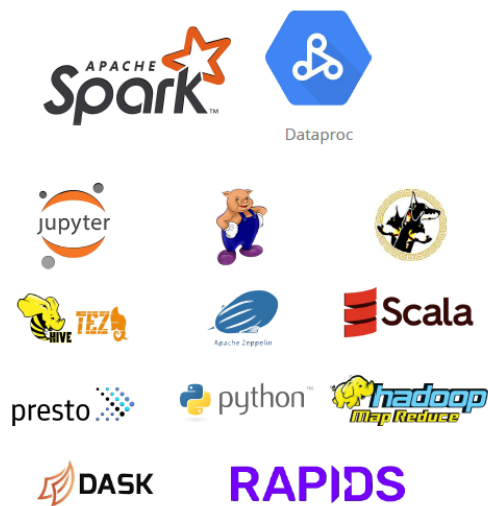
- ▶ Collaborative environment for data teams to build solutions together with integrated workflows, and enterprise security.
- ▶ Interactive notebooks to use Apache Spark™, SQL, Python, Scala, Delta Lake, MLflow, TensorFlow, Keras, Scikit-learn and more.
- ▶ Runs on every major public cloud, tightly integrated with the security, compute, storage, analytics, and AI services natively offered by the cloud providers.
- ▶ Data storage in low cost cloud object stores such as AWS S3, and Azure Data Lake Storage, and Google Cloud Storage.



Google

Google Cloud [Dataproc](#) is a managed cloud service for running Apache Spark and Apache Hadoop clusters in addition to other open source software of the extended Hadoop ecosystem. Dataproc combines the best software in the open-source ecosystem with the advantages of Google Cloud infrastructure making open-source analytics and data processing fast, easy, and more secure in the cloud.

Taking the best of open source



And opening up access to the best of GCP



Features of Dataproc include:

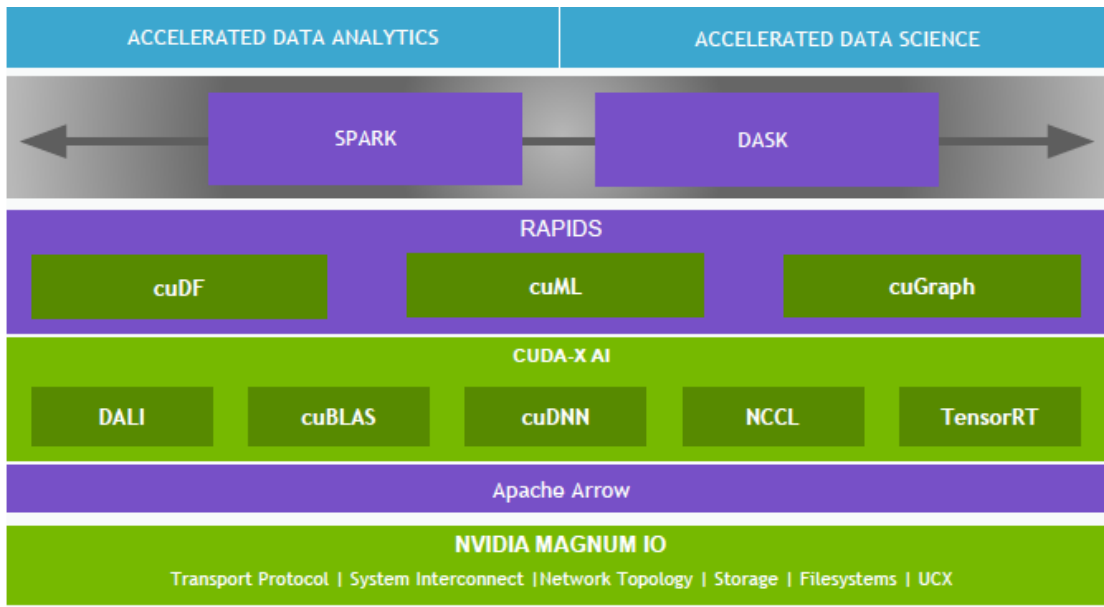
- ▶ Fully managed Apache Spark 3, built on top of Yarn or Kubernetes.
- ▶ Fully customizable compute resources - VMs, GPUs, RAM.
- ▶ Customizable OSS components including GPU drivers and NVIDIA RAPIDS.
- ▶ Spin up a cluster with [auto scaling](#) to dynamically change the computing power of the cluster.
- ▶ Only pay for the resources you use.



- ▶ Encryption and unified security built into every cluster.
- ▶ Integrates seamlessly with BigQuery and other Google Cloud services using open source connectors.
- ▶ OSS support via [optional components](#) and [initialization actions](#).
- ▶ Managed notebook environments using [Dataproc Hub](#).
- ▶ Easy access to web UIs using a [component gateway](#)

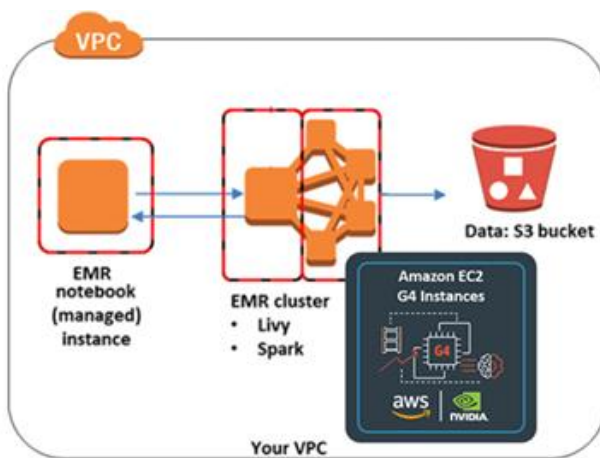
You can use Dataproc for preprocessing your data using Apache Spark and then use that same Spark cluster to power your Notebook for machine learning. The [machine learning initialization](#) action provides a set of commonly used libraries to reduce the time spent configuring your cluster for machine learning including:

- ▶ Python packages such as [TensorFlow](#), [PyTorch](#), [MxNet](#), [Scikit-learn](#) and [Keras](#).
- ▶ R packages including [XGBoost](#), [Caret](#), [randomForest](#), [sparklyr](#).
- ▶ [Dask](#) and [Dask-Yarn](#).



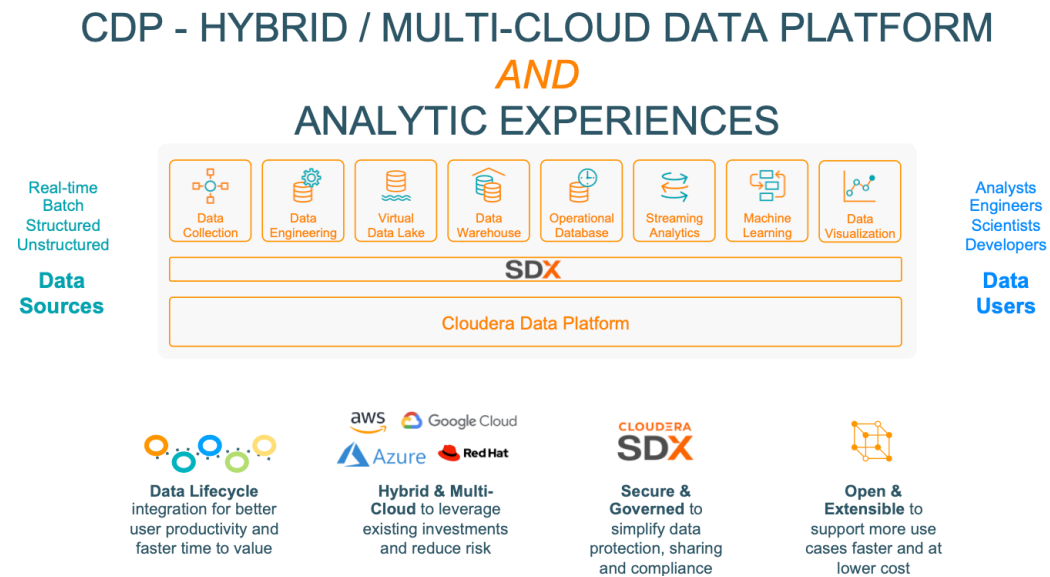
AWS

Amazon EMR is a managed cluster platform that simplifies running big data frameworks, such as [Apache Spark](#), on AWS to process and analyze vast amounts of data. With EMR release version 6.2.0 and later, you can quickly and easily create scalable and secure clusters with Apache Spark 3.x, the RAPIDS Accelerator, and NVIDIA GPU-powered Amazon EC2 instances with a few clicks on the EMR console. With EMR's per-second billing and [Spot Instances](#), you can easily run data science pipelines at a massive scale but low cost. Additionally, you can use AWS data stores, [Amazon SageMaker](#), open-source DL and ML tools such as TensorFlow and Apache MXNet, [Apache Zeppelin](#), or [Jupyter](#) notebooks, and [Apache Livy](#) to enable data scientists to easily and quickly build ETL and ML pipelines and move ML models into production.

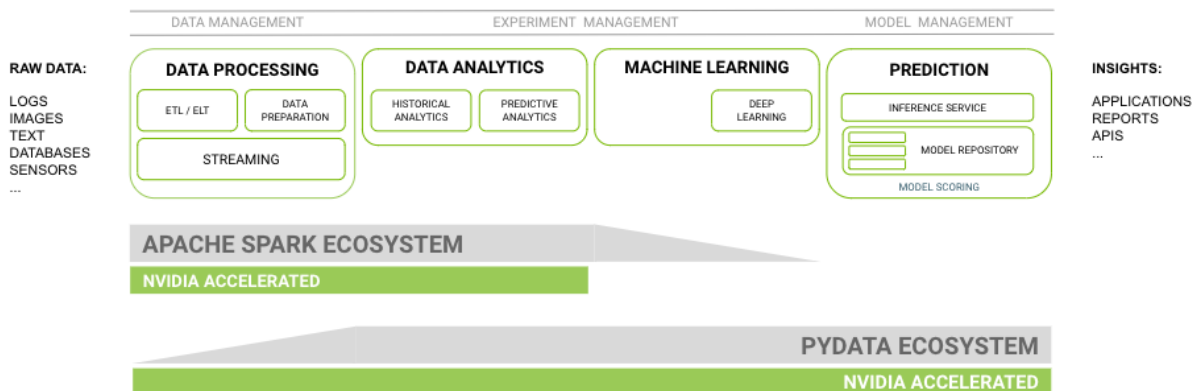


Cloudera

[Cloudera Data Platform](#) (CDP) is a software framework that provides big data management and analytics services for enterprises across hybrid public cloud, private cloud, and multi-cloud environments. CDP can manage data and data workloads, spin or scale the necessary cluster infrastructure and software up and down on-demand, and do that on-premises as well as across the three major public clouds. CDP enables structuring and optimizing data and data processing where they are best suited and allows existing on-prem implementations to “burst to the cloud” for scaling and performance.



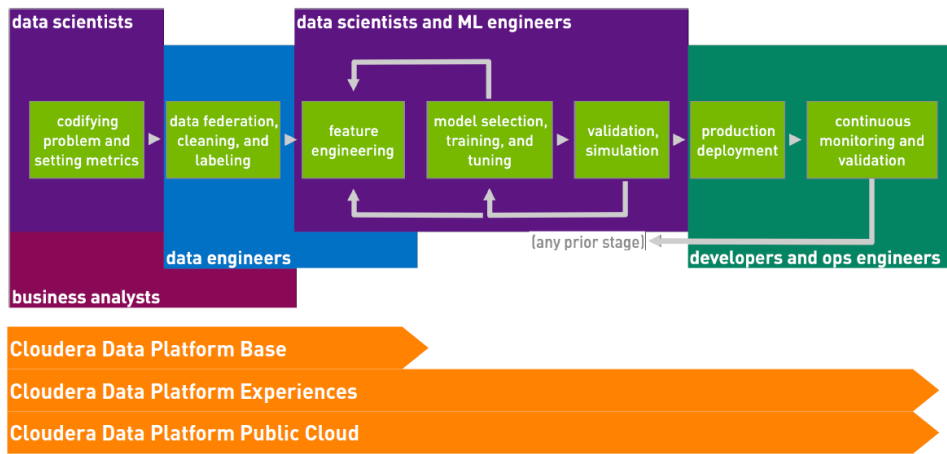
[RAPIDS](#) integration with ML/DL frameworks and Spark 3.x GPU task scheduling enables the acceleration of model training and tuning. This allows data scientists and ML engineers to have a unified, GPU-accelerated pipeline for [ETL and analytics](#), while ML and DL applications leverage the same GPU infrastructure, removing bottlenecks, increasing performance, and simplifying clusters. For IT teams, the simplest infrastructure path to enabling this accelerated data science is to deploy [NVIDIA Certified Servers](#).



CDP Powered by NVIDIA Accelerated Data Science

While effectively leveraging GPUs to accelerate end-to-end ETL and ML workflows with GPUs has been difficult in the past, enabling this capability on CDP powered by NVIDIA is turn-key. Cloudera, together with NVIDIA, makes it easier than ever to optimize data science workflows and execute compute-heavy processes in a fraction of the time previously required.

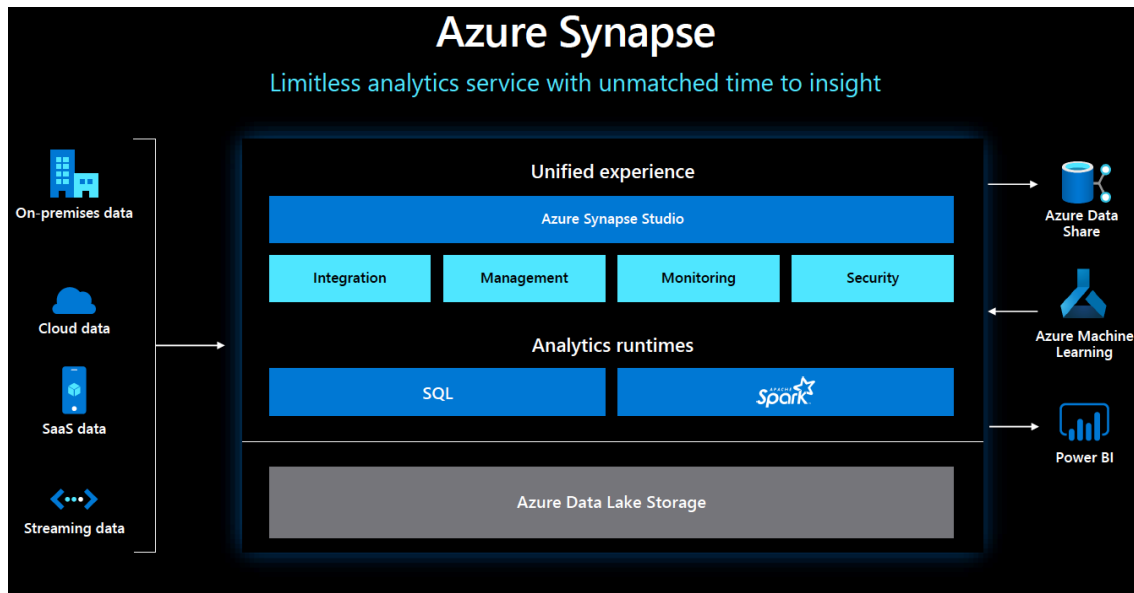
Cloudera-NVIDIA partnership



Microsoft

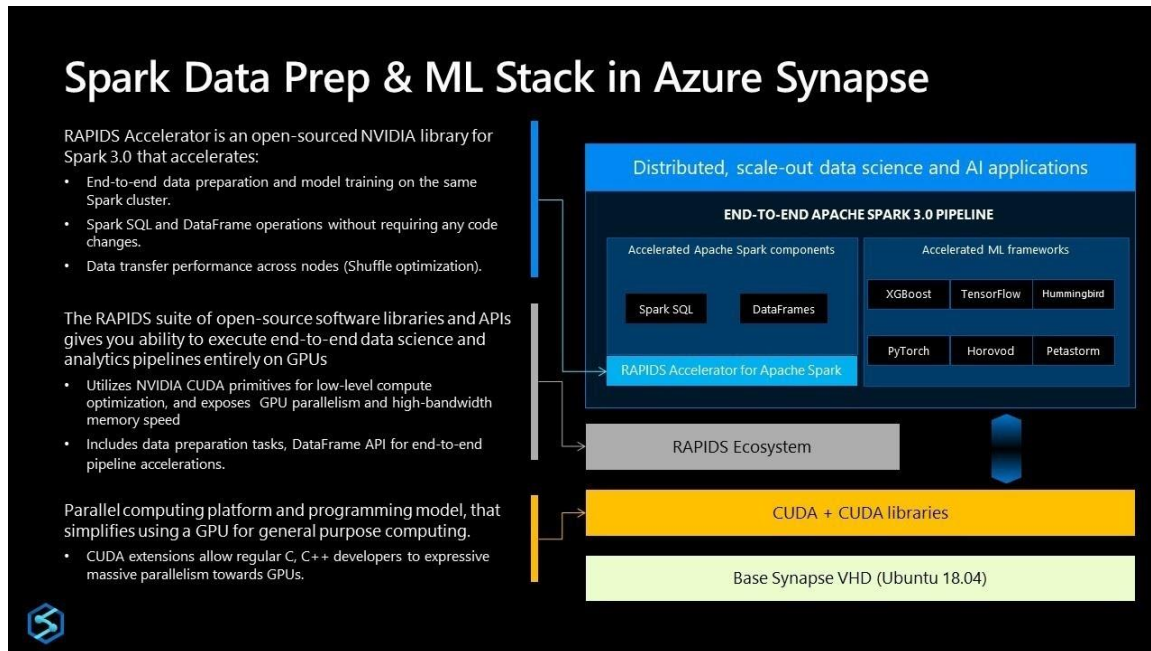
[Azure Synapse](#) is a large-scale analytics service, cohesively combining the convenience of data integration, data warehousing, and big data analytics capabilities. Azure Synapse enables developers, data engineers, and data scientists to work with large volumes of data on multiple levels and helps ingest, explore, prepare, manage, and serve data. Azure Synapse offers:

- ▶ Data processing engines: [Apache Spark pool](#), [Serverless SQL pool](#), [Dedicated SQL pool](#).
- ▶ [Separation of storage and compute](#).
- ▶ No-copy data sharing from [Azure Cosmos DB](#) via [Azure Synapse Link](#).
- ▶ Machine learning with [Azure Machine Learning](#), [Cognitive Services](#), [Apache Spark MLlib](#), or Microsoft Machine Learning for Apache Spark ([mmlspark](#)).
- ▶ Data governance and data cataloging with [Azure Purview](#).
- ▶ Integration with [Azure Data Factory](#) for end-to-end workflows, data movement, and creating automatable data pipelines.



With Apache Spark™ deployments tuned for NVIDIA GPUs, plus pre-installed libraries, Azure Synapse Analytics offers a simple way to leverage GPUs to power a variety of data processing and machine learning tasks. With built-in support for [NVIDIA's RAPIDS](#) acceleration, the Azure Synapse version of GPU-accelerated Spark offers gains of 2x on standard analytical benchmarks compared to running on CPUs, all without any code changes. Additionally, for machine learning workloads Azure Synapse offers Microsoft's [Hummingbird](#) out-of-box which can leverage these GPUs to offer significant acceleration on traditional ML workloads.

This GPU acceleration feature in Azure Synapse is available for [private preview by request](#).

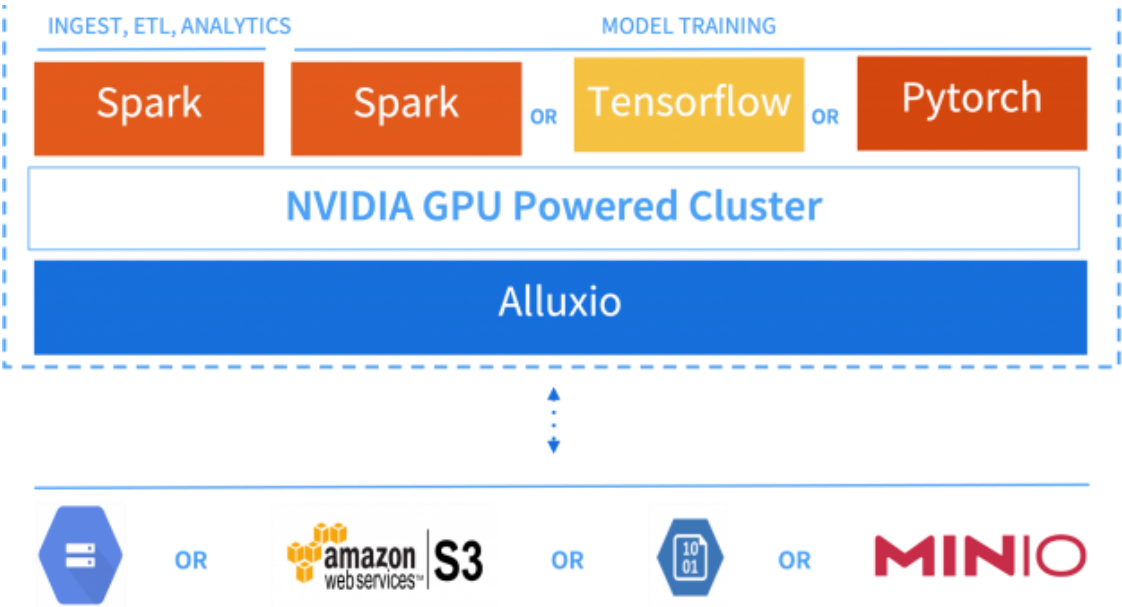


Alluxio

NVIDIA has worked with the Alluxio community to test a high-performance data orchestration system for caching large datasets and data availability for GPU processing. Apache Spark 3.x, the [RAPIDS Accelerator for Apache Spark](#), and [Alluxio](#) can be used both for:

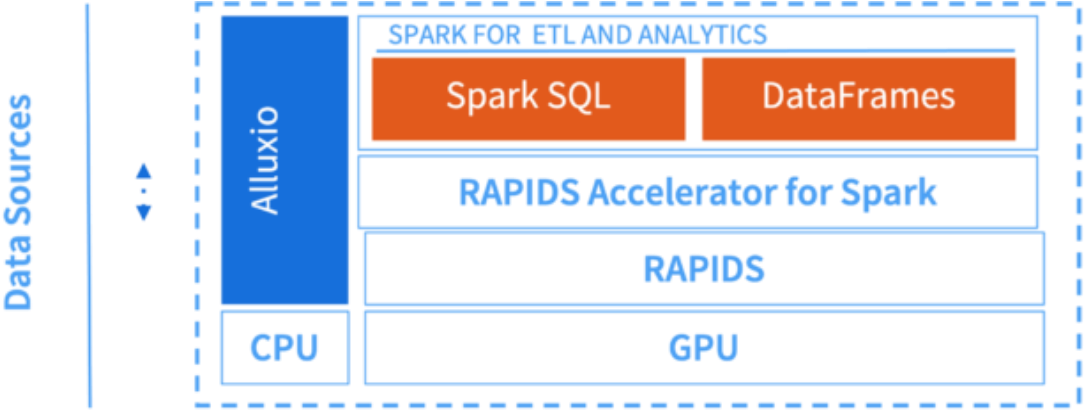
- ▶ Data analytics and business intelligence.
- ▶ Data preprocessing and feature engineering for data science.
- ▶ For model training or inference, Spark and distributed TensorFlow or PyTorch on GPU clusters all benefit from I/O acceleration using a distributed platform-agnostic orchestration layer.

Alluxio's data orchestration layer can be used as a distributed cache for multiple data sources shared across multiple steps of a data pipeline. Alluxio is agnostic to the platform, whether it's managed YARN or Kubernetes on-premises or in the cloud.



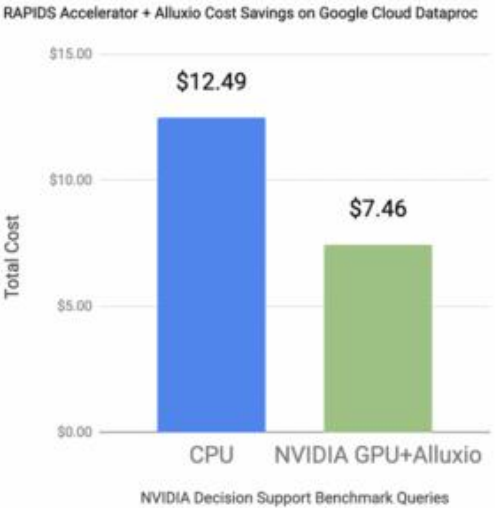
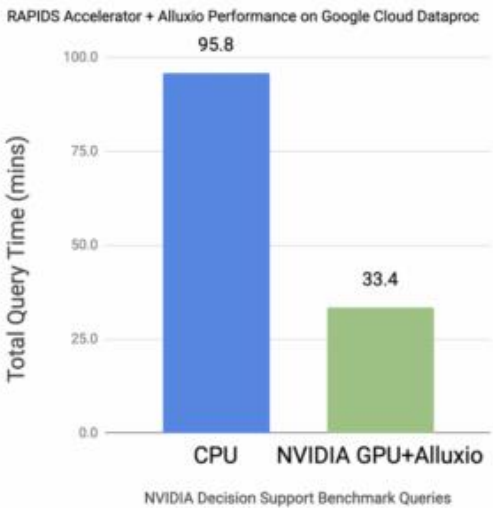
Apache Spark 3.x with RAPIDS Accelerator for Apache Spark is able to parallelize computation of a Spark SQL or DataFrame job on a GPU-accelerated cluster. However, a significant portion of such a job is reading the data from a remote data source. If data access is slow, I/O can dominate the end-to-end application runtime.

Using Alluxio as a data source for Spark applications requires no code changes to benefit from I/O acceleration using the data orchestration layer. The recommended software stack on GPU enabled instances is shown below, with Alluxio utilizing CPUs and local storage media such as NVMe for managing cached data while Spark utilizes GPU resources for computation.



The NVIDIA Decision Support (NDS) dataset and queries derived from a popular data analytics benchmark suite were used for an evaluation of GPU processing with RAPIDS Accelerator for Apache Spark and Alluxio. These benchmark queries operated on a 3 Terabyte (TB) dataset in Parquet format stored in a Google Cloud Storage bucket.

The following charts show that an NVIDIA GPU cluster with Alluxio has a nearly 2x improvement in performance when comparing the total elapsed time across the 90 NDS queries, and 70% better return on investment (ROI) compared to a CPU cluster. Google Cloud Dataproc was used to deploy services on compute instances for both CPU and GPU clusters.



CPU Hardware Config: Master: n1-standard-4, Slave: 4 x n1-standard-32 (128 cores, 480GB RAM), GPU Hardware Config: Master: n1-standard-4, Slave: 4 x n1-standard-32 (128 cores, 480GB RAM + 16 x T4), CPU Cloud Costs: \$7.82/hr (Based on [standard pricing on 4 x n1-standard-32](#) + [Dataproc Pricing](#)), GPU Cloud Costs: \$13.41/hr (Based on [standard pricing on 4 x n1-standard-32 + 16 x T4](#) + [DataProc Pricing](#))

Most of these improvements can be attributed to Alluxio's ability to cache the large datasets, and thereby eliminate the need for repeated access to cloud storage. Data scientists who perform multiple tasks across the data science life cycle such as data ingestion, data preparation and data exploration can benefit from increased data processing capabilities to improve performance and reduce costs.

For information on how to set up the RAPIDS Accelerator for Apache Spark 3.x on Databricks, Google, AWS, Alluxio, on Premise YARN or Kubernetes see [Getting-Started - spark-rapids](#). For Cloudera and Azure Synapse, see the provider documentation.

Configuration

The Spark shell and [./bin/spark-submit](#) support loading configuration properties dynamically, via command line options, such as `--conf`, or by reading configuration options from `conf/spark-defaults.conf`. (Refer to the [Spark Configuration Guide](#) for an overview and details on Spark configurations.)

On startup use: `--conf [conf key]=[conf value]`. For example:

```
${SPARK_HOME}/bin/spark --jars 'rapids-4-spark_2.12-0.5.0.jar,cudf-0.19.2-cuda10-1.jar' \
  --conf spark.plugins=com.nvidia.spark.SQLPlugin \
  --conf spark.rapids.sql.incompatibleOps.enabled=true
```

At runtime use: `spark.conf.set("[conf key]", [conf value])`. For example:

```
scala> spark.conf.set("spark.rapids.sql.incompatibleOps.enabled", true)
```

All configs can be set on startup, but some configs, especially for shuffle, will not work if they are set at runtime.

GPU Scheduling

You can use `--conf` key value pairs to request GPUs and assign them to tasks. The exact configuration you use will vary depending on your cluster manager. Here are a few of the configuration key value properties for assigning GPUs:

- Request your executor to have a GPU. The executor can only use one GPU, so this value should always be 1:

```
--conf spark.executor.resource.gpu.amount=1
```

- Specify the number of GPUs per task:

```
--conf spark.task.resource.gpu.amount=1
```

- Specify a GPU discovery script (required on YARN and K8S):

```
--conf spark.executor.resource.gpu.discoveryScript=./getGpusResources.sh
```

Note that `spark.task.resource.gpu.amount` can be a decimal amount, so if you want multiple tasks to be run on an executor at the same time and assigned to the same GPU you can set this to a decimal value less than 1. You would want this setting to correspond to the `spark.executor.cores` setting. For instance, if you have `spark.executor.cores=2` which would allow two tasks to run on each executor and you want those 2 tasks to run on the same GPU then you would set `spark.task.resource.gpu.amount=0.5`.

Advanced Configuration

Beyond these configurations, we have other plugin-specific configurations that may help performance as long as certain requirements are met. These configurations control what operations can run on the GPU. Enabling these allows more things to be optimized and run on the GPU. For more details on configuration refer to the [RAPIDS Accelerator for Spark Configuration](#).

Tuning General Recommendations

Number of Executors

The RAPIDS Accelerator plugin only supports a one-to-one mapping between GPUs and executors.

Number of Tasks per Executor

Running multiple, concurrent tasks per executor is supported in the same manner as standard Apache Spark. For example, if the cluster nodes each have 24 CPU cores and 4 GPUs then setting `spark.executor.cores=6` will run each executor with 6 cores and 6 concurrent tasks per executor, assuming the default setting of one core per task, i.e.:
`spark.task.cpus=1`.

It is recommended to run more than one concurrent task per executor as this allows overlapping I/O and computation. For example one task can be communicating with a distributed file system to fetch an input buffer while another task is decoding an input buffer on the GPU. Configuring too many concurrent tasks on an executor can lead to excessive I/O and overload host memory. Counter-intuitively leaving some CPU cores idle may actually speed up your overall job. We typically find that two times the number of concurrent GPU tasks is a good starting point.

The [number of concurrent tasks running on a GPU](#) is configured separately.

Input Files

Fewer large input files are better than lots of small files. GPUs process data much more efficiently when they have a large amount of data to process in parallel. Loading data from fewer, large input files will perform better than loading data from many small input files. Ideally input files should be on the order of a few gigabytes rather than megabytes or smaller.

Note that the GPU can encode Parquet and ORC data much faster than the CPU, so the costs of writing large files can be significantly lower.

Input Partition Size

Many queries can benefit from using a larger input partition size than the default setting of 128 MB. This allows the GPU to process more data at once, amortizing overhead costs across a larger set of data. Many queries perform better when this is set to 256MB or 512MB. Note that setting this value too high can cause tasks to fail with GPU out of memory errors. The configuration settings that control the input partition size depend upon the method used to read the input data:

► DataSource API: `spark.sql.files.maxPartitionBytes`

► Hive API:

```
spark.hadoop.mapreduce.input.fileinputformat.split.minsize
```

For more recommendations refer to [Tuning - spark-rapids](#).

Monitoring Using the Physical Plan

The RAPIDS Accelerator for Spark requires no API changes from the user, and it will replace SQL operations it supports with GPU operations. In order to see what operations were replaced with GPU operations, you can print out the physical plan for a DataFrame by calling the `explain` method, all of the operations prefixed with GPU take place on GPUs.

Now, compare the physical plan for a DataFrame with GPU processing for some of the same queries we looked at in Chapter 2. In the physical plan below, the DAG consists of a `GpuBatchScan`, a `GpuFilter` on `hour`, and a `GpuProject` (selecting columns) on `hour`, `fare_amount`, and `day_of_week`. With CPU processing it consisted of a `FileScan`, `Filter`, and a `Project`.

```
// select and filter are narrow transformations
df.select($"hour", $"fare_amount").filter($"hour" === "0.0" ).show(2)

result:
+----+-----+
|hour|fare_amount|
+----+-----+
| 0.0|         10.5|
| 0.0|         12.5|
```

```
+-----+-----+
df.select($"hour", $"fare_amount").filter($"hour" === "0.0" ).explain

result:
== Physical Plan ==
*(1) GpuColumnarToRow false<
+- !GpuProject [hour#10, fare_amount#9]
    +- GpuCoalesceBatches TargetSize(1000000,2147483647)
        +- !GpuFilter (gpuisnotnull(hour#10) AND (hour#10 = 0.0))
            +- GpuBatchScan[fare_amount#9, hour#10] GpuCSVScan Location:
InMemoryFileIndex[s3a://spark-taxi-dataset/raw-small/train], ReadSchema:
struct<fare_amount:double,hour:double>
```

Notice how most of the nodes in the original plan have been replaced with GPU versions. The RAPIDS Accelerator inserts data format conversion nodes, like `GpuColumnarToRow` and `GpuRowToColumnar` to convert between columnar processing for nodes that will execute on the GPU and row processing for nodes that will execute on the CPU. If some parts of your query did not run on the GPU, to see why set the config `spark.rapids.sql.explain` to `NOT_ON_GPU`. The output will be logged to the driver's log or to the screen in interactive mode.



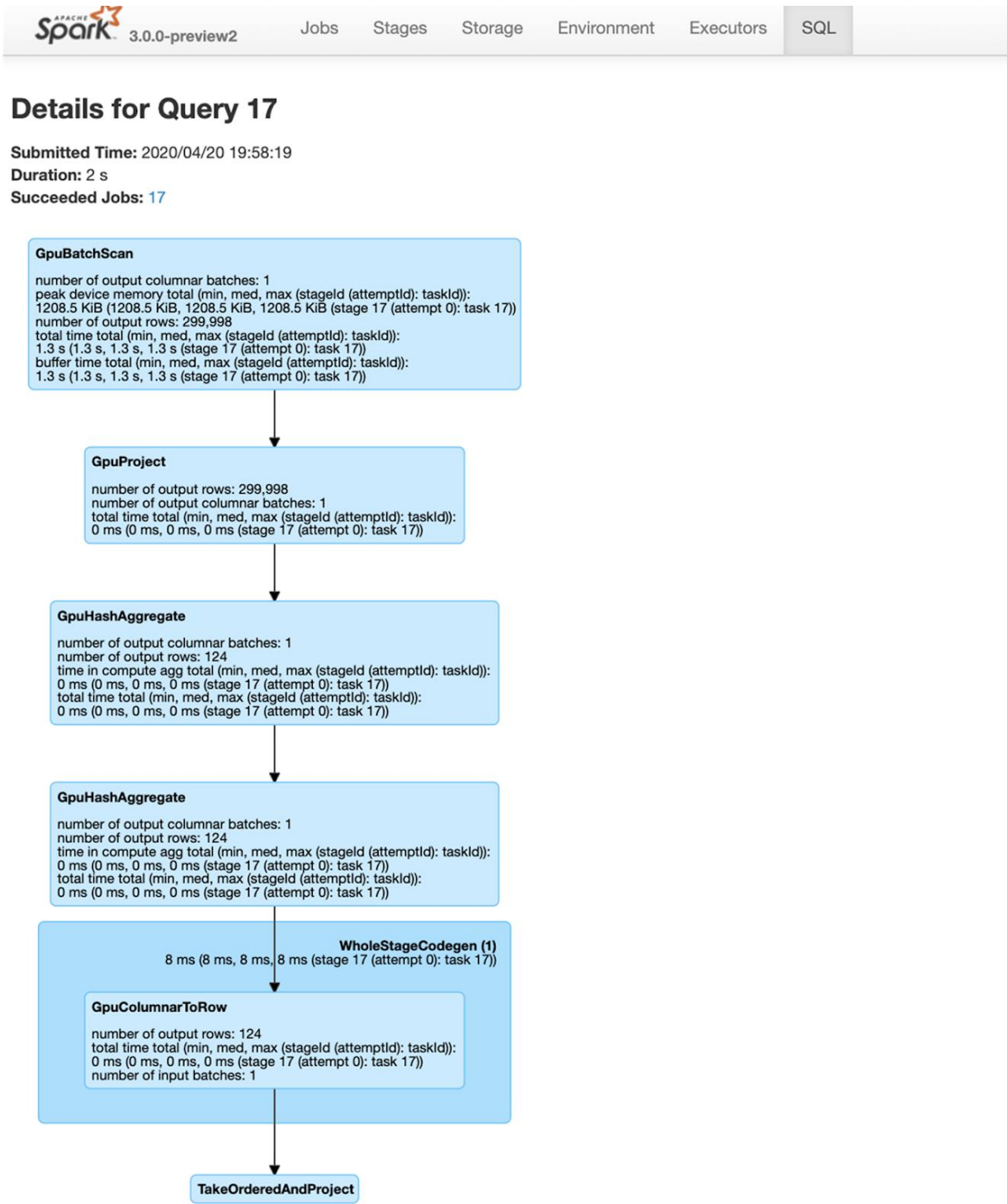
Note: When AQE is on, the explain plan may not show you the real GPU plan. If that is the case, you should [use the Spark UI SQL tab to find the real GPU plan](#).

Monitoring Using the Spark Web UI

SQL Tab

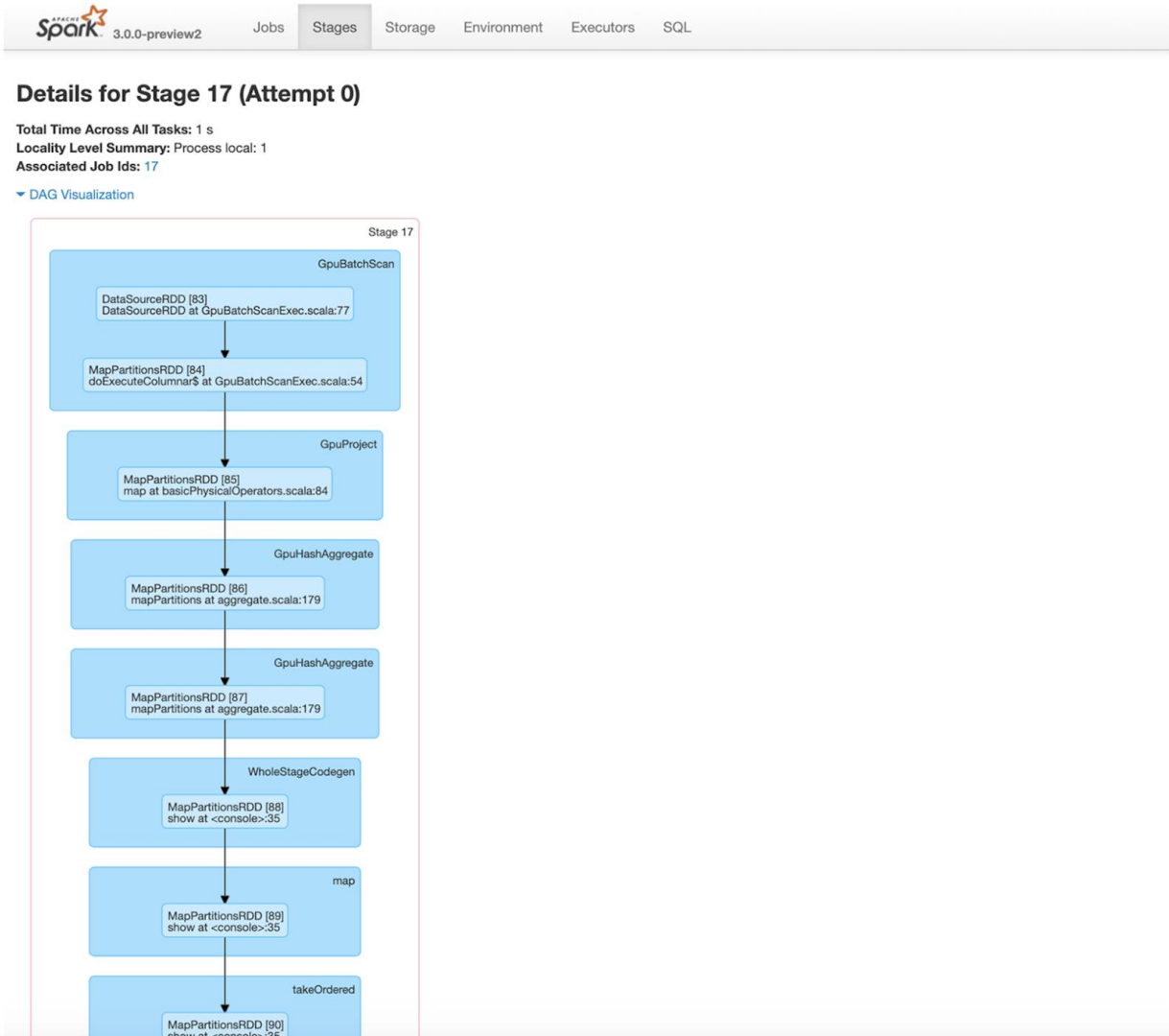
The easiest way to see what is running on the GPU is to look at the SQL tab in the Spark Web UI. In the DAG diagram from the SQL Tab for the query below, we see that the physical plan consists of a `GPUBatchScan`, `GPUProject`, `GPUHashAggregate`, and a `GPUHashAggregate`. With CPU processing Spark performs a hash aggregation for each partition before shuffling the data in the Exchange for the wide transformation. After the exchange, there is a hash aggregation of the previous sub-aggregations. Note that for GPU processing the Exchange shuffle has been avoided.

```
val df3 = df2.groupBy("month").count
              .orderBy(asc("month")).show(5)
```



Stages Tab

You can use the stage details page to view a stage details DAG, where the blue vertices (boxes) represent the RDDs or DataFrames and the edges (arrows between boxes) represent the operation applied to a DataFrame. Note that since there is no shuffle, this query takes place in one stage.



Environment Tab

You can use the Environment tab to view and check whether the GPU configuration for your Spark properties have been set correctly, for example the `spark.executor.resource.gpu.amount` and `spark.executor.resource.gpu.discoveryScript` properties. Here you can also view the System Properties classpath entries to check that the plugin jars are in the JVM classpath.

Table 1. Spark Properties

Name	Value
<code>spark.executor.resource.gpu.amount</code>	1
<code>spark.executor.resource.gpu.discoveryScript</code>	<code>/home/ubuntu/getGpusResources.sh</code>

Executors Tab

You can use the **Executors** tab to see which resources have been allocated for the executors for your application. In this instance, one GPU has been allocated.

Executors

▼ Show Additional Metrics

- ☐ Select All
- ☐ On Heap Memory
- ☐ Off Heap Memory
- ☒ Resources

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(2)	0	0.0 B / 13 GiB	0.0 B	8	0	0	839	839	5.4 min (1 s)	2.8 GiB	1.2 GiB	1.2 GiB	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	0
Total(2)	0	0.0 B / 13 GiB	0.0 B	8	0	0	839	839	5.4 min (1 s)	2.8 GiB	1.2 GiB	1.2 GiB	0

Executors

Show 20 entries

Search:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Resources	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs
0	10.19.183.153:45123	Active	0	0.0 B / 7.8 GiB	0.0 B	8	gpu: [1]	0	0	839	839	5.4 min (1 s)	2.8 GiB	1.2 GiB	1.2 GiB	stdout stderr
driver	spark-gashen:33965	Active	0	0.0 B / 5.2 GiB	0.0 B	0		0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	

Summary

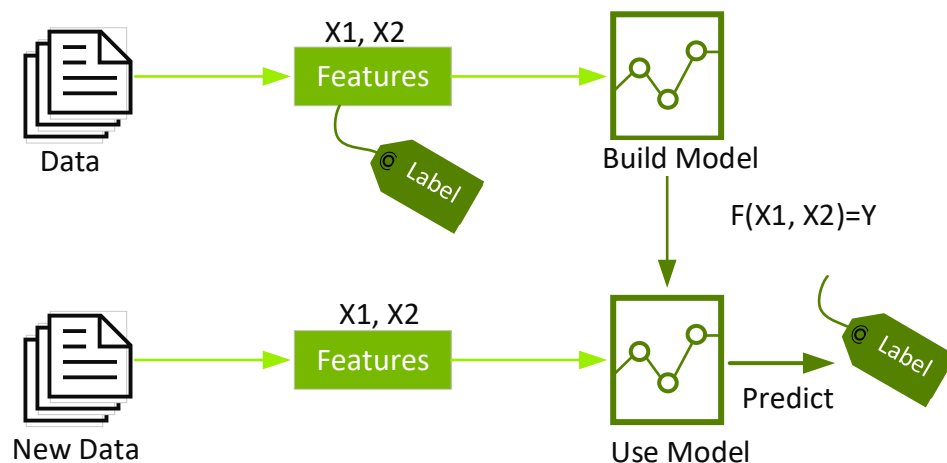
In this chapter, we covered the basics of getting started using the new RAPIDS APIs Plugin for Apache Spark 3.x that leverages GPUs to accelerate processing. For more information refer to the [RAPIDS Accelerator for Spark guide](#).

Chapter 5: Predicting Housing Prices Using Apache Spark Machine Learning

Zillow is one of the largest marketplaces for real estate information in the U.S. and a leading example of impactful machine learning (ML). [Zillow Research](#) uses ML models that analyze hundreds of data points on each property to estimate home values and predict market changes. In this chapter, we cover how to use Apache Spark ML Random Forest Regression to predict the median sales prices for homes in a region. Note that currently only XGBoost is GPU-Accelerated in Spark ML, which we will cover in the next chapter.

Classification and Regression

Classification and regression are two categories of supervised machine learning algorithms. Supervised ML, also called predictive analytics, uses algorithms to find patterns in labeled data and then uses a model that recognizes those patterns to predict the labels on new data. Classification and regression algorithms take a dataset with labels (also called the target outcome) and features (also called properties) and learn how to label new data based on those data features.



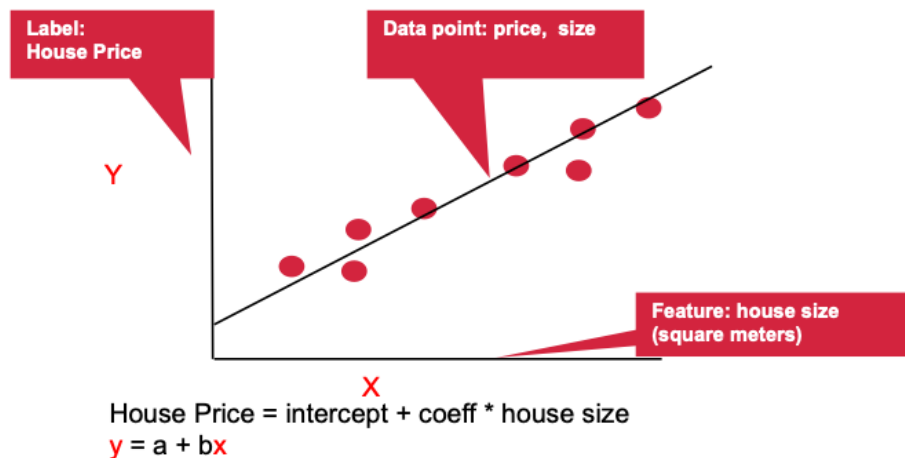
Classification identifies which category an item belongs to, such as whether a credit card transaction is legitimate. Regression predicts a continuous numeric value like a house price, for example.

Regression

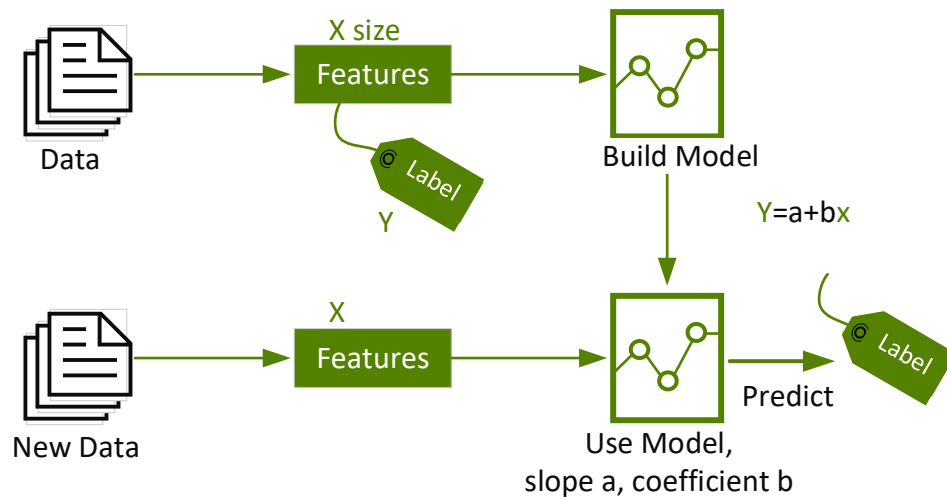
Regression estimates the relationship between a target outcome dependent variable (the label) and one or more independent variables (the features). Regression can be used to analyze the strength of the relationship between the label and the feature variables, determine how much the label changes with an adjustment in one or more feature variables, and predict trends between the label and feature variables.

Let's go through a linear regression example of housing prices, given historical house prices and features of houses (square feet, number of bedrooms, location, etc.):

- What are we trying to predict?
This is the label: the house price
- What are the data properties that you can use to predict?
These are the features: to build a regression model, you extract the features of interest that have the strongest relationship with the label and contribute the most to the prediction.
In the following example, we'll use the size of the house.



Linear regression models the relationship between the Y "Label" and the X "Feature", in this case the relationship between the house price and size, with the equation: $Y = \text{intercept} + (\text{coefficient} * X) + \text{error}$. The coefficient measures the impact of the feature on the label, in this case the impact of the house size on the price.



Multiple linear regression models the relationship between two or more "Features" and a "Label." For example, if we wanted to model the relationship between the price and the house size, the number of bedrooms, and the number of bathrooms, the multiple linear regression function would look like this:

$$Y_i = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p + \epsilon$$

Price = intercept + (coefficient1 size) + (coefficient2 bedrooms) + (coefficient3 * bathrooms) + error.

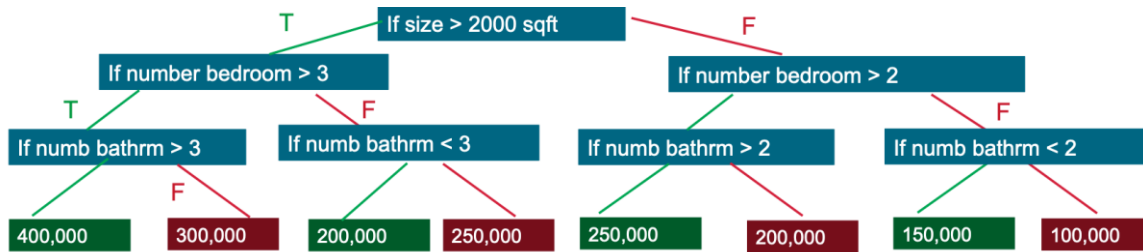
The coefficients measure the impact on the price of each of the features.

Decision Trees

Decision trees create a model that predicts the label by evaluating a set of rules that follow an if-then-else pattern. The if-then-else feature questions are the nodes, and the answers "true" or "false" are the branches in the tree to the child nodes.

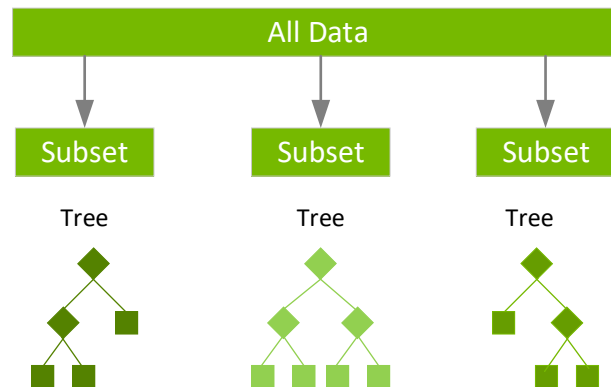
A decision tree model estimates the minimum number of true/false questions needed to assess the probability of making a correct decision. Decision trees can be used for classification to predict a category, or probability of a category, or regression to predict a continuous numeric value. Following is an example of a simplified decision tree to predict housing prices:

- Q1: If the size of the house > 2000sqft
 - T:Q2: If the number of bedrooms > 3
 - T:Q3: If the number of bathrooms is > 3
 - T: Price=\$400,000
 - F: Price=\$200,000



Random Forests

Ensemble learning algorithms combine multiple machine learning algorithms to obtain a better model. Random forest is a popular ensemble learning method for classification and regression. The algorithm builds a model consisting of multiple decision trees, based on different subsets of data at the training stage. Predictions are made by combining the output from all the trees, which reduces the variance and improves the predictive accuracy. For random forest classification, the label is predicted to be the class predicted by the majority of trees. For random forest regression, the label is the mean regression prediction of the individual trees.



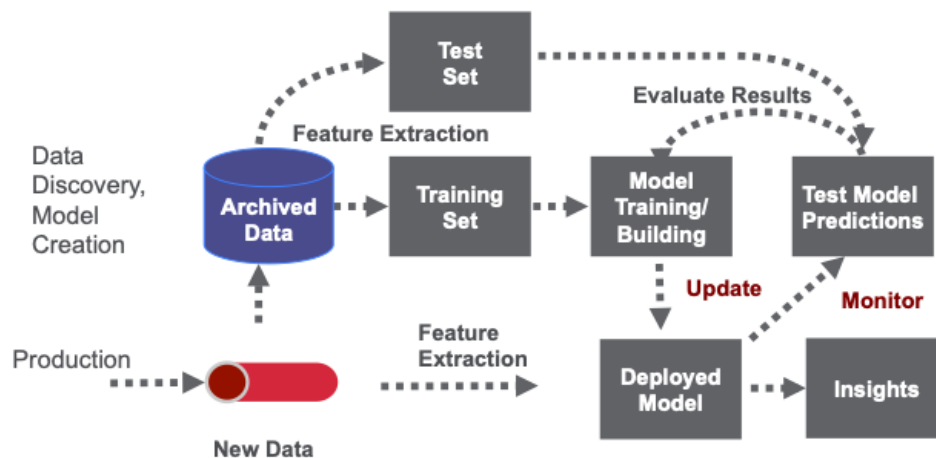
Spark provides the following algorithms for regression:

- ▶ Linear regression
- ▶ Generalized linear regression
- ▶ Decision tree regression
- ▶ Random forest regression
- ▶ Gradient-boosted tree regression
- ▶ XGBoost regression
- ▶ Survival regression
- ▶ Isotonic regression

Machine Learning Workflows

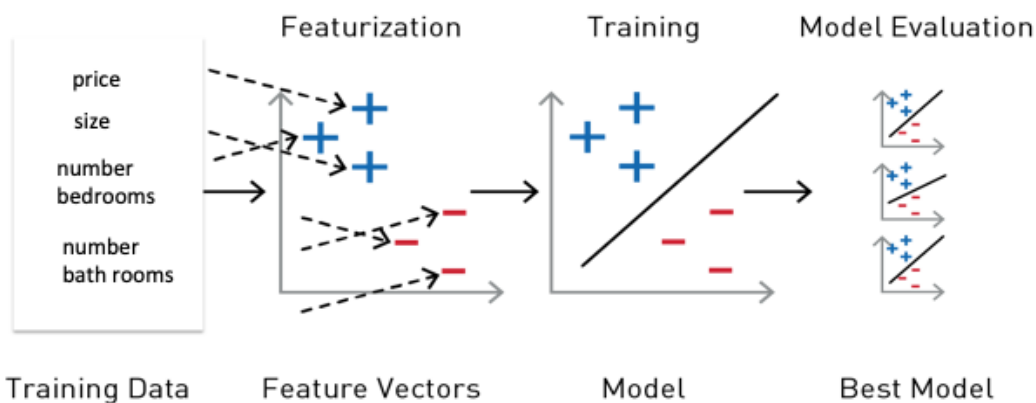
Machine learning is an iterative process which involves:

- ▶ Extracting, Transforming, Loading (ETL) and analyzing historical data in order to extract the significant features and label.
- ▶ Training, testing, and evaluating the results of ML algorithms to build a model.
- ▶ Using the model in production with new data to make predictions.
- ▶ Model monitoring and model updating with new data.



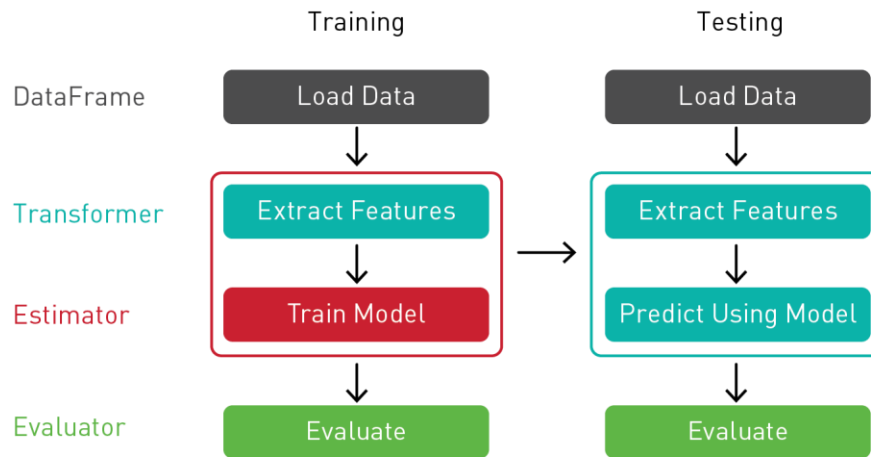
Using Spark ML Pipelines

For the features and label to be used by an ML algorithm, they must be put into a feature vector, which is a vector of numbers representing the value for each feature. Feature vectors are used to train, test, and evaluate the results of an ML algorithm to build the best model.



Reference Learning Spark

Spark ML provides a uniform set of high-level APIs, built on top of DataFrames for building ML pipelines or workflows. Having ML pipelines built on top of DataFrames provides the scalability of partitioned data processing with the ease of SQL for data manipulation.



We use a Spark ML Pipeline to pass the data through transformers and extract the features, an estimator to produce the model, and an evaluator to measure the accuracy of the model.

- **Transformer:** A Transformer is an algorithm that transforms one DataFrame into another DataFrame. We'll use a Transformer to create a DataFrame with a features vector column.
- **Estimator:** An Estimator is an algorithm that can be fit on a DataFrame to produce a transformer. We'll use an estimator to train a model, and return a model Transformer, which can add a predictions column to a DataFrame with a features vector column.
- **Pipeline:** A pipeline chains multiple Transformers and Estimators together to specify an ML workflow.
- **Evaluator:** An Evaluator measures the accuracy of a trained Model on label and prediction DataFrame columns.

Example Use Case Dataset

In this example, we'll be using the California housing prices dataset from the [StatLib repository](#). This dataset contains 20,640 records based on data from the 1990 California census, with each record representing a geographic block. The following list provides a description for the attributes of the data set.

- ▶ **Median House Value:** Median house value (in thousands of dollars) for households within a block.
- ▶ **Longitude:** East/west measurement, a higher value is further west.
- ▶ **Latitude:** North/south measurement, a higher value is further north.
- ▶ **Housing Median Age:** Median age of a house within a block, lower is newer.
- ▶ **Total Rooms:** Total number of rooms within a block.
- ▶ **Total Bedrooms:** Total number of bedrooms within a block.
- ▶ **Population:** Total number of people residing within a block.
- ▶ **Households:** Total number of households in a block.
- ▶ **Median Income:** Median income for households within a block of houses (measured in tens of thousands of dollars).

To build a model, you extract the features that most contribute to the prediction. In order to make some of the features more relevant for predicting the median house value, instead of using totals we'll calculate and use these ratios: rooms per house=total rooms/households, people per house=population/households, and bedrooms per rooms=total bedrooms/total rooms.

In this scenario, we use random forest regression on the following label and features:

- ▶ Label → median house value
- ▶ Features → {"median age", "median income", "rooms per house", "population per house", "bedrooms per room", "longitude", "latitude" }

Load the Data from a File into a DataFrame

Load Data → **Dataframe**

The first step is to load our data into a DataFrame. In the following code, we specify the data source and schema to load into a dataset.

```
import org.apache.spark._
import org.apache.spark.ml._
import org.apache.spark.ml.feature._
import org.apache.spark.ml.regression._
import org.apache.spark.ml.evaluation._
import org.apache.spark.ml.tuning._
```

```

import org.apache.spark.sql._
import org.apache.spark.sql.functions._
import org.apache.spark.sql.types._
import org.apache.spark.ml.Pipeline

val schema = StructType(Array(
  StructField("longitude", FloatType, true),
  StructField("latitude", FloatType, true),
  StructField("medage", FloatType, true),
  StructField("totalrooms", FloatType, true),
  StructField("totalbdrms", FloatType, true),
  StructField("population", FloatType, true),
  StructField("houshlds", FloatType, true),
  StructField("medincome", FloatType, true),
  StructField("medhvalue", FloatType, true)
))

var file = "/path/cal_housing.csv"

var df = spark.read.format("csv").option("inferSchema", "false").schema(schema).load(file)

df.show
:
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|longitude|latitude|medage|totalrooms|totalbdrms|population|houshlds|medincome|medhvalue|
result
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| -122.23| 37.88| 41.0| 880.0| 129.0| 322.0| 126.0| 8.3252| 452600.0|
| -122.22| 37.86| 21.0| 7099.0| 1106.0| 2401.0| 1138.0| 8.3014| 358500.0|
| -122.24| 37.85| 52.0| 1467.0| 190.0| 496.0| 177.0| 7.2574| 352100.0|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

In the following code example, we use the `DataFrame withColumn()` transformation, to add columns for the ratio features: rooms per house=total rooms/households, people per house=population/households, and bedrooms per rooms=total bedrooms/total rooms. We then cache the `DataFrame` and create a temporary view for better performance and ease of using SQL.

```

// create ratios for features
df = df.withColumn("roomsPhouse", col("totalrooms")/col("houshlds"))
df = df.withColumn("popPhouse", col("population")/col("houshlds"))
df = df.withColumn("bedrmsPRoom", col("totalbdrms")/col("totalrooms"))

df=df.drop("totalrooms","houshlds", "population" , "totalbdrms")

df.cache
df.createOrReplaceTempView("house")
spark.catalog.cacheTable("house")

```

Summary Statistics

Spark DataFrames include some [built-in functions](#) for statistical processing. The `describe()` function performs summary statistics calculations on numeric columns and returns them as a DataFrame. The following code shows some statistics for the label and some features.

```
df.describe("medincome", "medhvalue", "roomsPhouse", "popPhouse").show
```

```
result:
```

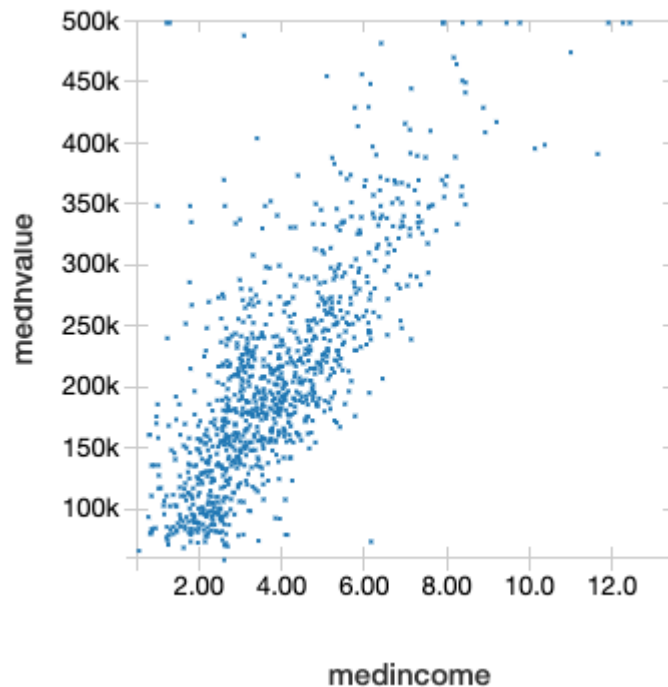
summary	medincome	medhvalue	roomsPhouse	popPhouse
count	20640	20640	20640	20640
mean	3.8706710030346416	206855.81690891474	5.428999742190365	3.070655159436382
stddev	1.8998217183639696	115395.61587441359	2.4741731394243205	10.38604956221361
min	0.4999	14999.0	0.8461538461538461	0.6923076923076923
max	15.0001	500001.0	141.9090909090909	1243.3333333333333

The `DataFrame Corr()` function calculates the Pearson correlation coefficient of two columns of a DataFrame. This measures the statistical relationship between two variables based on the method of covariance. Correlation coefficient values range from 1 to -1, where 1 indicates a perfect positive relationship, -1 indicates a perfect negative relationship, and a 0 indicates no relationship. Below we see that the median income and the median house value have a positive correlation relationship.

```
df.select(corr("medhvalue", "medincome")).show()
```

corr(medhvalue, medincome)
0.688075207464692

The following scatterplot of the median house value on the Y axis and median income on the X axis shows that they are linearly related to each other.



The following code uses the DataFrame `randomSplit` method to randomly split the Dataset into two, with 80% for training and 20% for testing.

```
val Array(trainingData, testData) = df.randomSplit(Array(0.8, 0.2), 1234)
```

Feature Extraction and Pipelining

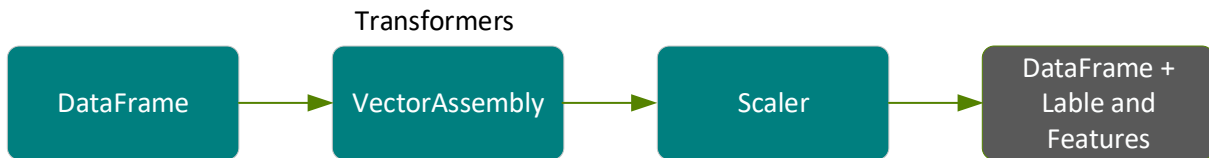
The following code creates a `VectorAssembler` (a transformer), which will be used in a pipeline to combine a given list of columns into a single feature vector column.

```
val featureCols = Array("medage", "medincome", "roomsPhouse", "popPhouse",  
    "bedrmsPRoom", "longitude", "latitude")  
  
//put features into a feature vector column  
val assembler = new  
    VectorAssembler().setInputCols(featureCols).setOutputCol("rawfeatures")
```

The following code creates a **StandardScaler** (a transformer), which will be used in a pipeline to standardize features by scaling to unit variance using DataFrame column summary statistics.

```
val scaler = new
StandardScaler().setInputCol("rawfeatures").setOutputCol("features").setWith
Std(true).setWithMean(true)
```

The result of running these transformers in a pipeline will be to add a scaled features column to the dataset as shown in the following figure.



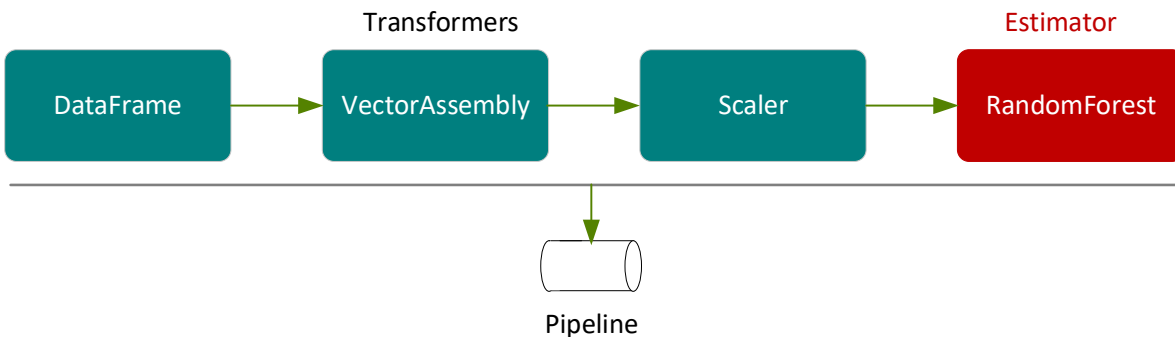
The final element in our pipeline is a **RandomForestRegressor** (an estimator), which trains on the vector of features and label, and then return a **RandomForestRegressorModel** (a transformer).

```
val rf = new
RandomForestRegressor().setLabelCol("medhvalue").setFeaturesCol("features")
```

In the following example, we put the **VectorAssembler**, **Scaler** and **RandomForestRegressor** in a **Pipeline**. A pipeline chains multiple transformers and estimators together to specify an ML workflow for training and using a model.

```
val steps = Array(assembler, scaler, rf)

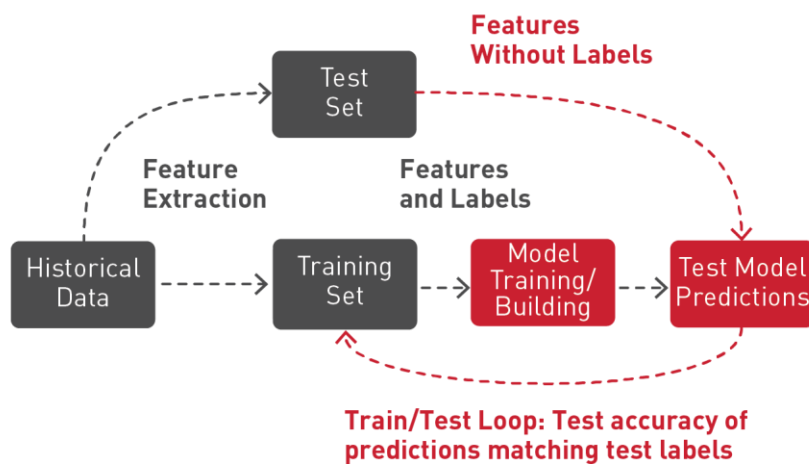
val pipeline = new Pipeline().setStages(steps)
```



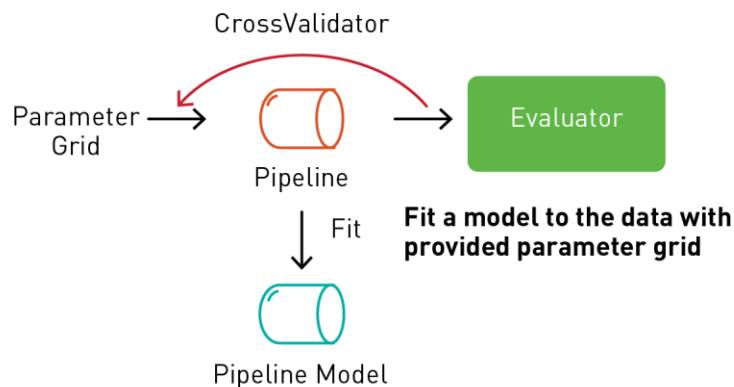
Train the Model

Spark ML supports a technique called k-fold cross-validation to try out different combinations of parameters in order to determine which parameter values of the ML algorithm produce the best model. With k-fold cross-validation, the data is randomly split into k partitions. Each partition is used once as the test dataset, while the rest are used for training. Models are then generated using the training sets and evaluated with the testing sets, resulting in k model accuracy measurements. The model parameters leading to the highest accuracy measurements produce the best model.

ML Cross-Validation Process



Spark ML supports k-fold cross-validation with a transformation/estimation pipeline which tries out different combinations of parameters, using a process called grid search, where you set up the parameters to test in a cross-validation workflow.



```
val cvModel = crossval.fit(ntrain)
```

The following code uses a **ParamGridBuilder** to construct the parameter grid for the model training. We define a **RegressionEvaluator**, which will evaluate the model by comparing the test medhvalue column with the test prediction column. We use a **CrossValidator** for model selection. The **CrossValidator** uses the pipeline, the parameter grid, and the evaluator to fit the training dataset and return the best model. The **CrossValidator** uses the **ParamGridBuilder** to iterate through the **maxDepth**, **maxBins**, and **numTrees** parameters of the **RandomForestRegressor** estimator and to evaluate the models, repeating three times per parameter value for reliable results.

```
val paramGrid = new ParamGridBuilder()
    .addGrid(rf.maxBins, Array(100, 200))
    .addGrid(rf.maxDepth, Array(2, 7, 10))
    .addGrid(rf.numTrees, Array(5, 20))
    .build()

val evaluator = new RegressionEvaluator()
    .setLabelCol("medhvalue")
    .setPredictionCol("prediction")
    .setMetricName("rmse")

val crossvalidator = new CrossValidator()
    .setEstimator(pipeline)
    .setEvaluator(evaluator)
    .setEstimatorParamMaps(paramGrid)
    .setNumFolds(3)

// fit the training data set and return a model
val pipelineModel = crossvalidator.fit(trainingData)
```

Next, we can get the best model in order to print out the feature importances. The results show that the median income, population per house, and the longitude are the most important features.

```
val featureImportances = pipelineModel
    .bestModel.asInstanceOf[PipelineModel]
    .stages(2)
    .asInstanceOf[RandomForestRegressionModel]
    .featureImportances

assembler.getInputCols
    .zip(featureImportances.toArray)
    .sortBy(_._2)
    .foreach { case (feat, imp) =>
        println(s"feature: $feat, importance: $imp") }

result:
```

```
feature: medincome, importance: 0.4531355014139285
feature: popPhouse, importance: 0.12807843645878508
feature: longitude, importance: 0.10501162983981065
feature: latitude, importance: 0.1044621179898163
feature: bedrmsPRoom, importance: 0.09720295935509805
feature: roomsPhouse, importance: 0.058427239343697555
feature: medage, importance: 0.05368211559886386
```

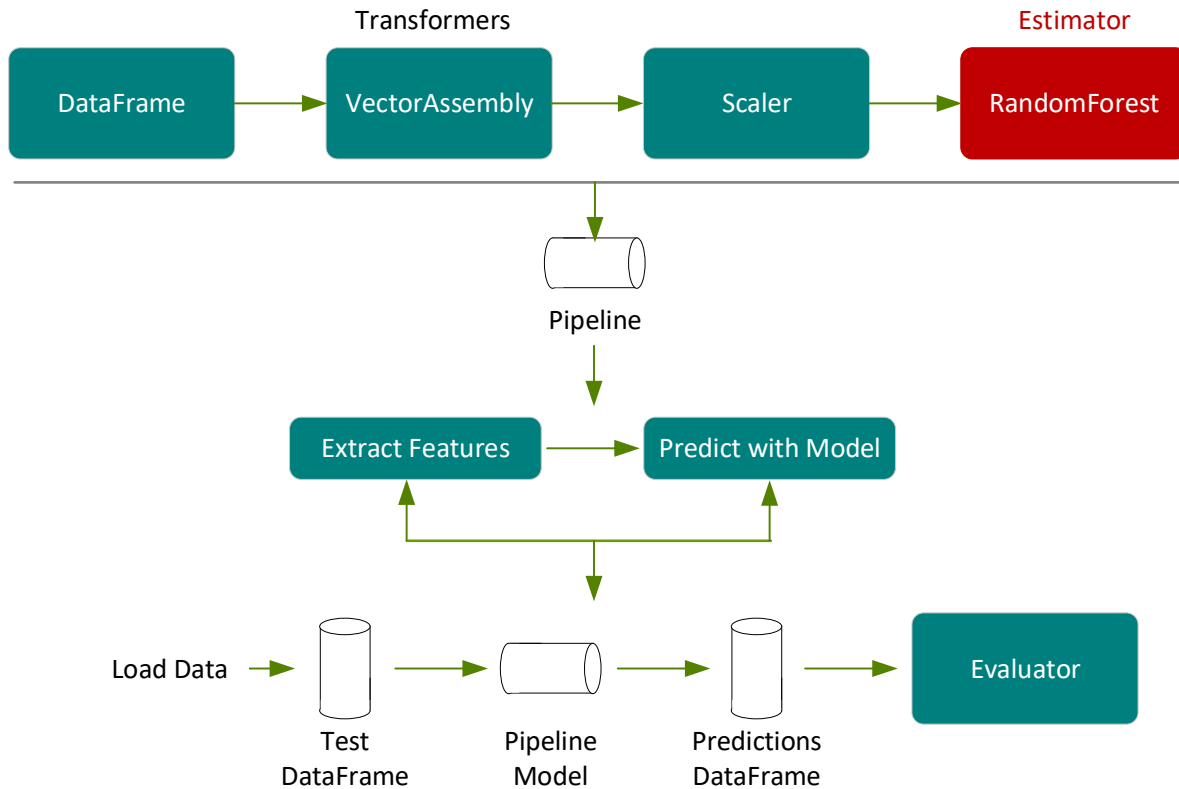
In the following example we get the parameters for the best random forest model produced, using the cross-validation process, which returns: max depth of 2, max bins of 50 and 5 trees.

```
val bestEstimatorParamMap = pipelineModel
    .getEstimatorParamMaps
    .zip(pipelineModel.avgMetrics)
    .maxBy(_._2)
    ._1
println(s"Best params:\n$bestEstimatorParamMap")

result:
  rfr_maxBins: 50,
  rfr_maxDepth: 2,
  rfr_-numTrees: 5
```


Predictions and Model Evaluation

Next we use the test DataFrame, which was a 20% random split of the original DataFrame, and was not used for training, to measure the accuracy of the model.



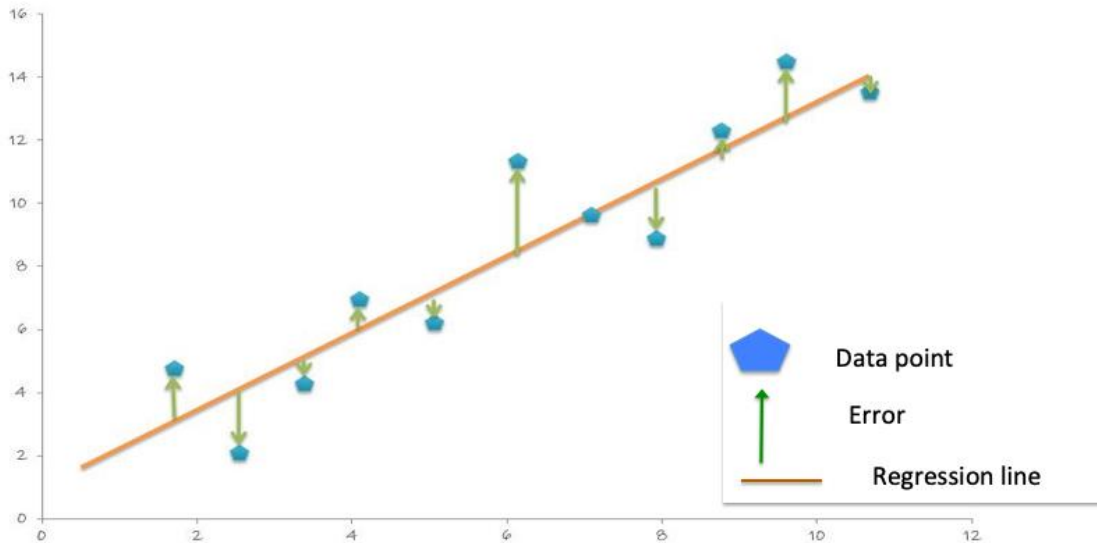
In the following code we call `transform` on the pipeline model, which will pass the test DataFrame, according to the pipeline steps, through the feature extraction stage, estimate with the random forest model chosen by model tuning, and then return the predictions in a column of a new DataFrame.

```
val predictions = pipelineModel.transform(testData)
predictions.select("prediction", "medhvalue").show(5)
```

result:

```
+-----+-----+
|      prediction|medhvalue|
+-----+-----+
|104349.59677450571| 94600.0|
| 77530.43231856065| 85800.0|
|111369.71756877871| 90100.0|
| 97351.87386020401| 82800.0|
+-----+-----+
```

With the predictions and labels from the test data, we can now evaluate the model. To evaluate the linear regression model, you measure how close the predictions values are to the label values. The error in a prediction, shown by the green lines below, is the difference between the prediction (the regression line Y value) and the actual Y value, or label. (Error = prediction-label).



The Mean Absolute Error (MAE) is the mean of the absolute difference between the label and the model's predictions. The absolute removes any negative signs.

$$\text{MAE} = \frac{\sum(\text{absolute}(\text{prediction} - \text{label}))}{\text{number of observations}}$$

The Mean Square Error (MSE) is the sum of the squared errors divided by the number of observations. The squaring removes any negative signs and also gives more weight to larger differences.

$$\text{MSE} = \frac{\sum(\text{squared}(\text{prediction} - \text{label}))}{\text{number of observations}}$$

The Root Mean Squared Error (RMSE) is the square root of the MSE. RMSE is the standard deviation of the prediction errors. The Error is a measure of how far from the regression line label data points are and RMSE is a measure of how spread out these errors are.

The following code example uses the DataFrame `withColumn` transformation, to add a column for the error in prediction: `error=prediction-medhvalue`. Then we display the summary statistics for the prediction, the median house value, and the error (in thousands of dollars).

```
predictions = predictions.withColumn("error", col("prediction") -
col("medhvalue"))
```

```
predictions.select("prediction", "medhvalue", "error").show
```

result:

```
+-----+-----+-----+
| prediction|medhvalue| error|
+-----+-----+-----+
| 104349.5967745057| 94600.0| 9749.596774505713|
```

```
| 77530.4323185606| 85800.0| -8269.567681439352|
| 101253.3225967887| 103600.0| -2346.677403211302|
+-----+-----+-----+

predictions.describe("prediction", "medhvalue", "error").show
result:
+-----+-----+-----+-----+
|summary|      prediction|      medhvalue|      error|
+-----+-----+-----+-----+
|  count|           4161|           4161|           4161|
|   mean|206307.4865123929|205547.72650805095| 759.7600043416329|
| stddev|97133.45817381598|114708.03790345002| 52725.56329678355|
|    min|56471.09903814694|          26900.0|-339450.5381565819|
|    max|499238.1371374392|          500001.0|293793.71945819416|
+-----+-----+-----+-----+
```

The following code example uses the Spark `RegressionEvaluator` to calculate the MAE on the predictions DataFrame, which returns 36636.35 (in thousands of dollars).

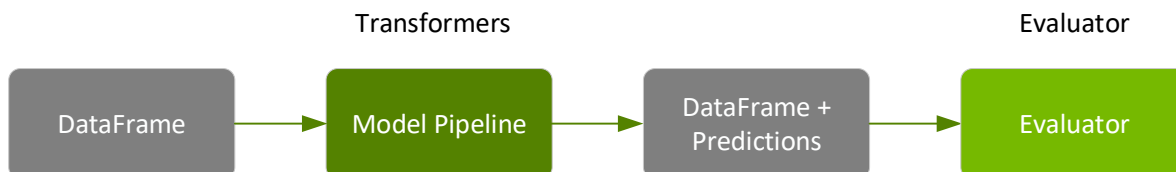
```
val maevaluator = new RegressionEvaluator()
  .setLabelCol("medhvalue")
  .setMetricName("mae")

val mae = maevaluator.evaluate(predictions)
result:
mae: Double = 36636.35
```

The following code example uses the Spark `RegressionEvaluator` to calculate the RMSE on the predictions DataFrame, which returns 52724.70.

```
val evaluator = new RegressionEvaluator()
  .setLabelCol("medhvalue")
  .setMetricName("rmse")
val rmse = evaluator.evaluate(predictions)

result:
rmse: Double = 52724.70
```



Save the Model

We can now save our fitted pipeline model to the distributed file store for later use in production. This saves both the feature extraction stage and the random forest model chosen by model tuning.

```
pipelineModel.write.overwrite().save(modeldir)
```

The result of saving the pipeline model is a JSON file for metadata and a Parquet for model data. We can reload the model with the load command; the original and reloaded models are the same:

```
val sameModel = CrossValidatorModel.load("modeldir")
```

Summary

In this chapter, we discussed Regression, Decision Trees, and Random Forest algorithms. We covered the fundamentals of Spark ML pipelines and worked through a real world example to predict median house prices.

Chapter 6: Predicting Taxi Fares Using GPU-Accelerated XGBoost

Big data is [one of the 10 major areas](#) used to improve cities. The analysis of location and behavior patterns within cities allows for optimization of traffic, better planning decisions, and smarter advertising. For example, the analysis of GPS car data enables cities to optimize traffic flows based on real-time traffic information. Telecom companies are using mobile phone location data to provide insights by identifying and predicting the location activity trends and patterns of a population in large metropolitan areas. And, the application of machine learning (ML) to geolocation data is proving instrumental in identifying patterns and trends for the telecom, travel, marketing, and manufacturing industries.

In this chapter, we'll use public New York Taxi trip data to examine regression analysis on taxi trip data as it pertains to predicting NYC taxi fares. We'll start with an overview of the XGBoost algorithm and then explore the use case.

XGBoost

[XGBoost](#), which stands for Extreme Gradient Boosting, is a scalable, distributed [gradient-boosted](#) decision tree (GBDT) machine learning library. XGBoost provides parallel tree boosting and is the leading ML library for regression, classification, and ranking problems. The RAPIDS team works closely with the Distributed Machine Learning Common (DMLC) XGBoost organization, and XGBoost now includes seamless, drop-in GPU acceleration, significantly speeding up model training and improving accuracy for better predictions.

Gradient Boosting Decision Trees (GBDTs) is a decision tree ensemble algorithm similar to Random Forest, the difference is in how the trees are built and combined. Random Forest uses a technique called bagging to build full decision trees in parallel from bootstrap samples of the data set. The final prediction is an average of all of the decision tree predictions. Gradient Boosting Decision Trees use a technique called boosting to iteratively train an ensemble of shallow decision trees, with each iteration using weights given to records in the previous sample, which did not predict correctly, to decrease the error of the succeeding tree. The final prediction is a weighted average of all of the decision tree predictions. Bagging minimizes the variance and overfitting, boosting minimizes the bias and underfitting.

XGBoost is a variation of GBDTs. With GBDTs, the decision trees are built sequentially. With XGBoost, trees are built in parallel, following a level-wise strategy, scanning across gradient values and using these partial sums to evaluate the quality of splits at every possible split in the training set.

GPU-Accelerated XGBoost

The [GPU-accelerated XGBoost](#) algorithm makes use of fast parallel prefix sum operations to scan through all possible splits, as well as parallel radix sorting to repartition data. It builds a decision tree for a given boosting iteration, one level at a time, processing the entire Dataset concurrently on the GPU.

GPU-accelerated Spark XGBoost offers the following [key features](#):

- ▶ **Partitioning of ORC, CSV, and Parquet input files across multi GPUs**
Essentially any number/size of supported input file formats can be divided up evenly among the different training nodes.
- ▶ **GPU-accelerated training**
Improved XGBoost training time with a dynamic in-memory representation of the training data that optimally stores features based on the sparsity of a dataset rather than a fixed in-memory representation based on the largest number of features amongst different training instances. Decision trees are built using gradient pairs that can be reused to save memory, reducing copies to increase performance.
- ▶ **Efficient GPU memory utilization**
XGBoost requires data to fit into memory which creates a restriction on data size using either a single GPU or distributed multi-GPU multi-node training. Now, with improved GPU memory utilization, users can train with five times the size of data as compared to the first version. This is one of the critical factors to improve total cost of training without impacting performance.
- ▶ **Histogram based tree construction algorithms on GPUs**
The construction of decision trees can be slow since finding the exact best split for a feature requires going through all feature values and evaluating the loss function for each of them. For large datasets, it is unnecessary and repetitive to check every possible position to find the exact split location; instead, an approximately best split works quite well. One way to find the approximate best split is to test only k split positions, and this can be done efficiently using feature histograms. Finding optimal splits for a decision tree then reduces to the simpler problem of searching over histogram bins in a discrete space. The end result of this is a significantly faster and more memory efficient algorithm that still retains its accuracy. Because building histograms is a rather straight-forward process, it is easy to implement efficiently on GPU hardware.

Example Use Case Dataset

The example dataset is a [New York Taxi Dataset](#), which has already been cleaned up and transformed to add features, such as the [haversine distance](#) using this [Spark ETL notebook](#).

In this scenario, we'll build a model to predict the taxi fare amount, based on the following features:

- ▶ Label → fare amount
- ▶ Features → {passenger count, trip distance, pickup longitude, pickup latitude, rate code, dropoff longitude, dropoff latitude, hour, day of week, is weekend}

Load the Data from a File into a DataFrame

First, we import the packages needed for both GPU version and CPU versions of Spark xgboost:

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.types._
import org.apache.spark.sql._
import org.apache.spark.ml._
import org.apache.spark.ml.feature._
import org.apache.spark.ml.evaluation._
import org.apache.spark.sql.types._
import ml.dmlc.xgboost4j.scala.spark.{XGBoostRegressor,
XGBoostRegressionModel}
```

For the GPU version of Spark xgboost you also need the following import:

```
import ml.dmlc.xgboost4j.scala.spark.rapids.{GpuDataReader, GpuDataset}
```

We specify the schema with a Spark [StructType](#).

```
lazy val schema =
  StructType(Array(
    StructField("vendor_id", DoubleType),
    StructField("passenger_count", DoubleType),
    StructField("trip_distance", DoubleType),
    StructField("pickup_longitude", DoubleType),
    StructField("pickup_latitude", DoubleType),
    StructField("rate_code", DoubleType),
    StructField("store_and_fwd", DoubleType),
    StructField("dropoff_longitude", DoubleType),
    StructField("dropoff_latitude", DoubleType),
    StructField(labelName, DoubleType),
    StructField("hour", DoubleType),
    StructField("year", IntegerType),
    StructField("month", IntegerType),
    StructField("day", DoubleType),
    StructField("day_of_week", DoubleType),
    StructField("is_weekend", DoubleType)
  ))
```

In the following code we create a Spark session and set the training and evaluation data file paths. (Note: If you are using a notebook, then you do not have to create the SparkSession.)

```
val trainPath = "/FileStore/tables/taxi_tsmall.csv"
val evalPath  = "/FileStore/tables/taxi_esmall.csv"
val spark = SparkSession.builder().appName("Taxi-GPU").getOrCreate
```

We load the data from a CSV file into a Spark DataFrame, specifying the datasource and schema to load into a DataFrame, as shown below.

Load Data → DataFrame

```
val tdf = spark.read.option("inferSchema",
  "false").option("header", true).schema(schema).csv(trainPath)
val edf = spark.read.option("inferSchema", "false").option("header",
  true).schema(schema).csv(evalPath)
```

DataFrame show(5) displays the first 5 rows:

```
tdf.select("trip_distance", "rate_code", "fare_amount").show(5)
```

result:

trip_distance	rate_code	fare_amount
2.72	-6.77418915E8	11.5
0.94	-6.77418915E8	5.5
3.63	-6.77418915E8	13.0
11.86	-6.77418915E8	33.5
3.03	-6.77418915E8	11.0

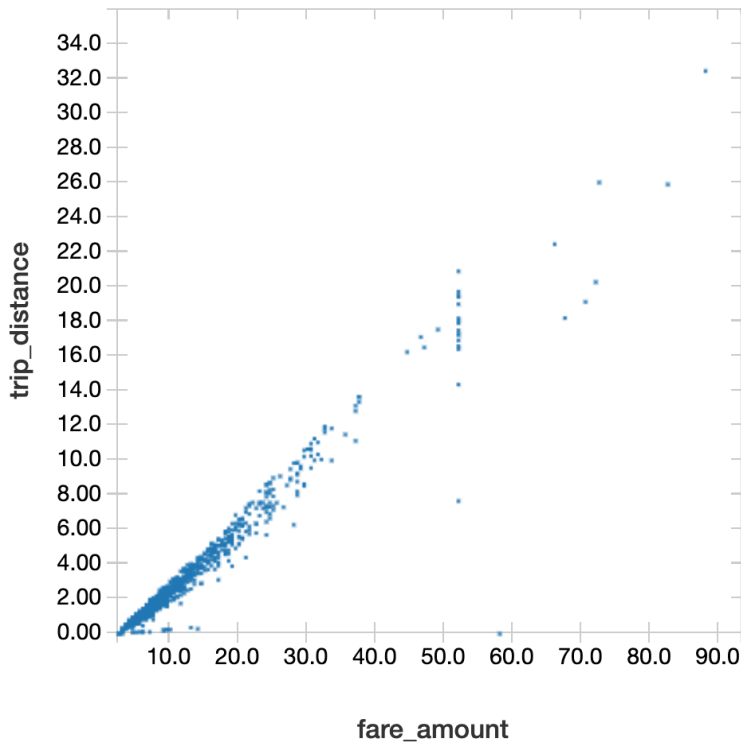
The function describe returns a DataFrame containing descriptive summary statistics, such as count, mean, standard deviation, and minimum and maximum value for each numerical column.

```
tdf.select("trip_distance", "rate_code", "fare_amount").describe().show
```

summary	trip_distance	rate_code	fare_amount
count	7999	7999	7999
mean	3.278923615451919	-6.569284350812602E8	12.348543567945994
stddev	3.6320775770793547	1.6677419425906155E8	10.221929466939088
min	0.0	-6.77418915E8	2.5
max	35.970000000000006	1.957796822E9	107.5

The following scatter plot is used to explore the correlation between the fare amount and the trip distance.

```
%sql
select trip_distance, fare_amount
from taxi
```



Define Features Array

For the features to be used by an ML algorithm, they are transformed and put into feature vectors, which are vectors of numbers representing the value for each feature. Below, a `VectorAssembler` transformer is used to return a new `DataFrame` with a label and a vector features column.



```
// feature column names
val featureNames = Array("passenger_count", "trip_distance",
    "pickup_longitude", "pickup_latitude", "rate_code", "dropoff_longitude",
    "dropoff_latitude", "hour", "day_of_week", "is_weekend")
// create transformer
object Vectorize {
```

```
def apply(df: DataFrame, featureNames: Seq[String], labelName: String):
DataFrame = {
  val toFloat = df.schema.map(f => col(f.name).cast(FloatType))
  new VectorAssembler()
    .setInputCols(featureNames.toArray)
    .setOutputCol("features")
    .transform(df.select(toFloat:_*))
    .select(col("features"), col(labelName))
}
}
// transform method adds features column
var trainSet = Vectorize(tdf, featureNames, labelName)
var evalSet = Vectorize(edf, featureNames, labelName)
trainSet.take(1)
result:
res8: Array[org.apache.spark.sql.Row] = Array([[5.0,2.7200000286102295,-
73.94813537597656,40.82982635498047,-6.77418944E8,-
73.96965026855469,40.79747009277344,10.0,6.0,1.0],11.5])
```

When using the XGBoost GPU version, the VectorAssembler is not needed.

For the CPU version the `num_workers` should be set to the number of CPU cores, the `tree_method` to "hist," and the features column to the output features column in the Vector Assembler.



```
lazy val paramMap = Map(
  "learning_rate" -> 0.05,
  "max_depth" -> 8,
  "subsample" -> 0.8,
  "gamma" -> 1,
  "num_round" -> 500
)
// set up xgboost parameters
val xgbParamFinal = paramMap ++ Map("tree_method" -> "hist", "num_workers" -
> 12)
// create the xgboostregressor estimator
val xgbRegressor = new XGBoostRegressor(xgbParamFinal)
  .setLabelCol(labelName)
  .setFeaturesCol("features")
```

For the GPU version the `num_workers` should be set to the number of machines with GPU in the Spark cluster, the `tree_method` to `"gpu_hist,"` and the features column to an array of strings containing the feature names.

```
val xgbParamFinal = paramMap ++ Map("tree_method" -> "gpu_hist",
  "num_workers" -> 1)
// create the estimator
val xgbRegressor = new XGBoostRegressor(xgbParamFinal)
  .setLabelCol(labelName)
  .setFeaturesCols(featureNames)
```

The following code uses the `XGBoostRegressor` estimator fit method on the training dataset to train and return an `XGBoostRegressor` model. We also use a time method to return the time to train the model and we use this to compare the time training with CPU vs. GPU.

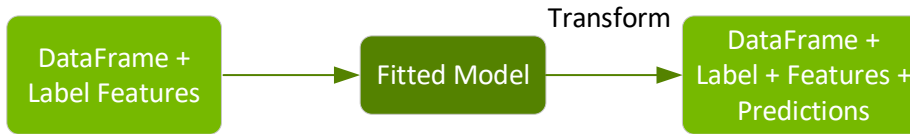


```
object Benchmark {
  def time[R](phase: String)(block: => R): (R, Float) = {
    val t0 = System.currentTimeMillis
    val result = block // call-by-name
    val t1 = System.currentTimeMillis
    println("Elapsed time [" + phase + "]: " +
      ((t1 - t0).toFloat / 1000) + "s")
    (result, (t1 - t0).toFloat / 1000)
  }
}

// use the estimator to fit (train) a model
val (model, _) = Benchmark.time("train") {
  xgbRegressor.fit(trainSet)
}
```

The performance of the model can be evaluated using the eval dataset which has not been used for training. We get predictions on the test data using the model transform method.

The model will estimate with the trained XGBoost model, and then return the fare amount predictions in a new predictions column of the returned DataFrame. Here again, we use the Benchmark time method in order to compare prediction times.



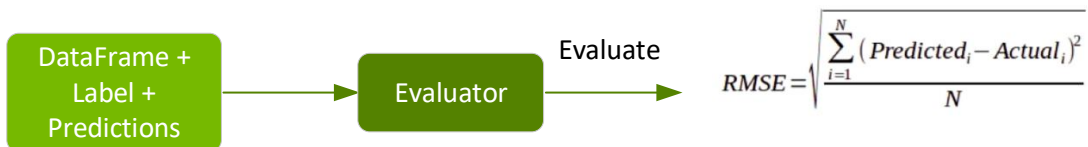
```
val (prediction, _) = Benchmark.time("transform") {
  val ret = model.transform(evalSet).cache()
  ret.foreachPartition(_ => ())
  ret
}
```

```
prediction.select( labelName, "prediction").show(10)
```

Result:

```
+-----+-----+
|fare_amount|      prediction|
+-----+-----+
|         5.0| 4.749197959899902|
|        34.0|38.651187896728516|
|        10.0|11.101678848266602|
|        16.5| 17.23284912109375|
|         7.0| 8.149757385253906|
|         7.5|7.5153608322143555|
|         5.5| 7.248467922210693|
|         2.5|12.289423942565918|
|         9.5|10.893491744995117|
|        12.0| 12.06682014465332|
+-----+-----+
```

The [RegressionEvaluator](#) evaluate method calculates the root mean square error, which is the square root of the mean squared error, from the prediction and label columns.



```
val evaluator = new RegressionEvaluator().setLabelCol(labelName)
val (rmse, _) = Benchmark.time("evaluation") {
  evaluator.evaluate(prediction)
}
```

```
println(s"RMSE == $rmse")
```

Result:

```
Elapsed time [evaluation]: 0.356s
```

```
RMSE == 2.6105287283128353
```

Save the Model

The model can be saved to disk, as shown below, in order to use later.

```
model.write.overwrite().save(savepath)
```

The result of saving the model is a JSON file for metadata and a Parquet file for model data. We can reload the model with the load command. The original and reloaded models are the same.

```
val sameModel = XGBoostRegressionModel.load(savepath)
```

Summary

In this chapter, we covered the basics of how XGBoost works and how to use XGBoost Regression with Spark to predict taxi fare amounts. You can now run this example on CPUs and GPUs with a larger dataset to compare time and accuracy of predictions.

Chapter 7: Real-world Examples of Accelerating End-to-End Machine Learning Pipelines

Machine learning on customer data can help marketers avoid bombarding customers with mass generic messaging or ads. Mass messaging is easy, but leveraging machine learning to surpass the one-size-fits-all approach with more targeted methods is more effective. As more and more companies are taking a personalized approach to content and marketing, an important step toward providing personalized recommendations is to predict the probability that a customer will perform a certain action. Machine learning can analyze large datasets of customer's preferences and past behaviors to predict this propensity, to identify and classify individuals who will benefit the most, similar to the way recommendation engines work, in order to maximize the impact with personalized and relevant email campaigns or ads.

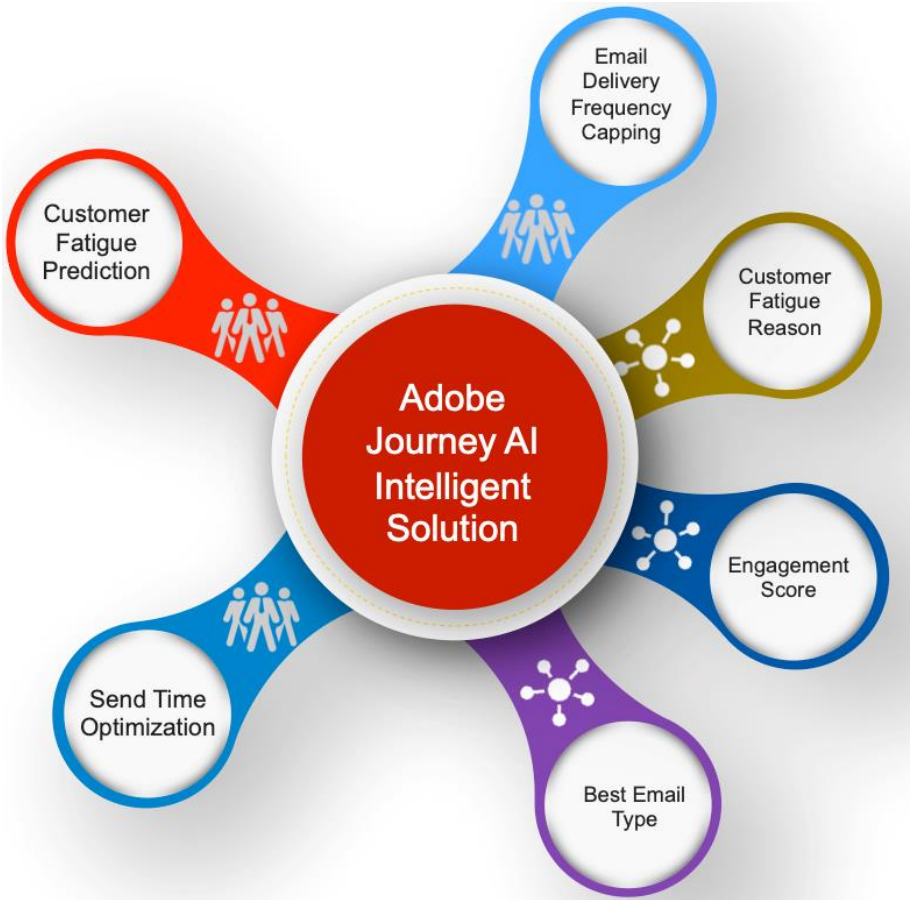
In this chapter, we will discuss how Adobe, AWS, and Uber are building accelerated end-to-end pipelines using Spark and GPU computing to enhance their customer services with ML or DL.

Adobe: Accelerated End-to-End Customer AI

In Adobe Intelligent Services, Journey AI optimizes the delivery of marketing messages and helps brands orchestrate the right message, to the right channel, at the right time. Journey AI uses advanced machine learning models to optimize customers' experience, maintain customers' brand awareness, and ensure that customers are not overwhelmed, by providing predictive insights in the following areas:

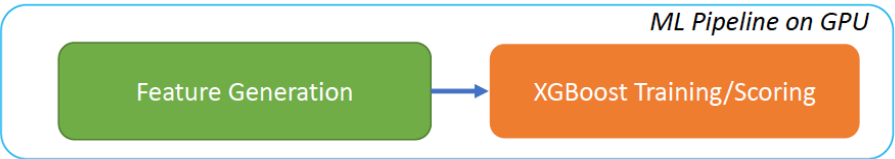
- ▶ **Send Time Optimization:** Predict best email time so that a customer is most likely to engage
- ▶ **Customer Fatigue prediction:** Predict probability of user engaging with email within the next 7 days after sending email.
- ▶ **Best email type:** recommend the best email type to promote engagement and conversion.
- ▶ **Engagement score:** Predict the probability of user unsubscribing after opening email
- ▶ **Customer Fatigue reason:** Why the fatigue ml model thinks the user's un-subscription risk is high.

- **Email Delivery Frequency Capping:** Recommend max number of emails that should be sent to customer in the next x number of days to maintain low fatigue risk level.



GPU-Accelerated, End-to-End ML

Adobe's AI Services team and NVIDIA collaborated on a high performance GPU accelerated, end-to-end ML pipeline consisting of Spark 3.x, the RAPIDS Accelerator for Spark, XGBoost, and RAPIDS software running on Databricks Azure.



For CPU vs. GPU cluster cost and runtime evaluation, the team used the following type of worker nodes on Databricks Azure:

	Node Name	CPU Cores	Memory	GPU	1 Year Reserved Cost
CPU Worker	Standard_L4	4	32 GB		\$0.853
GPU Worker	Standard NC6s_v3	6	112 GB	1 (TESLA V100)	\$4.7

Benchmark results of a Spark SQL and XGBoost pipeline, in the table below, showed significant performance improvement and cost reduction. Results showed time savings between 24-62% and cost savings between 16-58%, with larger datasets on GPUs benefiting the most.

Data Size	Customer	Cluster Type	Cluster Size	Run Time	Cost	Speed-up Ratio	Time Saving %	Cost Saving %
Small	Customer 1	GPU Worker	1	264	0.34466667	1.32575758	24.57	16.88
		CPU Workers	5	350	0.41465278			
	NVIDIA	GPU Worker	1	539	0.70369444	1.57884972	36.66	30.20
		CPU Workers	5	851	1.00819861			
Medium	Customer 2	GPU Workers	2	323	0.84338889	2.11764706	52.78	47.96
		CPU Workers	10	684	1.6207			
	Customer 3	GPU Workers	2	433	1.13061111	2.37413395	57.88	53.58
		CPU Workers	10	1028	2.43578889			
	Customer 4	GPU Workers	2	1170	3.055	1.44444444	30.77	23.71
		CPU Workers	10	1690	4.00436111			
Large	Customer 5	GPU Workers	5	1621	10.5815278	2.6668723	62.50	56.68
		CPU Workers	25	4323	25.6077708			
	Customer 6	GPU Workers	10	3010	39.2972222	1.7551495	43.02	37.21
		CPU Workers	50	5283	62.588875			

Another benchmark on the team's largest real customer production Spark SQL and XGBoost pipeline, with 2.88 TB data and complicated joins and aggregations, resulted in time savings of 54.6% and cost savings of 50%

Customer	Cluster Type	Cluster Size	Run Time (s)	Cost (\$)	Speed-up Ratio	Time Saving %	Cost Saving%
***** (Adobe)	GPU Workers	20	2692	70.29	2.20653789	54.68	50.06
	CPU Workers	100	5940	140.75			

To learn more listen to [GPU-Accelerated High-Performance Machine Learning Pipeline](#).

AWS: Accelerating Deep Learning on the JVM with Apache Spark and NVIDIA GPUs

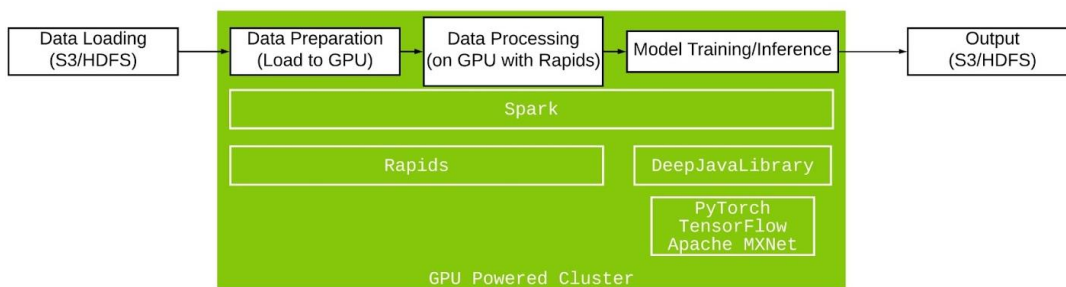
Many AWS customers are interested in adopting deep learning with business use cases ranging from customer service (including object detection from images and video streams, sentiment analysis) to fraud detection and collaboration. However, until recently, there were multiple difficulties with implementing deep learning in enterprise applications:

- ▶ The adoption learning curve was steep and required development of internal technical expertise in new programming languages (e.g., Python) and frameworks.
- ▶ Deep Learning training and Inference is compute intensive and typically performed on GPUs, while large-scale data engineering was typically programmed in Scala on multi-CPU distributed Apache Spark.

Developed by Amazon, Deep Java Library ([DJL](#)) is an open-source Deep Learning Framework implemented in Java on top of modern Deep Learning engines (TensorFlow, PyTorch, MXNet, etc). While Java remains the first or second most popular programming language since the late 90s, Python is the most used language for machine learning, with numerous resources and deep-learning frameworks. DJL aims to make deep-learning open source tools accessible to developers who primarily use Java or Scala with familiar concepts and intuitive APIs.

By combining Spark 3.x, the Rapids Accelerator for Spark and DJL, users can build end-to-end GPU accelerated Scala-based big data + DL pipelines using Apache Spark.

This combination of Deep Java Learning, Apache Spark 3.x, and NVIDIA GPU computing simplifies deep learning pipelines while improving performance and reducing costs.



End-to-End Spark and DJL on AWS EMR

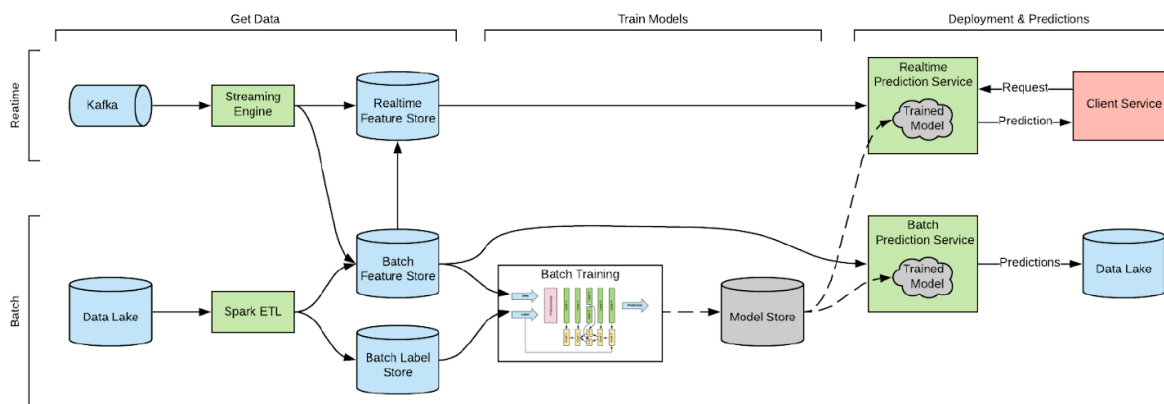
At Amazon, the retail systems team created a multi-label classification model to understand customer action propensity across thousands of product categories and used these propensities to create a personalized experience for customers. To achieve this goal at Amazon scale, the team built a Scala-based big data pipeline using Apache Spark 3.x, DJL, and NVIDIA GPU computing on Amazon EMR.

To learn more read [Accelerating Deep Learning on the JVM with Apache Spark and NVIDIA GPUs](#).

Uber: Accelerated End-to-End ETL and DL

Uber applies Deep Learning across their business, from self-driving research to trip forecasting and fraud prevention. Uber developed Horovod, a distributed DL training framework for TensorFlow, Keras, PyTorch, and Apache MXNet, to make it easier to speed up DL projects with GPUs and a data parallel approach to distributed training.

Horovod has support for Spark 3.x with GPU scheduling, and a new KerasEstimator class that uses Spark Estimators with Spark ML Pipelines for better integration with Spark and ease of use. This enables TensorFlow and PyTorch models to be trained directly on Spark DataFrames, leveraging Horovod's ability to scale to hundreds of GPUs in parallel, without any specialized code for distributed training. With the new accelerator-aware scheduling and columnar processing APIs in Apache Spark 3.x, a production ETL job can hand off data to Horovod running distributed DL training on GPUs within the same pipeline.



End-to-End Spark ETL and Deep Learning at Uber

To learn more listen to [Distributed deep learning with Horovod](#).

Summary

In this chapter, we discussed how Adobe, AWS, and Uber are building accelerated end-to-end ML pipelines using Spark and GPU computing to enhance their customer services.

Appendix: Code

Code

You can download the code to run the examples in the book from here:

- ▶ <https://github.com/caroljmcDonald/spark3-book>
- ▶ <https://github.com/rapidsai/spark-examples>

Additional Resources

- ▶ Spark documentation including deployment and configuration:
<https://spark.apache.org/docs/latest/>
- ▶ RAPIDS Accelerator for Spark
<https://nvidia.github.io/spark-rapids/>
- ▶ RAPIDS AI home <https://rapids.ai/>
- ▶ NVIDIA Developer portal <https://developer.nvidia.com/>

About the Author

Carol has experience in many roles including software architecture/development, training, technology evangelism, and developer outreach. As a Solutions Architect and Instructor at MapR, Carol focused on big data, Apache Spark, Apache HBase, Apache Drill, and machine learning in healthcare, finance, and telecom. As a Java Technology Evangelist at Sun, Carol traveled worldwide, speaking, and giving Hands-on-Labs at Sun Tech Days. As a software developer and architect, Carol developed complex mission-critical applications in the banking, health insurance and telecom industries including: a large health information exchange connecting over one million providers and health plans, a large loan servicing application for Wells Fargo, Drug Development Intranet applications for Hoffman La Roche, Telecom Network Management applications for HP, Open Systems Interconnection messaging applications for IBM, and Sigint applications for the NSA. Carol holds an MS in computer science from the University of Tennessee and a BS in geology from Vanderbilt University and is an O'Reilly Certified Spark Developer, MapR certified Spark and HBase developer, and Sun Certified Java Architect and Java Programmer. Carol is fluent in English, French, and German.

NVIDIA Contributors

- ▶ Jim Scott
- ▶ Karthikeyan Rajendran
- ▶ Andrew Feng
- ▶ Thomas Graves
- ▶ Alessandro Bellina
- ▶ Sameer Raheja

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Trademarks

NVIDIA, the NVIDIA logo, CUDA, and NVLink, are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

■ Apache Spark® is the registered trademark or trademark of the Apache Software Foundation in the United States and/or other countries.

Copyright

© 2021 NVIDIA Corporation. All rights reserved.