

EBOOK

Getting Started with **Apache Spark** from Inception to Production

Carol McDonald

with contribution from Ian Downard

EBOOK

Getting Started with
Apache Spark
from Inception to Production

Carol McDonald
with contribution from Ian Downard

Getting Started with Apache Spark
From Inception to Production

By Carol McDonald

Copyright © 2018

Carol McDonald, Ian Downard, and MapR Technologies, Inc. All rights reserved.

Printed in the United States of America

Published by MapR Technologies, Inc.

4555 Great America Parkway, Suite 201

Santa Clara, CA 95054

October 2018: Second Edition

Revision History for the First Edition

2015-09-01: First release

Apache, Apache Spark, Apache Hadoop, Spark, and Hadoop are trademarks of The Apache Software Foundation. Used with permission. No endorsement by The Apache Software Foundation is implied by the use of these marks. While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions or for damages resulting from the use of the information contained herein.

Table of Contents

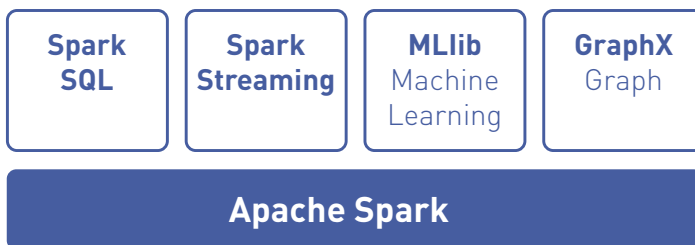
Chapter 1	Spark 101: What It Is, What It Does, and Why It Matters	5
Chapter 2	Datasets, DataFrames, and Spark SQL	13
Chapter 3	How Spark Runs Your Applications	30
Chapter 4	Demystifying AI, Machine Learning, and Deep Learning	47
Chapter 5	Predicting Flight Delays Using Apache Spark Machine Learning	71
Chapter 6	Cluster Analysis on Uber Event Data to Detect and Visualize Popular Uber Locations	94
Chapter 7	Real-Time Analysis of Popular Uber Locations Using Apache APIs: Spark Structured Streaming, Machine Learning, Kafka, and MapR-DB	110
Chapter 8	Predicting Forest Fire Locations with <i>K</i> -Means in Spark	135
Chapter 9	Using Apache Spark GraphFrames to Analyze Flight Delays and Distances	144
Chapter 10	Tips and Best Practices to Take Advantage of Spark 2.x	172
	Appendix	192

Spark 101: What It Is, What It Does, and Why It Matters

In this chapter, we introduce Apache Spark and explore some of the areas in which its particular set of capabilities show the most promise. We discuss the relationship to other key technologies and provide some helpful pointers, so that you can hit the ground running and confidently try Spark for yourself.

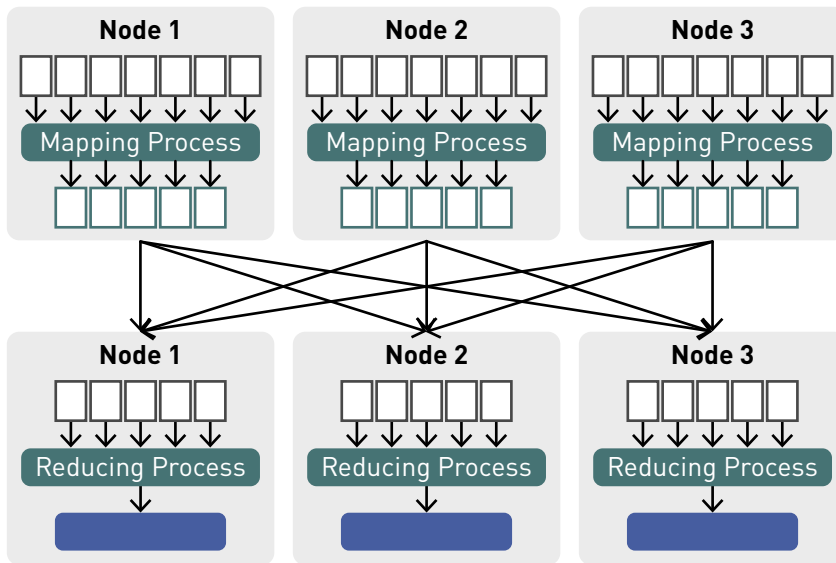
What Is Apache Spark?

Spark is a general-purpose distributed data processing engine that is suitable for use in a wide range of circumstances. On top of the Spark core data processing engine, there are libraries for SQL, machine learning, graph computation, and stream processing, which can be used together in an application. Programming languages supported by Spark include: Java, Python, Scala, and R. Application developers and data scientists incorporate Spark into their applications to rapidly query, analyze, and transform data at scale. Tasks most frequently associated with Spark include ETL and SQL batch jobs across large data sets, processing of streaming data from sensors, IoT, or financial systems, and machine learning tasks.



History

In order to understand Spark, it helps to understand its history. Before Spark, there was MapReduce, a resilient distributed processing framework, which enabled Google to index the exploding volume of content on the web, across large clusters of commodity servers.

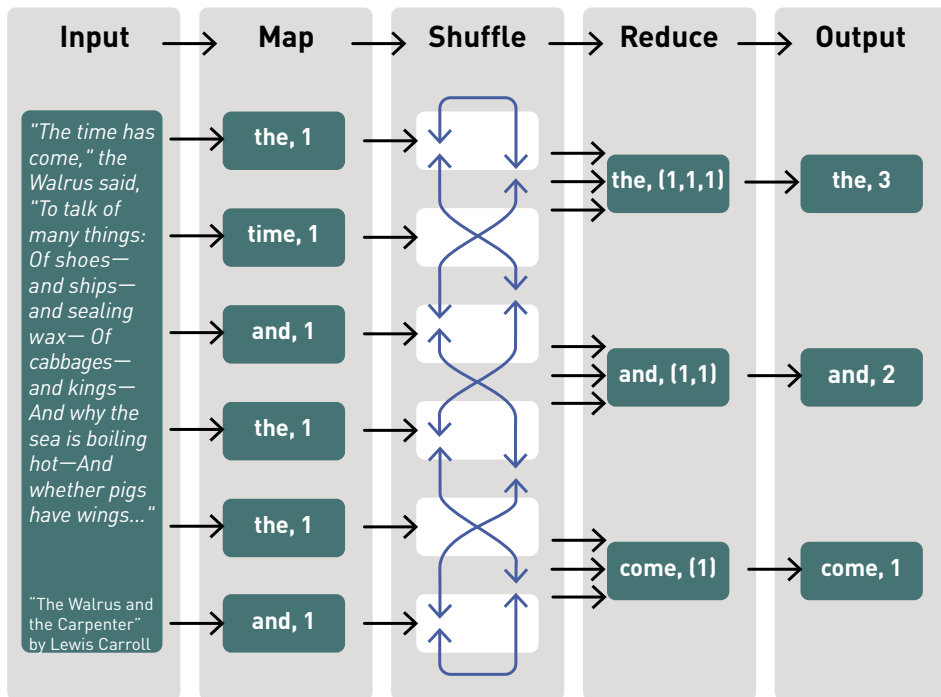


There were 3 core concepts to the Google strategy:

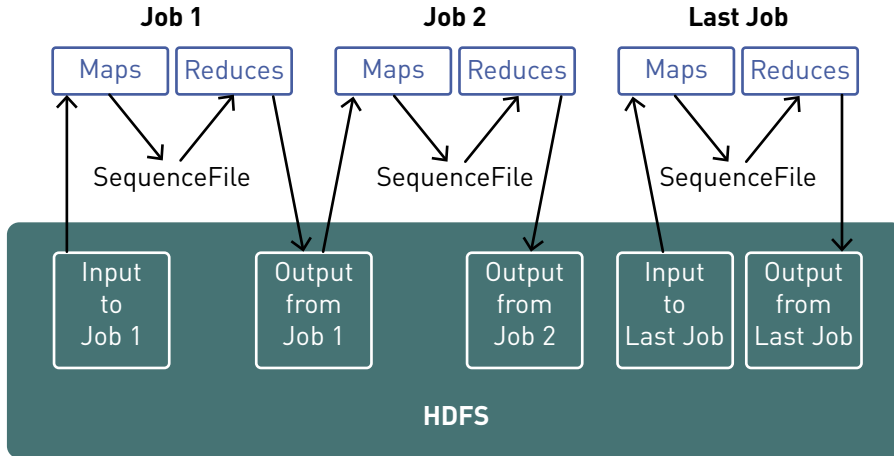
1. **Distribute data:** when a data file is uploaded into the cluster, it is split into chunks, called data blocks, and distributed amongst the data nodes and replicated across the cluster.
2. **Distribute computation:** users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs and a *reduce* function that merges all intermediate values associated with the same intermediate key. Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines in the following way:
 - The mapping process runs on each assigned data node, working only on its block of data from a distributed file.
 - The results from the mapping processes are sent to the reducers in a process called “shuffle and sort”: key/value pairs from the mappers are sorted by key, partitioned by the number of reducers, and then sent across the network and written to key sorted “sequence files” on the reducer nodes.

- The reducer process executes on its assigned node and works only on its subset of the data (its sequence file). The output from the reducer process are written to an output file.
- 3. Tolerate faults:** both data and computation can tolerate failures by failing over to another node for data or processing.

MapReduce word count execution example:



Some iterative algorithms, like PageRank, which Google used to rank websites in their search engine results, require chaining multiple MapReduce jobs together, which causes a lot of reading and writing to disk. When multiple MapReduce jobs are chained together, for each MapReduce job, data is read from a distributed file block into a map process, written to and read from a SequenceFile in between, and then written to an output file from a reducer process.



A year after Google published a [white paper describing the MapReduce](#) framework (2004), Doug Cutting and Mike Cafarella created [Apache Hadoop™](#)

Apache Spark™ began life in 2009 as a project within the AMPLab at the University of California, Berkeley. Spark became an incubated project of the Apache Software Foundation in 2013, and it was promoted early in 2014 to become one of the Foundation's top-level projects. Spark is currently one of the most active projects managed by the Foundation, and the community that has grown up around the project includes both prolific individual contributors and well-funded corporate backers, such as Databricks, IBM, and China's Huawei.

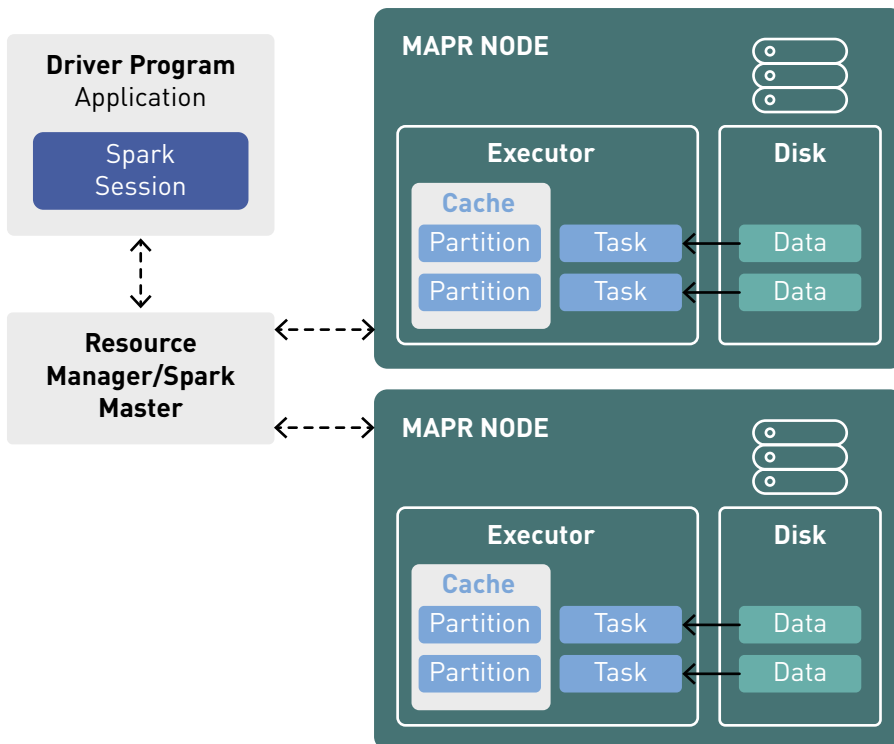
The goal of the Spark project was to keep the benefits of MapReduce's scalable, distributed, fault-tolerant processing framework, while making it more efficient and easier to use. The advantages of Spark over MapReduce are:

- Spark executes much faster by caching data in memory across multiple parallel operations, whereas MapReduce involves more reading and writing from disk.
- Spark runs multi-threaded tasks inside of JVM processes, whereas MapReduce runs as heavier weight JVM processes. This gives Spark faster startup, better parallelism, and better CPU utilization.
- Spark provides a richer functional programming model than MapReduce.
- Spark is especially useful for parallel processing of distributed data with **iterative** algorithms.

How a Spark Application Runs on a Cluster

The diagram below shows a Spark application running on a cluster.

- A Spark application runs as independent processes, coordinated by the SparkSession object in the driver program.
- The resource or cluster manager assigns tasks to workers, one task per partition.
- A task applies its unit of work to the dataset in its partition and outputs a new partition dataset. Because iterative algorithms apply operations repeatedly to data, they benefit from caching datasets across iterations.
- Results are sent back to the driver application or can be saved to disk.



Spark supports the following resource/cluster managers:

Spark Standalone – a simple cluster manager included with Spark

Apache Mesos – a general cluster manager that can also run Hadoop applications

Apache Hadoop YARN – the resource manager in Hadoop 2

Kubernetes – an open source system for automating deployment, scaling, and management of containerized applications

Spark also has a local mode, where the driver and executors run as threads on your computer instead of a cluster, which is useful for developing your applications from a personal computer.

What Does Spark Do?

Spark is capable of handling several petabytes of data at a time, distributed across a cluster of thousands of cooperating physical or virtual servers. It has an extensive set of developer libraries and APIs and supports languages such as Java, Python, R, and Scala; its flexibility makes it well-suited for a range of use cases. Spark is often used with distributed data stores such as MapR-XD, Hadoop's HDFS, and Amazon's S3, with popular NoSQL databases such as MapR-DB, Apache HBase, Apache Cassandra, and MongoDB, and with distributed messaging stores such as MapR-ES and Apache Kafka.

Typical use cases include:

Stream processing: From log files to sensor data, application developers are increasingly having to cope with “streams” of data. This data arrives in a steady stream, often from multiple sources simultaneously. While it is certainly feasible to store these data streams on disk and analyze them retrospectively, it can sometimes be sensible or important to process and act upon the data as it arrives. Streams of data related to financial transactions, for example, can be processed in real time to identify – and refuse – potentially fraudulent transactions.

Machine learning: As data volumes grow, machine learning approaches become more feasible and increasingly accurate. Software can be trained to identify and act upon triggers within well-understood data sets before applying the same solutions to new and unknown data. Spark's ability to store data in memory and rapidly run repeated queries makes it a good choice for training machine learning algorithms. Running broadly similar queries again and again, at scale, significantly reduces the time required to go through a set of possible solutions in order to find the most efficient algorithms.

Interactive analytics: Rather than running pre-defined queries to create static dashboards of sales or production line productivity or stock prices, business analysts and data scientists want to explore their data by asking a question, viewing the result, and then either altering the initial question slightly or drilling deeper into results. This interactive query process requires systems such as Spark that are able to respond and adapt quickly.

Data integration: Data produced by different systems across a business is rarely clean or consistent enough to simply and easily be combined for reporting or analysis. Extract, transform, and load (ETL) processes are often used to pull data from different systems, clean and standardize it, and then load it into a separate system for analysis. Spark (and Hadoop) are increasingly being used to reduce the cost and time required for this ETL process.

Who Uses Spark?

A wide range of technology vendors have been quick to support Spark, recognizing the opportunity to extend their existing big data products into areas where Spark delivers real value, such as interactive querying and machine learning. Well-known companies such as IBM and Huawei have invested significant sums in the technology, and a growing number of startups are building businesses that depend in whole or in part upon Spark. For example, in 2013 the Berkeley team responsible for creating Spark founded Databricks, which provides a hosted end-to-end data platform powered by Spark. The company is well-funded, having received \$47 million across two rounds of investment in 2013 and 2014, and Databricks employees continue to play a prominent role in improving and extending the open source code of the Apache Spark project.

The major Hadoop vendors, including MapR, Cloudera, and Hortonworks, have all moved to support YARN-based Spark alongside their existing products, and each vendor is working to add value for its customers. Elsewhere, IBM, Huawei, and others have all made significant investments in Apache Spark, integrating it into their own products and contributing enhancements and extensions back to the Apache project. Web-based companies, like Chinese search engine Baidu, e-commerce operation Taobao, and social networking company Tencent, all run Spark-based operations at scale, with Tencent's 800 million active users reportedly generating over 700 TB of data per day for processing on a cluster of more than 8,000 compute nodes.

In addition to those web-based giants, pharmaceutical company Novartis depends upon Spark to reduce the time required to get modeling data into the hands of researchers, while ensuring that ethical and contractual safeguards are maintained.

What Sets Spark Apart?

There are many reasons to choose Spark, but three are key:

Simplicity: Spark's capabilities are accessible via a set of rich APIs, all designed specifically for interacting quickly and easily with data at scale. These APIs are well-documented and structured in a way that makes it straightforward for data scientists and application developers to quickly put Spark to work.

Speed: Spark is designed for speed, operating both in memory and on disk. Using Spark, a team from Databricks [tied for first place](#) with a team from the University of California, San Diego, in the 2014 Daytona GraySort benchmarking challenge (<https://spark.apache.org/news/spark-wins-daytona-gray-sort-100tb-benchmark.html>). The challenge involves processing a static data set; the Databricks team was able to process 100 terabytes of data stored on solid-state drives in just 23 minutes, and the previous winner took 72 minutes by using Hadoop and a different cluster configuration. Spark can perform even better when supporting interactive queries of data stored in memory. In those situations, there are claims that Spark can be 100 times faster than Hadoop's MapReduce.

Support: Spark supports a range of programming languages, including Java, Python, R, and Scala. Spark includes support for tight integration with a number of leading storage solutions in the Hadoop ecosystem and beyond, including: MapR (file system, database, and event store), Apache Hadoop (HDFS), Apache HBase, and Apache Cassandra. Furthermore, the Apache Spark community is large, active, and international. A growing set of commercial providers, including Databricks, IBM, and all of the main Hadoop vendors, deliver comprehensive support for Spark-based solutions.

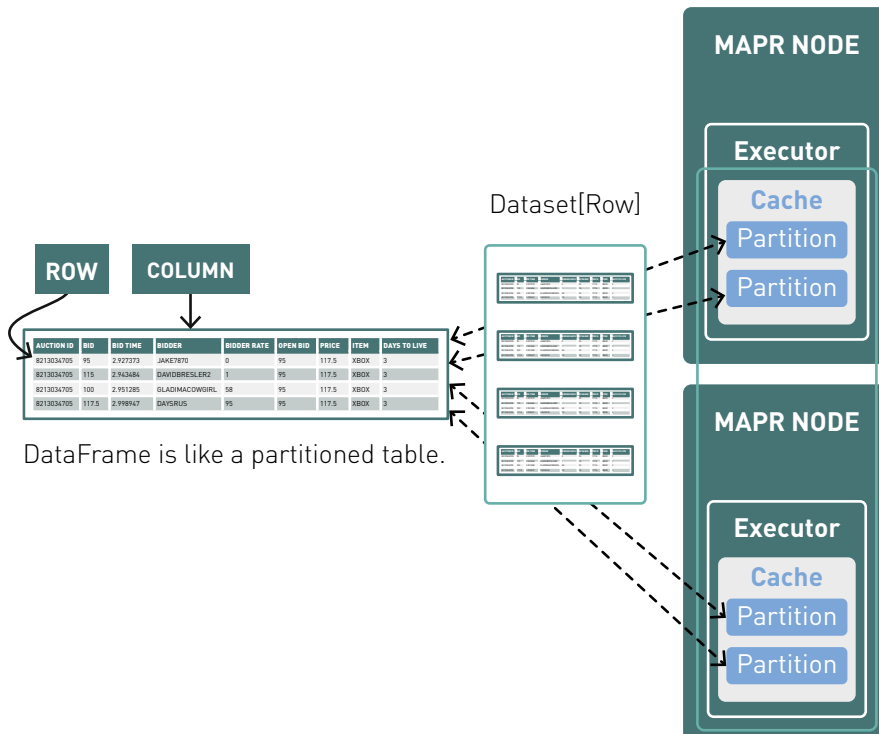
The Power of Data Pipelines

Much of Spark's power lies in its ability to combine very different techniques and processes together into a single, coherent whole. Outside Spark, the discrete tasks of selecting data, transforming that data in various ways, and analyzing the transformed results might easily require a series of separate processing frameworks, such as Apache Oozie. Spark, on the other hand, offers the ability to combine these together, crossing boundaries between batch, streaming, and interactive workflows in ways that make the user more productive.

Spark jobs perform multiple operations consecutively, in memory, and only spilling to disk when required by memory limitations. Spark simplifies the management of these disparate processes, offering an integrated whole – a data pipeline that is easier to configure, easier to run, and easier to maintain. In use cases such as ETL, these pipelines can become extremely rich and complex, combining large numbers of inputs and a wide range of processing steps into a unified whole that consistently delivers the desired result.

Datasets, DataFrames, and Spark SQL

A Spark Dataset is a distributed collection of typed objects, which are partitioned across multiple nodes in a cluster and can be operated on in parallel. Datasets can be created from MapR-XD files, MapR-DB tables, or MapR-ES topics, and can be cached, allowing reuse across parallel operations. A Dataset can be manipulated using functional transformations (map, flatMap, filter, etc.) and/or Spark SQL. A DataFrame is a Dataset of Row objects and represents a table of data with rows and columns. A DataFrame consists of partitions, each of which is a range of rows in cache on a data node.



The SparkSession Object

As discussed before, a Spark application runs as independent processes, coordinated by the SparkSession object in the driver program. The entry point to programming in Spark is the `org.apache.spark.sql.SparkSession` class, which you use to create a SparkSession object as shown below:

```
val spark = SparkSession.builder().appName("example").master(
  "local[*]").getOrCreate()
```

If you are using the spark-shell or a notebook, the SparkSession object is already created and available as the variable `spark`.

Interactive Analysis with the Spark Shell

The Spark shell provides an easy way to learn Spark interactively. You can start the shell with the following command:

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

You can enter the code from the rest of this chapter into the Spark shell; outputs from the shell are prefaced with **result**.

Exploring U.S. Flight Data with Spark Datasets and DataFrames

To go over some core concepts of Spark Datasets, we will be using some flight information from the [United States Department of Transportation](#). Later, we will use this same data to predict flight delays, so we want to explore the flight attributes that most contribute to flight delays. Using Spark Datasets, we will explore the data to answer questions, like: which airline carriers, days of the week, originating airport, and hours of the day have the highest number of flight delays, when a delay is greater than 40 minutes.

The flight data is in JSON files, with each flight having the following information:

- id: ID composed of carrier, date, origin, destination, flight number
- dofW: day of week (1=Monday, 7=Sunday)
- carrier: carrier code
- origin: origin airport code
- dest: destination airport code
- crsdephour: scheduled departure hour

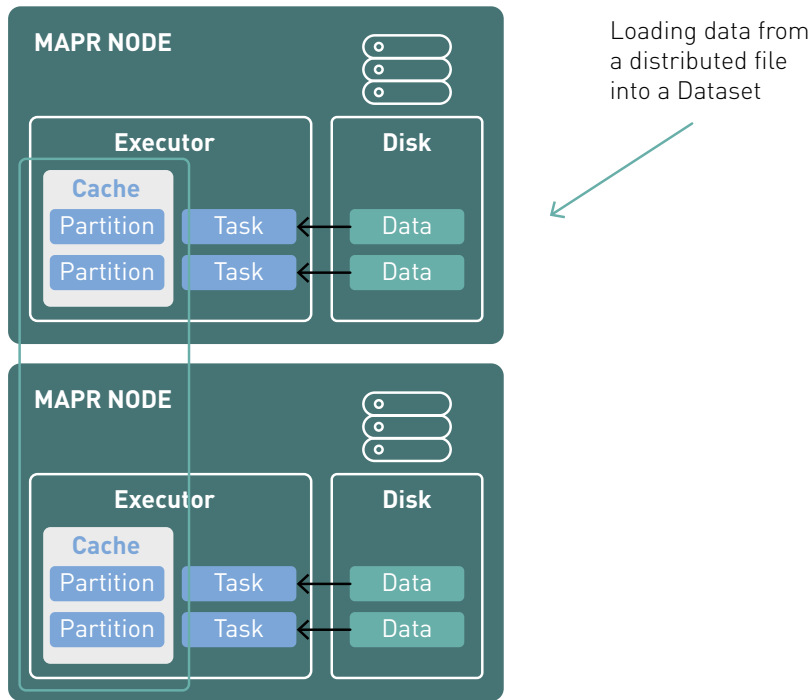
- crsdeptime: scheduled departure time
- depdelay: departure delay in minutes
- crsarrrtime: scheduled arrival time
- arrdelay: arrival delay minutes
- crselapsedtime: elapsed time
- dist: distance

It appears in the following format:

```
{
  "_id": "AA_2017-01-01_ATL_LGA_1678",
  "dofw": 7,
  "carrier": "AA",
  "origin": "ATL",
  "dest": "LGA",
  "crsdephour": 17,
  "crsdeptime": 1700,
  "depdelay": 0.0,
  "crsarrrtime": 1912,
  "arrdelay": 0.0,
  "crselapsedtime": 132.0,
  "dist": 762.0
}
```

(The complete data and code for all examples are available in the GitHub link in the appendix.)

Loading Data from a File into a Dataset



With the `SparkSession.read` method, we can read data from a file into a `DataFrame`, specifying the file type, file path, and input options for the schema. The schema can optionally be inferred from the contents of the JSON file, but you will get better performance and accuracy by specifying the schema.

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

val schema = StructType(Array(
  StructField("_id", StringType, true),
  StructField("dofW", IntegerType, true),
  StructField("carrier", StringType, true),
  StructField("origin", StringType, true),
  StructField("dest", StringType, true),
  StructField("crsdephour", IntegerType, true),
  StructField("crsdeptime", DoubleType, true),
  StructField("crsarrrtime", DoubleType, true),
  StructField("crselapsedtime", DoubleType, true),
  StructField("label", DoubleType, true),
  StructField("pred_dtrees", DoubleType, true)
))
var file = "maprfs:///data/flights.json"

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

result:
df: org.apache.spark.sql.DataFrame = [_id: string, dofW: int ...
10 more fields]
```

The `take` method returns an array with objects from this Dataset, which we see is of type `Row`.

```
df.take(1)

result:
Array[org.apache.spark.sql.Row] =
Array([ATL_LGA_2017-01-01_17_AA_1678, 7, AA, ATL, LGA, 17, 1700.0,
0.0, 1912.0, 0.0, 132.0, 762.0])
```

If we supply a case class with the `as` method when loading the data, then the data is read into a Dataset of typed objects corresponding to the case class.

```
case class Flight(_id: String, dofW: Integer, carrier: String,
origin: String, dest: String, crsdephour: Integer, crsdeptime:
Double, depdelay: Double, crsarrrtime: Double, arrdelay: Double,
crselapsedtime: Double, dist: Double) extends Serializable

val df = spark.read.format("json").option("inferSchema", "false").
schema(schema).load(file).as[Flight]

result:
df: org.apache.spark.sql.Dataset[Flight] = [_id: string, dofW: int
... 10 more fields]
```

Now the `take` method returns an array of Flight objects.

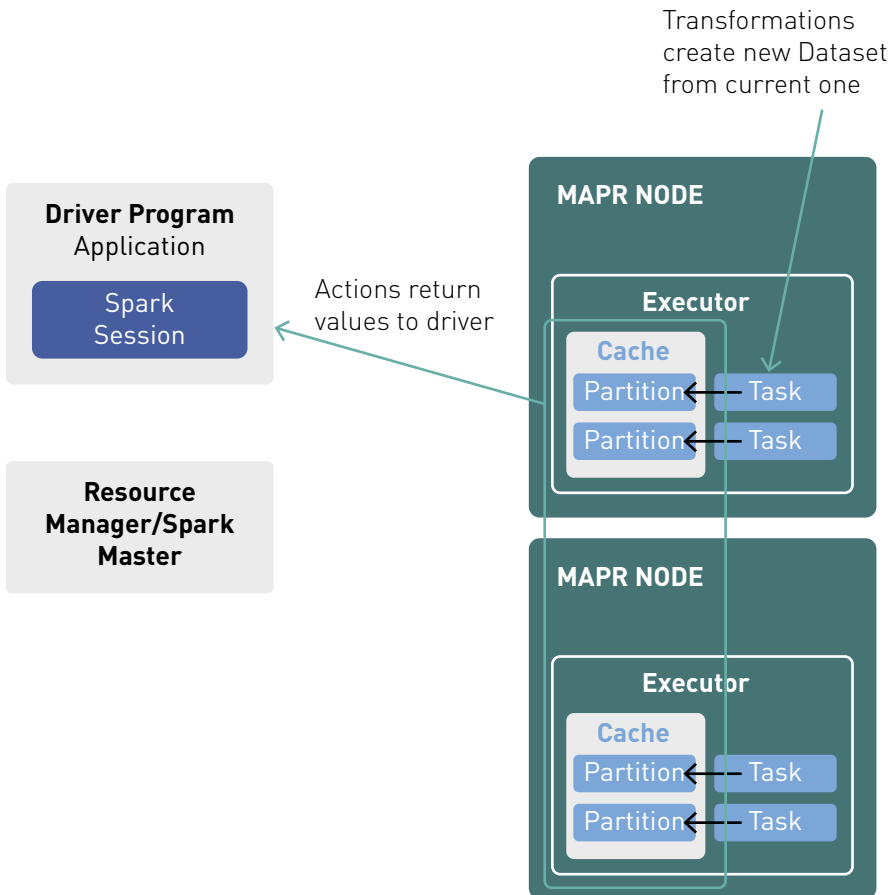
```
df.take(1)

result:
Array[Flight] = Array(Flight(ATL_LGA_2017-01-01_17_AA_1678,
7,AA,ATL,LGA,17,1700.0,0.0,1912.0,0.0,132.0,762.0))
```

Transformations and Actions

There are two types of operations you can perform on a Dataset:

- transformations: create a new Dataset from the current Dataset
- actions: trigger computation and return a result to the driver program



Transformations are lazily evaluated, which means they are not computed immediately. A transformation is executed only when it is triggered by an action. Once an action has run and the value is returned, the Dataset is no longer in memory, unless you call the cache method on the Dataset. If you will reuse a Dataset for more than one action, you should cache it.

Datasets and Type Safety

Datasets are composed of typed objects, which means that transformation syntax errors (like a typo in the method name) and analysis errors (like an incorrect input variable type) can be caught at compile time. DataFrames are composed of untyped Row objects, which means that only syntax errors can be caught at compile time. Spark SQL is composed of a string, which means that syntax errors and analysis errors are only caught at runtime. Datasets save a developer's time by catching errors sooner, even while typing when using an IDE.

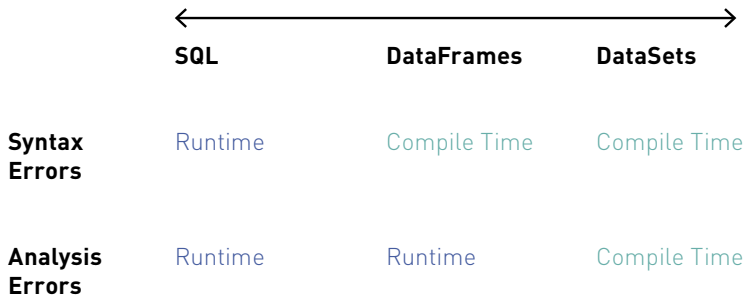


Image reference: Databricks

Dataset Transformations

Here is a list of some commonly used typed transformations, which can be used on Datasets of typed objects (Dataset[T]).

map	Returns new Dataset with result of applying input function to each element
filter	Returns new Dataset containing elements where input function is true
groupByKey	Returns a KeyValueGroupedDataset where the data is grouped by the given key function

This example filter transformation on the flight Dataset returns a Dataset with flights that departed at 10 AM. The take action returns an array of flight objects to the driver program.

```
df.filter(flight => flight.crsdephour == 10).take(3)
```

result:

```
Array[Flight] = Array(Flight(ORD_DEN_2017-01-01_AA_2300, 7, AA, ORD,
DEN, 10, 1005.0, 5.0, 1145.0, 3.0, 160.0, 888.0), Flight(MIA_ORD_2017-01-
01_AA_2439, 7, AA, MIA, ORD, 10, 1005.0, 4.0, 1231.0, 0.0, 206.0, 1197.0))
```

DataFrame Transformations

Here is a list of some commonly used untyped transformations, which can be used on Dataframes (Dataset[Row]).

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 flight Dataset by carrier, then the count action counts the number of flights for each carrier. The show action prints out the resulting DataFrame rows in tabular format.

```
df.groupBy("carrier").count().show()
```

result:

```
+-----+-----+
|carrier|count|
+-----+-----+
|      UA|18873|
|      AA|10031|
|      DL|10055|
|      WN| 2389|
+-----+-----+
```

Here is a list of some commonly used Dataset actions.

<code>show(n)</code>	Displays the first <code>n</code> rows in a tabular form
<code>take(n)</code>	Returns the first <code>n</code> objects in the Dataset in an array
<code>count</code>	Returns the number of rows in the Dataset

Here is an example using typed and untyped transformations and actions to get the destinations with the highest number of departure delays, where a delay is greater than 40 minutes. We count the departure delays greater than 40 minutes by destination and sort them with the highest first.

```
df.filter($"depdelay" > 40).groupBy("dest").count()
.orderBy(desc("count")).show(3)
```

result:

```
+----+-----+
|dest|count|
+----+-----+
| SFO|   711|
| EWR|   620|
| ORD|   593|
+----+-----+
```

Exploring the Flight Dataset with Spark SQL

Now let's explore the flight Dataset using Spark SQL and DataFrame transformations. After we register the DataFrame as a 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, since we will reuse it and because Spark can cache DataFrames or Tables in columnar format in memory, which can improve memory usage and performance.

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

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

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

Below we display information for the top five longest departure delays with Spark SQL and with DataFrame transformations (where a delay is considered greater than 40 minutes):

```
// Spark SQL
spark.sql("select carrier,origin, dest, depdelay,crsdephour, dist,
dofW from flights where depdelay > 40 order by depdelay desc limit
5").show

// same query using DataFrame transformations

df.select($"carrier",$"origin",$"dest",$"depdelay", $"crsdephour").
filter($"depdelay" > 40).orderBy(desc( "depdelay" )).show(5)
```

result:

carrier	origin	dest	depdelay	crsdephour
AA	SFO	ORD	1440.0	8
DL	BOS	ATL	1185.0	17
UA	DEN	EWR	1138.0	12
DL	ORD	ATL	1087.0	19
UA	MIA	EWR	1072.0	20

Below we display the average departure delay by carrier:

```
// DataFrame transformations

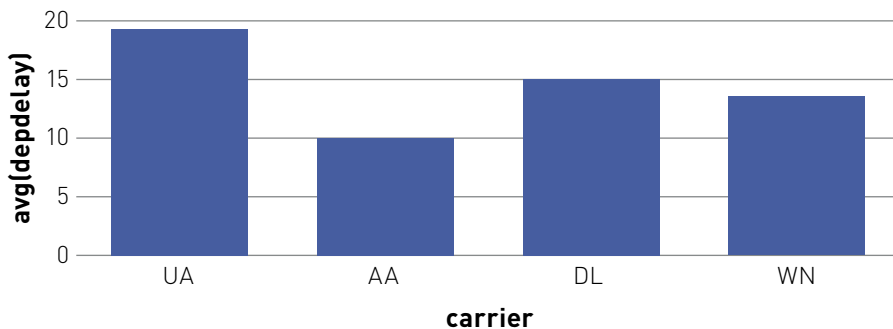
df.groupBy("carrier").agg(avg("depdelay")).show
```

result:

carrier	avg(depdelay)
UA	17.477878450696764
AA	10.45768118831622
DL	15.316061660865241
WN	13.491000418585182

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

```
// Spark SQL
%sql select carrier, avg(depdelay)
      from flights
      group by carrier
```



Let's explore this data for flight delays, when the departure delay is greater than 40 minutes. Below we see that United Airlines and Delta have the highest count of flight delays for January and February 2017 (the training set).

// Count of Departure Delays by Carrier (where delay=40 minutes)

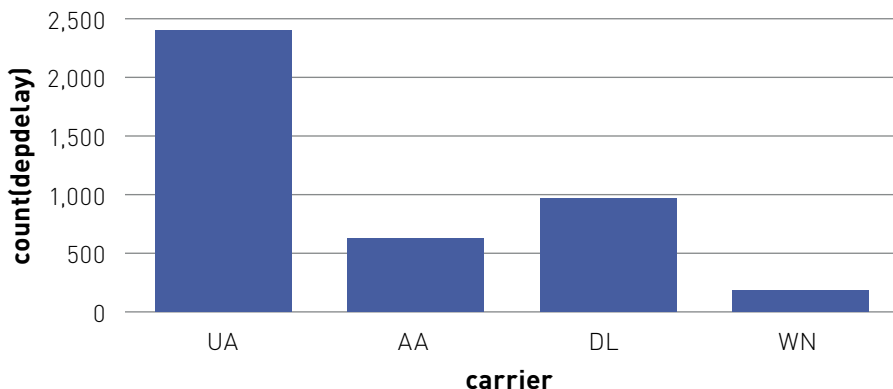
```
df.filter($"depdelay" > 40)
  .groupBy("carrier").count.orderBy(desc("count")).show(5)
```

result:

```
+-----+-----+
|carrier|count|
+-----+-----+
|      UA| 2420|
|      DL| 1043|
|      AA|  757|
|      WN|  244|
+-----+-----+
```

// Count of Departure Delays by Carrier (where delay=40 minutes)

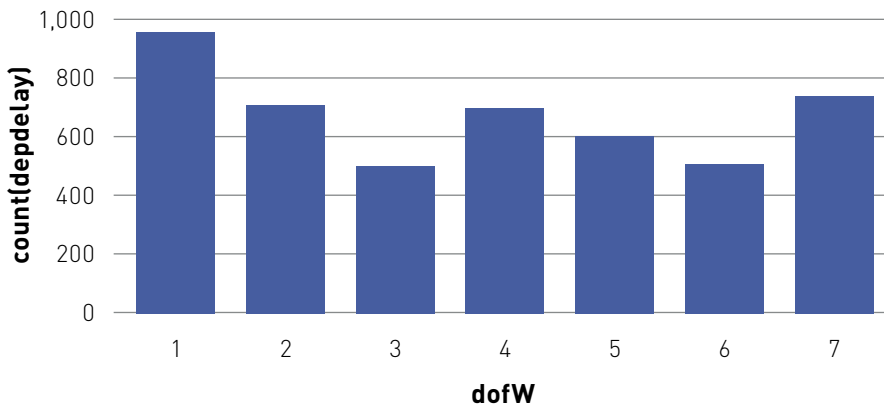
```
%sql
select carrier, count(depdelay)
from flights where depdelay > 40
group by carrier
```



In the query below, we see that Monday (1), Tuesday (2), and Sunday (7) have the highest count of flight delays.

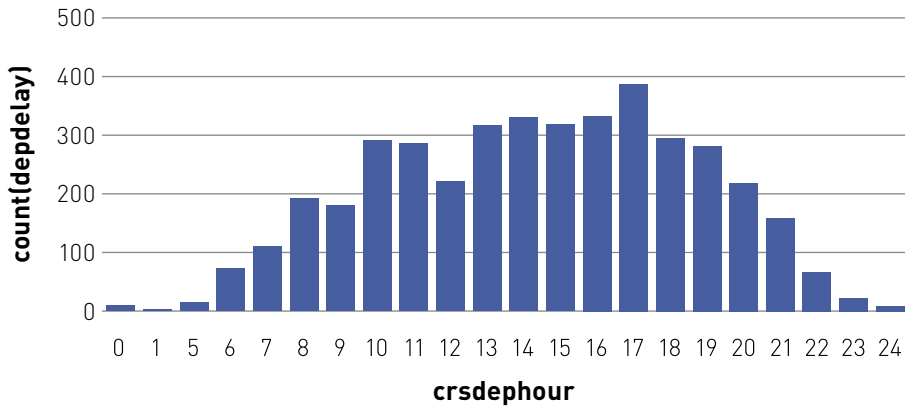
// Count of Departure Delays by Day of the Week

```
%sql
select dofW, count(depdelay)
from flights where depdelay > 40
group by dofW
```



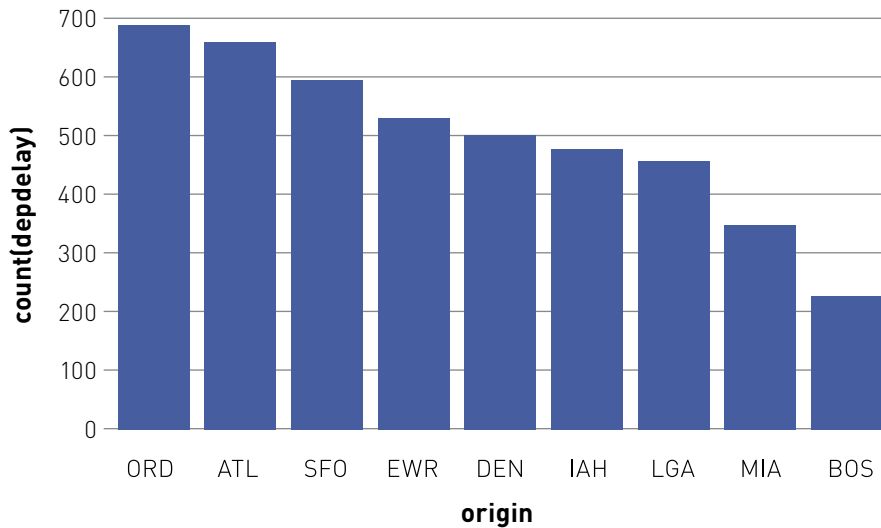
In the query below, we see that the hours between 13:00-19:00 have the highest count of flight delays.

```
%sql
select crsdephour, count(depdelay)
from flights where depdelay > 40
group by crsdephour order by crsdephour
```



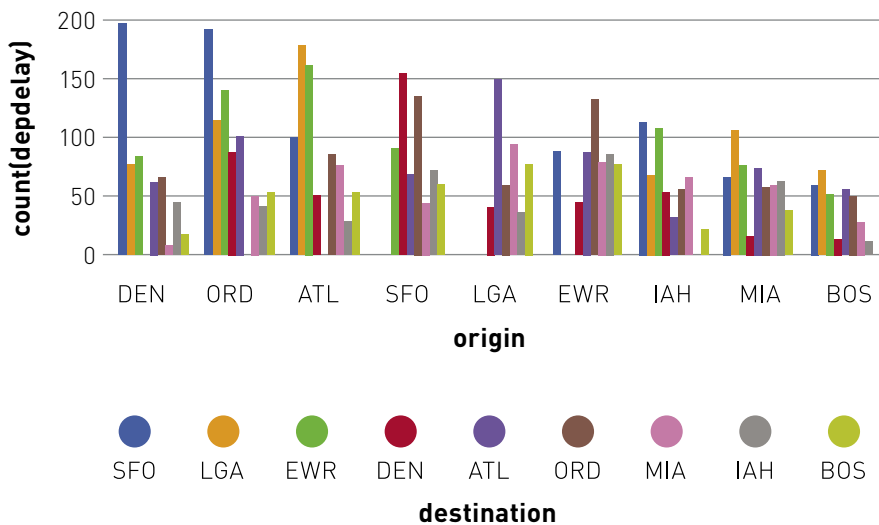
In the query below, we see that the originating airports, Chicago and Atlanta, have the highest count of flight delays.

```
%sql
select origin, count(depdelay)
from flights where depdelay > 40
group by origin
ORDER BY count(depdelay) desc
```



In the query below, we see the count of departure delays by origin and destination. The routes ORD->SFO and DEN->SFO have the highest delays, maybe because of weather in January and February. Adding weather to this Dataset would give better results.

```
%sql
select origin, dest, count(depdelay)
from flights where depdelay > 40
group by origin, dest
ORDER BY count(depdelay) desc
```

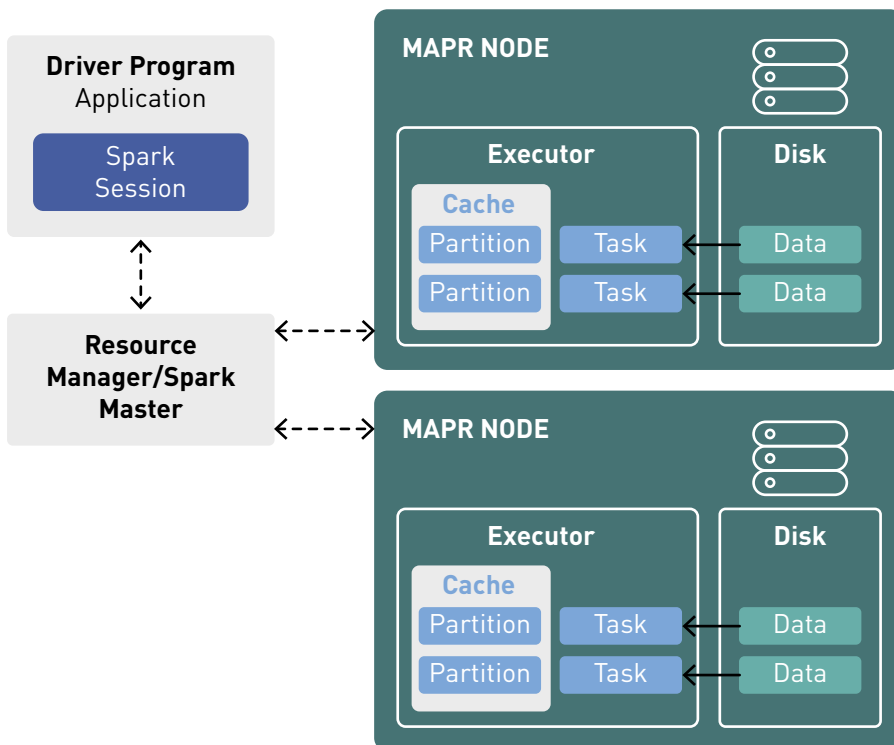


Summary

You have now learned how to load data into Spark Datasets and DataFrames and how to explore tabular data with Spark SQL. These code examples can be reused as the foundation to solve many types of business problems. In later chapters, we will use the same data with DataFrames for machine learning and graph analysis of flight delays.

How Spark Runs Your Applications

Recall from [chapter 1](#) that your Spark Application runs as a set of parallel tasks. In this chapter, we will go over how Spark translates Dataset transformations and actions into an execution model. In order to understand how your application runs on a cluster, an important thing to know about Dataset transformations is that they fall into two types, narrow and wide, which we will discuss first, before explaining the execution model.



Narrow and Wide Transformations

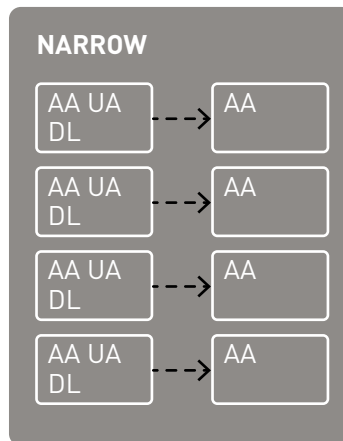
As a review, transformations create a new Dataset from an existing one. Narrow transformations do not have to move data between partitions when creating a new Dataset from an existing one. Some example narrow transformations are “filter” and “select,” which are used in the example below to retrieve flight information for the carrier “AA”:

```
// select and filter are narrow transformations
df.select($"carrier", $"origin", $"dest", $"depdelay",
  $"crsdephour").filter($"carrier" === "AA" ).show(2)
```

result:

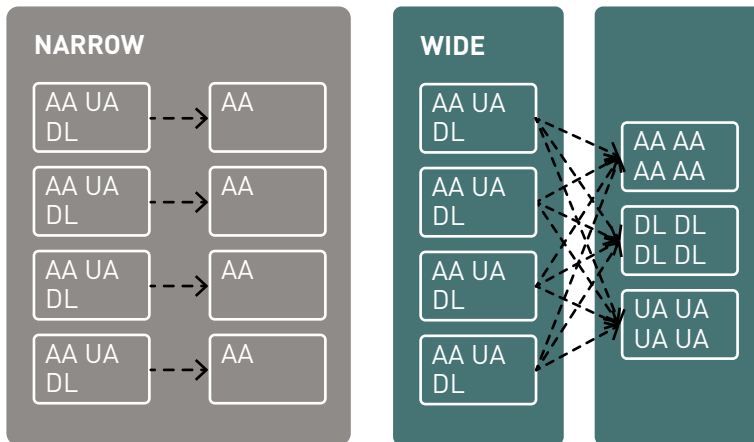
```
+-----+-----+-----+-----+-----+
|carrier|origin|dest|depdelay|crsdephour|
+-----+-----+-----+-----+-----+
|      AA|   ATL|  LGA|      0.0|         17|
|      AA|   LGA|  ATL|      0.0|         13|
+-----+-----+-----+-----+-----+
```

Multiple narrow transformations can be performed on a Dataset in memory, in a process called pipelining, making narrow transformations very efficient.



Wide transformations cause data to be moved between partitions when creating a new Dataset, in a process called the shuffle. With wide transformation shuffles, data is sent across the network to other nodes and written to disk, causing network and disk I/O, and making the shuffle a costly operation. Some example wide transformations are “groupBy,” “agg,” “sortBy,” and “orderBy.” Below is a wide transformation to count the number of flights by carrier.

```
df.groupBy("carrier").count.show
+-----+-----+
|carrier|count|
+-----+-----+
|      UA|18873|
|      AA|10031|
|      DL|10055|
|      WN| 2389|
+-----+-----+
```



The Spark Execution Model

The Spark execution model can be defined in three phases: creating the logical plan, translating that into a physical plan, and then executing the tasks on a cluster.

You can view useful information about your Spark jobs in real time in a web browser with this URL: <http://<driver-node>:4040>. For Spark applications that have finished, you can use the Spark history server to see this information in a web browser at this URL: <http://<server-url>:18080>. Let’s walk through the three phases and the Spark UI information about the phases, with some sample code.

The Logical Plan

In the first phase, the logical plan is created. This is the plan that shows which steps will be executed when an action gets applied. Recall that when you apply a transformation on a Dataset, a new Dataset is created. When this happens, that new Dataset points back to the parent, resulting in a lineage or directed acyclic graph (DAG) for how Spark will execute these transformations.

The Physical Plan

Actions trigger the translation of the logical DAG into a physical execution plan. The Spark Catalyst query optimizer creates the physical execution plan for DataFrames, as shown in the diagram below:

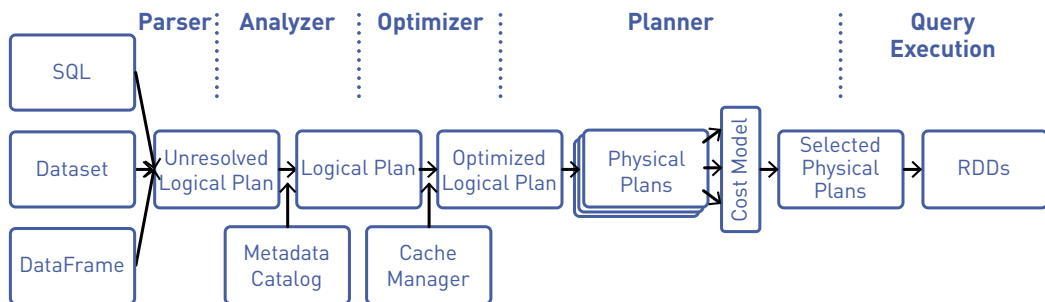


Image reference: Databricks

The physical plan identifies resources, such as memory partitions and compute tasks, that will execute the plan.

Viewing the Logical and Physical Plan

You can see the logical and physical plan for a Dataset by calling the `explain(true)` method. In the code below, we see that the DAG for `df2` consists of a `FileScan`, a `Filter` on `depdelay`, and a `Project` (selecting columns).

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

var file = "maprfs:///data/flights20170102.json"

case class Flight(_id: String, dofW: Long, carrier: String,
  origin: String, dest: String, crsdephour: Long, crsdeptime:
  Double, depdelay: Double, crsarrrtime: Double, arrdelay: Double,
  crselapsedtime: Double, dist: Double) extends Serializable

val df = spark.read.format("json").option("inferSchema", "true").
  load(file).as[Flight]

val df2 = df.filter($"depdelay" > 40)

df2.take(1)

result:
Array[Flight] = Array(Flight(MIA_IAH_2017-01-01_AA_2315,
7,AA,MIA,IAH,20,2045.0,80.0,2238.0,63.0,173.0,964.0))

df2.explain(true)
```

```

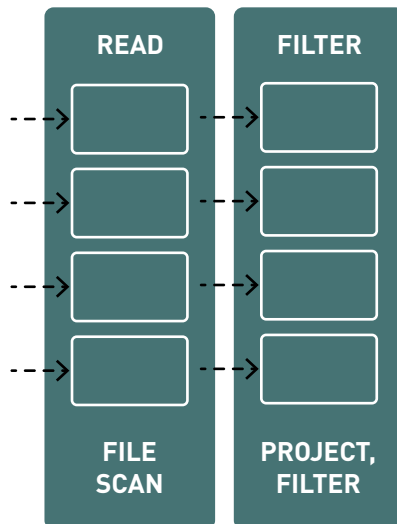
result:
== Parsed Logical Plan ==
`Filter (`depdelay > 40)
+- Relation[_id#8,arrdelay#9,...] json

== Analyzed Logical Plan ==
_id: string, arrdelay: double...
Filter (depdelay#15 > cast(40 as double))
+- Relation[_id#8,arrdelay#9,...] json

== Optimized Logical Plan ==
Filter (isnotnull(depdelay#15) && (depdelay#15 > 40.0))
+- Relation[_id#8,arrdelay#9,...] json

== Physical Plan ==
*Project [_id#8, arrdelay#9,...]
+- *Filter (isnotnull(depdelay#15) && (depdelay#15 > 40.0))
   +- *FileScan json [_id#8,arrdelay#9,...] Batched: false, Format:
      JSON, Location: InMemoryFileIndex[maprfs:///...],

```



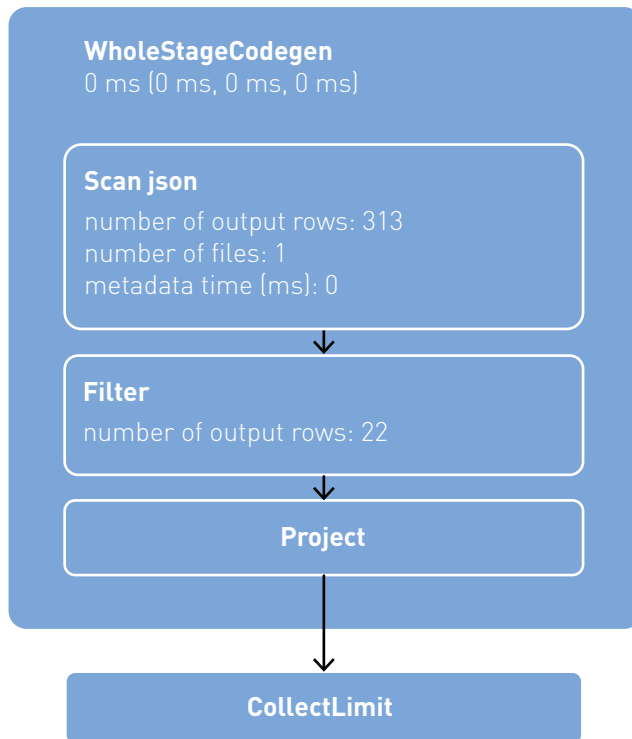
You can see more details about the plan produced by Catalyst on the web UI SQL tab (<http://<driver-node>:4040/SQL/>). Clicking on the query description link displays the DAG and details for the query.

DETAILS FOR QUERY 0

Submitted Time: 2018/07/31 1:22:12

Duration: 0.9 s

Succeeded Jobs: 1



In the code below, after the `explain`, we see that the physical plan for `df3` consists of a `FileScan`, `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.

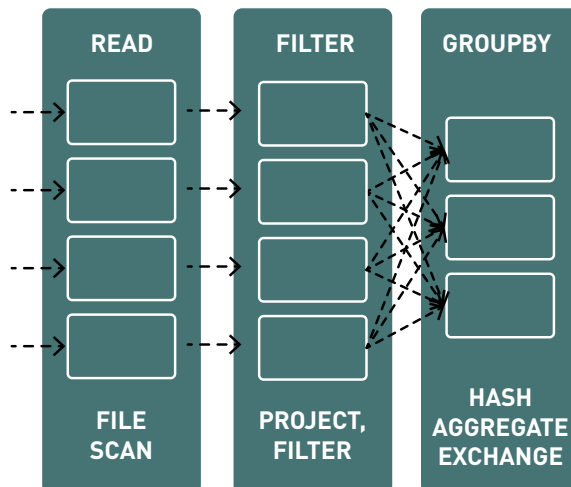
```
val df3 = df2.groupBy("carrier").count

df3.collect

result:
Array[Row] = Array([UA,2420], [AA,757], [DL,1043], [WN,244])

df3.explain

result:
== Physical Plan ==
*HashAggregate(keys=[carrier#124], functions=[count(1)])
+- Exchange hashpartitioning(carrier#124, 200)
   +- *HashAggregate(keys=[carrier#124], functions=[partial_
count(1)])
      +- *Project [carrier#124]
         +- *Filter (isNotNull(depdelay#129) && (depdelay#129 >
40.0))
            +- *FileScan json [carrier#124,depdelay#129]
```

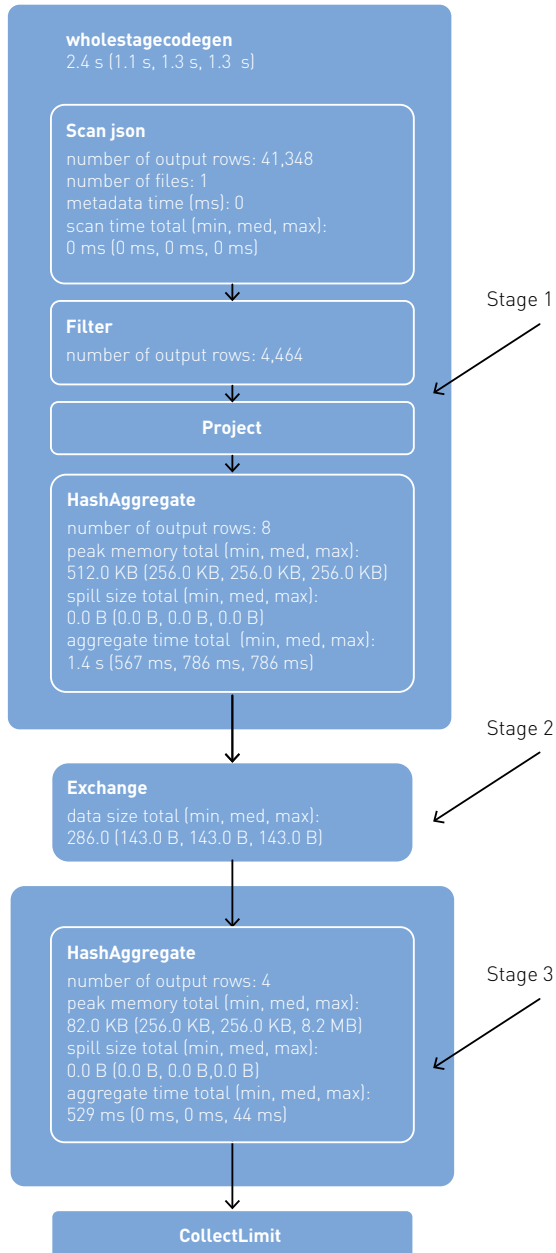


Clicking on the SQL tab link for this query displays the DAG below.

Submitted Time: 2018/07/31 18:51:49

Duration: 5 s

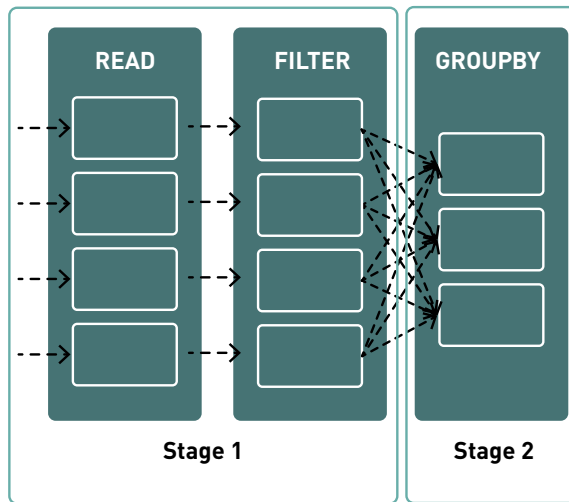
Succeeded Jobs: 2 3 4 5 6



Executing the Tasks on a Cluster

In the third phase, 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 (pipe-lined) together into a single stage. The physical plan for this example has two stages, with everything before the exchange in the first stage.

Physical Plan

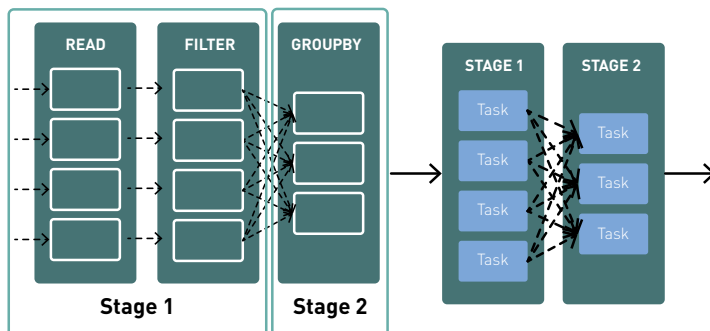


Each stage is comprised of tasks, based on partitions of the Dataset, which will perform the same computation in parallel.

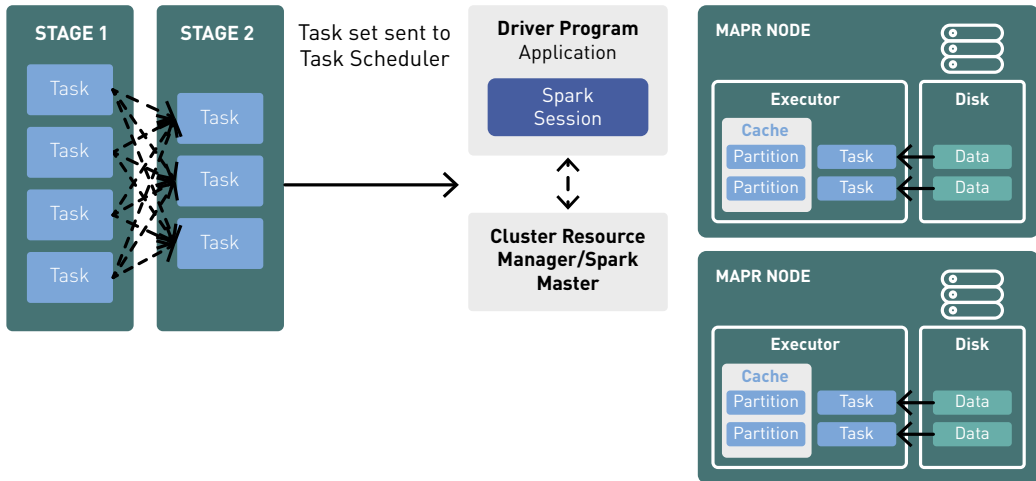
Physical Plan

Split into Tasks

Task Set



The scheduler submits the stage task set to the task scheduler, which launches tasks via a cluster manager. These phases are executed in order and the action is considered complete when the final phase in a job completes. This sequence can occur many times when new Datasets are created.

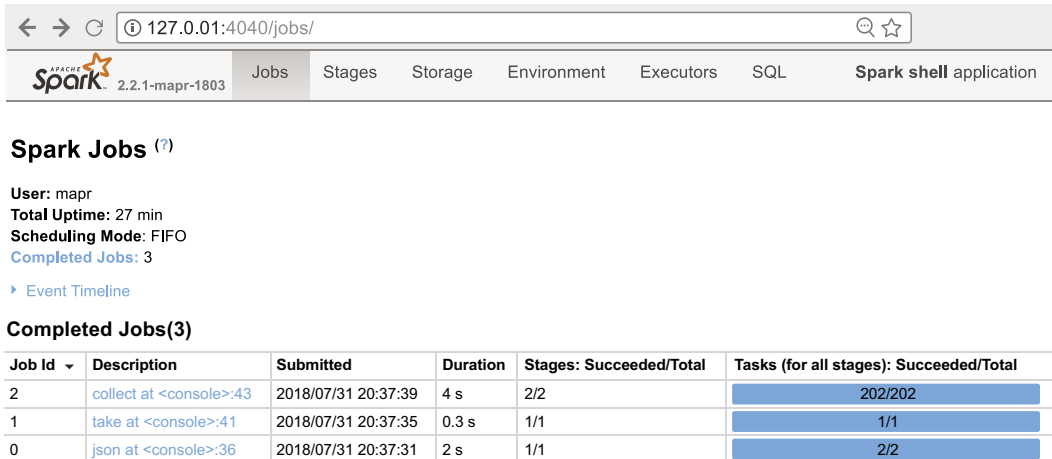


Here is a summary of the components of execution:

- **Task:** a unit of execution that runs on a single machine
- **Stage:** a group of tasks, based on partitions of the input data, which will perform the same computation in parallel
- **Job:** has one or more stages
- **Pipelining:** collapsing of Datasets into a single stage, when Dataset transformations can be computed without data movement
- **DAG:** Logical graph of Dataset operations

Exploring the Task Execution on the Web UI

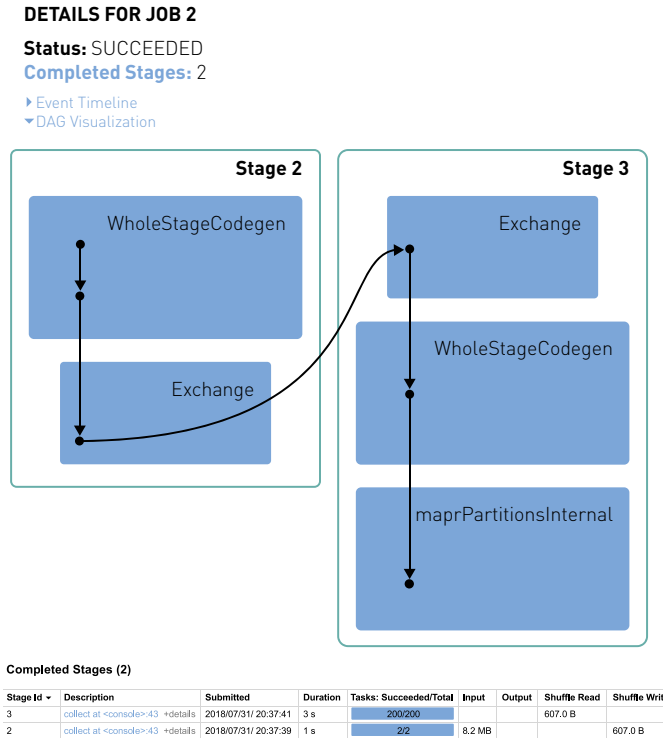
Here is a screen shot of the web UI Jobs tab, after running the code above. The Jobs page gives you detailed execution information for active and recently completed Spark jobs. It gives you the performance of a job and also the progress of running jobs, stages, and tasks. In this example, Job Id 2 is the job that was triggered by the collect action on df3.



The screenshot shows the Apache Spark Web UI interface. The browser address bar displays '127.0.0.1:4040/jobs/'. The navigation bar includes tabs for 'Jobs', 'Stages', 'Storage', 'Environment', 'Executors', 'SQL', and 'Spark shell application'. The 'Jobs' tab is selected, showing job details for 'User: mapr', 'Total Uptime: 27 min', 'Scheduling Mode: FIFO', and 'Completed Jobs: 3'. A link for 'Event Timeline' is visible. Below, the 'Completed Jobs(3)' section contains a table with columns: Job Id, Description, Submitted, Duration, Stages: Succeeded/Total, and Tasks (for all stages): Succeeded/Total. The table lists three jobs, with Job Id 2 being the most recent and having 202/202 tasks completed.

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	collect at <console>:43	2018/07/31 20:37:39	4 s	2/2	202/202
1	take at <console>:41	2018/07/31 20:37:35	0.3 s	1/1	1/1
0	json at <console>:36	2018/07/31 20:37:31	2 s	1/1	2/2

Clicking the link in the Description column on the Jobs page takes you to the Job Details page. This page gives you details on the progress of the job, stages, and tasks. We see this job consists of 2 stages, with 2 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 2 partitions; after a shuffle, the default number of partitions is 200. You can see the number of partitions on a Dataset with the `rdd.partitions.size` method shown below.

```
df3.rdd.partitions.size
result: Int = 200

df2.rdd.partitions.size
result: Int = 2
```

Under the Stages tab, you can see the details for a stage by clicking on its link in the description column.

Stages for All Jobs

Completed Jobs: 4

Completed Stages(4)

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
3	collect at <console>:43 +details	2018/07/31/ 20:37:41	3 s	200/200			607.0 B	
2	collect at <console>:43 +details	2018/07/31/ 20:37:39	1 s	2/2	8.2 MB			607.0 B
1	take at <console>:41 +details	2018/07/31/ 20:37:35	0.2 s	1/1	6.1 MB			
0	json at <console>:36 +details	2018/07/31/ 20:37:31	2 s	2/2	8.2 MB			

Here we have summary metrics and aggregated metrics for tasks, and aggregated metrics by executor. You can use these metrics to identify problems with an executor or task distribution. If your task process time is not balanced, then resources could be wasted.

The Storage tab provides information about persisted Datasets. The dataset is persisted if you called `persist` or `cache` on the dataset, followed by an action to compute on that Dataset. This page tells you which fraction of the Dataset's underlying RDD is cached and the quantity of data cached in various storage media. Look at this page to see if important Datasets are fitting into memory. You can also click on the link to view more details about the persisted Dataset. If you no longer need a cached Dataset, you can call `Unpersist` to uncache it.

Storage

RDDs

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
*Project [_id#8, arrdelay#9, carrier#10, crstime#11L, crsdephour#12L, crsdeptime#13L, crselapsedtime#14, depdelay#15, dest#16, dist#17, dofW#18L, origin#19] +- Filter (isontnull(depdelay#15) \$\$ (depdelay #15> 40.0)) +-FileScan json [_id#8, arrdelay#9, carrier#10, crstime#11L, crsdephour#12L, crsdeptime#13L, crselapsedtime#14, depdel...	Memory Deserialized 1x Replicated	2	100%	330.4 KB	0.0 B

Try caching df2, performing an action, then seeing how this gets persisted on the storage tab and how it changes the plan and execution time for df3 on the job details page. Notice how the execution time is faster after caching.

```
df2.cache
df2.count
df3.collect
```

Notice how the first stage is skipped in job4, when df2 is cached and df3 collect is executed again.

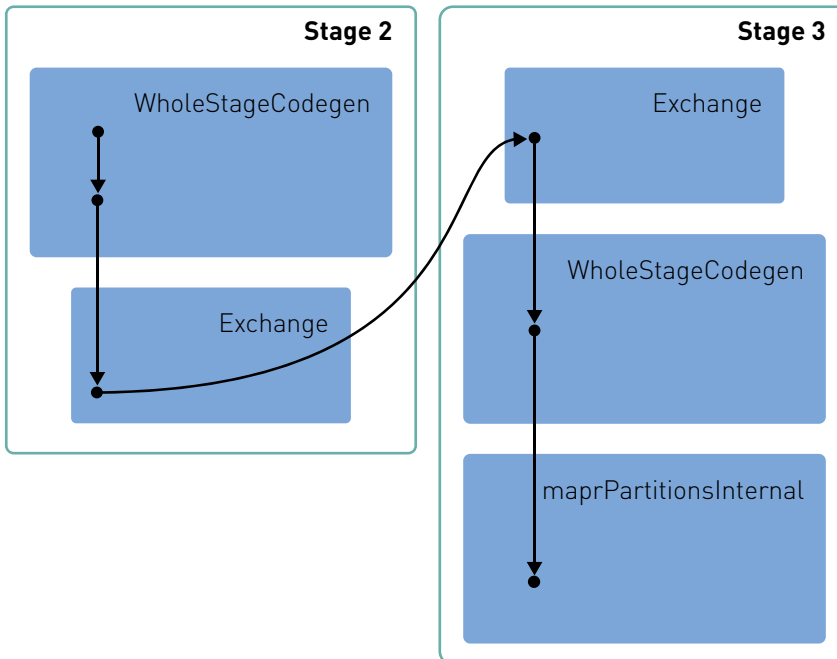
DETAILS FOR JOB 2

Status: SUCCEEDED

Completed Stages: 2

► Event Timeline

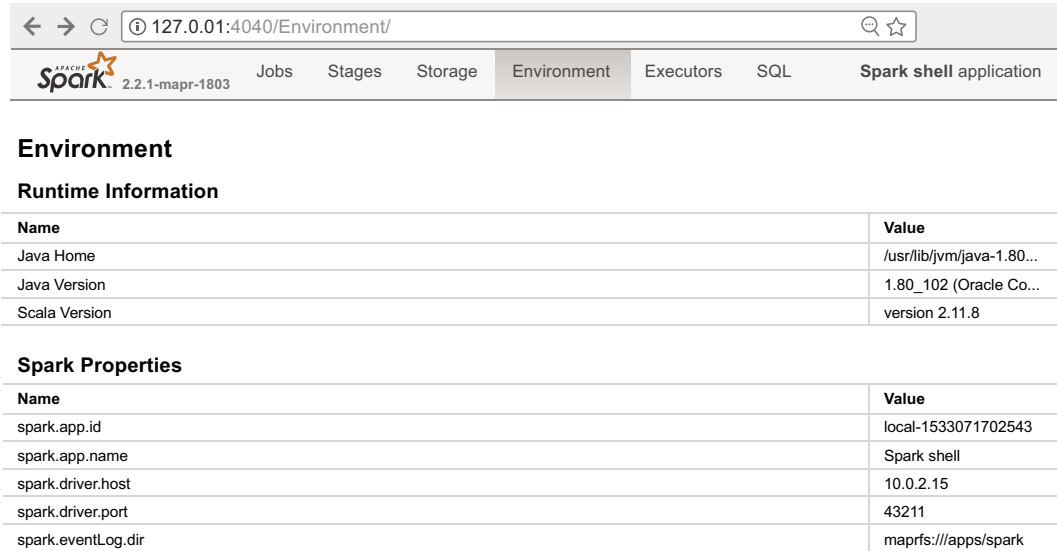
▼ DAG Visualization



Completed Stages (2)

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
3	collect at <console>:43 +details	2018/07/31/ 20:37:41	3 s	<div><div>200/200</div></div>			607.0 B	
2	collect at <console>:43 +details	2018/07/31/ 20:37:39	1 s	<div><div>2/2</div></div>	8.2 MB			607.0 B

The Environment tab lists all the active properties of your Spark application environment. Use this page when you want to see which configuration flags are enabled. Only values specified through `spark-defaults.conf`, `SparkSession`, or the command line will be displayed here. For all other configuration properties, the default value is used.



Environment

Runtime Information

Name	Value
Java Home	/usr/lib/jvm/java-1.80...
Java Version	1.80_102 (Oracle Co...
Scala Version	version 2.11.8

Spark Properties


Name	Value
spark.app.id	local-1533071702543
spark.app.name	Spark shell
spark.driver.host	10.0.2.15
spark.driver.port	43211
spark.eventLog.dir	maprfs:///apps/spark

Under the Executors tab, you can see processing and storage for each executor:

- Shuffle Read Write Columns: shows size of data transferred between stages
- Storage Memory Column: shows the current used/available memory
- Task Time Column: shows task time/garbage collection time

Use this page to confirm that your application has the amount of resources you were expecting. You can look at the thread call stack by clicking on the thread dump link.

← → ↻ 127.0.0.1:4040/Executors/ 🔍 ☆

 2.2.1-mapr-1803
 Jobs Stages Storage Environment **Executors** SQL Spark shell application

Executors

[▶ Show Additional Metrics](#)

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Tasks Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(1)	0	0.0 B / 384.1 MB	0.0 B	2	0	0	204	204	9 s (0.2 s)	17.2 MB	0.0 B	607 B	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Total (1)	0	0.0 B / 384.1 MB	0.0 B	2	0	0	204	204	9 s (0.2 s)	17.2 MB	0.0 B	607 B	0

Executors

Show entries

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Tasks Time (GC Time)	Input	Shuffle Read	Shuffle Write	Thread Dump
driver	10.0.2.15:36610	Active	0	0.0 B / 384.1 MB	0.0 B	2	0	0	204	204	9 s (0.2 s)	17.2 MB	0.0 B	607 B	Thread Dump

Showing 1 to 1 of 1 entries

Summary

In this chapter, we discussed the Spark execution model, and we explored task execution on the Spark Web UI. This understanding of how Spark runs your applications is important when debugging, analyzing, and tuning the performance of your applications.

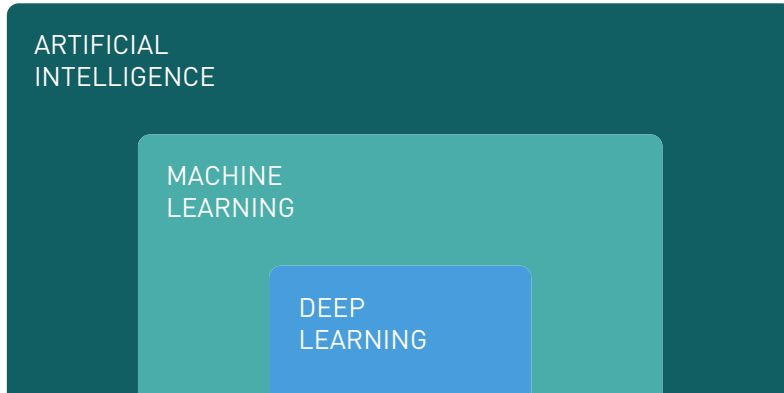
Demystifying AI, Machine Learning, and Deep Learning

Deep learning, machine learning, artificial intelligence – all buzzwords and representative of the future of analytics. In this chapter, we will explain machine learning and deep learning at a high level with some real world use cases. In the next three chapters, we will explore some machine learning examples with Apache Spark. The goal is to give you a better understanding of what you can do with machine learning. Machine learning is becoming more accessible to developers, and data scientists work with domain experts, architects, developers, and data engineers, so it is important for everyone to have a better understanding of the possibilities. Every piece of information that your business generates has potential to add value. This overview is meant to provoke a review of your own data to identify new opportunities.

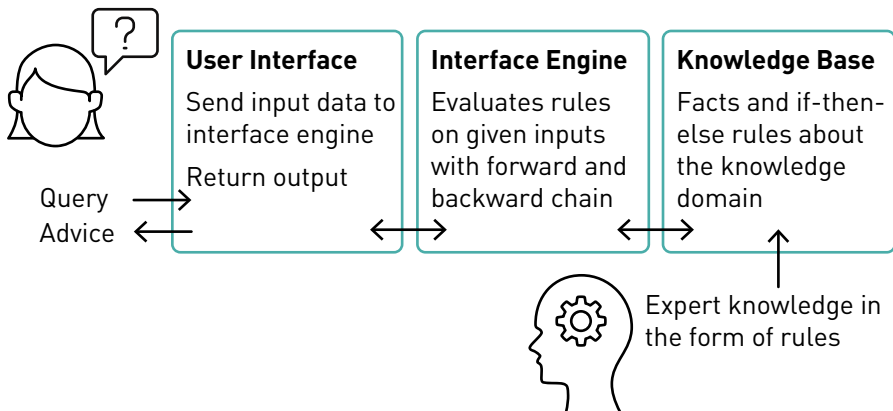
Retail	Marketing	Healthcare	Telco	Finance
Demand Forecasting	Recommendation engines and targeting	Predicting patient disease risk	Customer churn	Risk analytics
Supply chain optimization	Customer 360	Diagnostics and alerts	System log analysis	Customer 360
Pricing optimization	Click-stream analysis	Fraud	Anomaly detection	Fraud
Market segmentation and targeting	Social media analysis		Preventive maintenance	Credit scoring
Recommendations	Ad optimization		Smart meter analysis	

What is Artificial Intelligence?

Throughout the history of AI, the definition has been continuously redefined. AI is an umbrella term for an idea that started in the 50s; machine learning is a subset of AI; and deep learning is a subset of ML.



In the late 80s, when I was a student interning at the NSA, AI was also a very hot topic. At the NSA, I took an MIT video (VCR) class on AI, which was about expert systems. Expert systems capture an expert's knowledge in a rules engine.



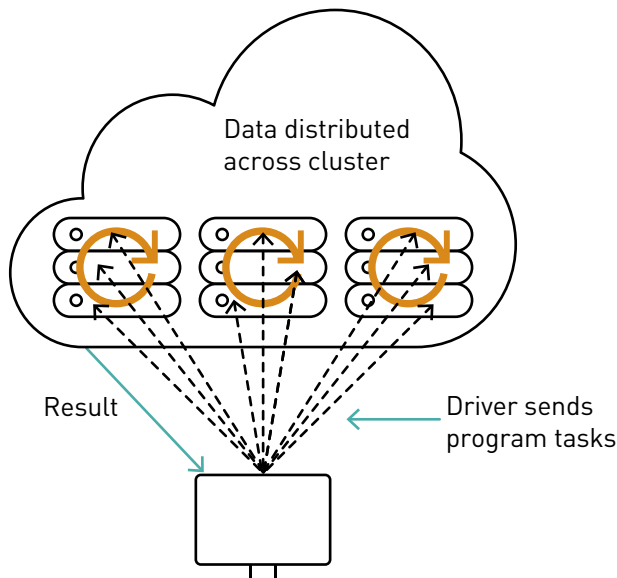
Rules engines have wide use in industries such as finance and healthcare, and more recently for [event processing](#), but when data is changing, rules can become difficult to update and maintain. Machine learning has the advantage that it [learns from the data](#), and it can give finer grained data-driven probabilistic predictions.

[According to Ted Dunning](#), it is better to use precise terminology, like machine learning or deep learning, instead of the word AI, because before we get something to work well, we call it AI; afterwards, we always call it something else. AI is better used as a word for the next frontier.

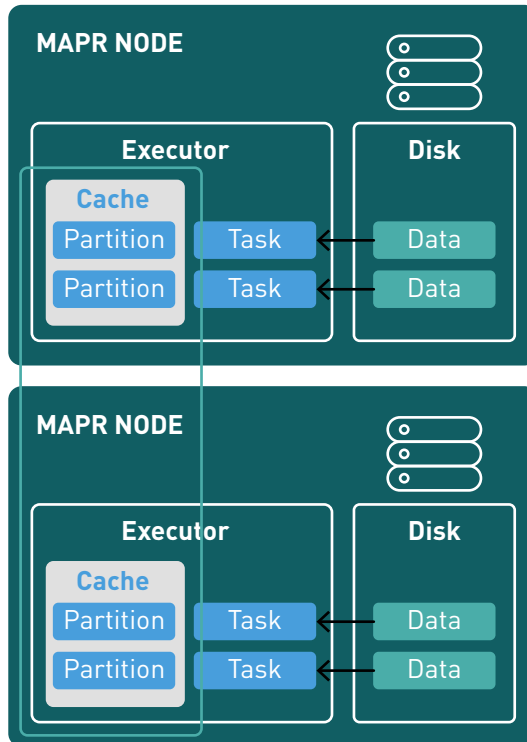
How Has Analytics Changed in the Last 10 Years?

According to Thomas Davenport's update to the *Competing on Analytics* book, [analytical technology has changed dramatically over the last decade](#), with more powerful and less expensive distributed computing across commodity servers, streaming analytics, and improved machine learning technologies, enabling companies to store and analyze both far more data and many different types of it.

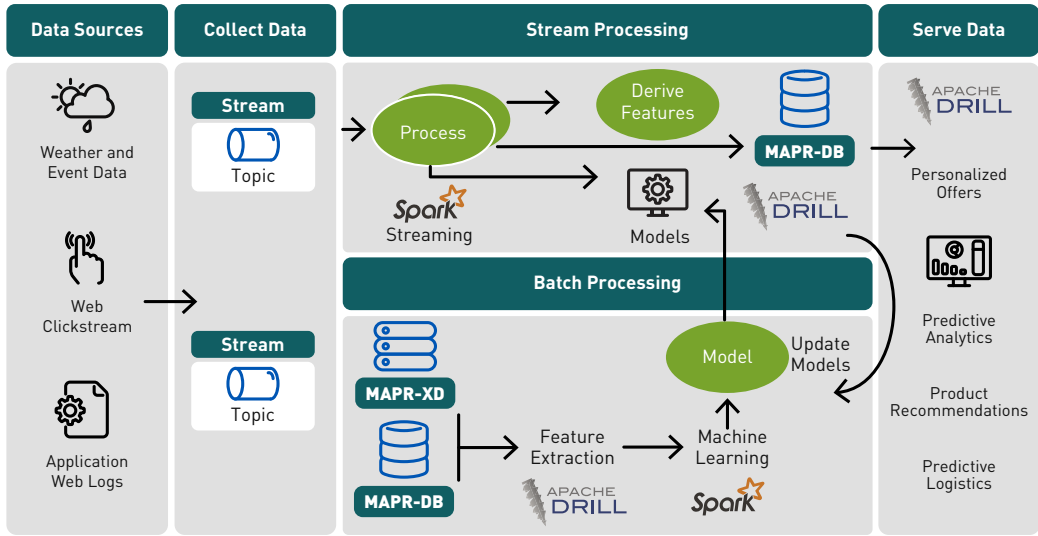
Traditionally, data was stored on a RAID system, sent to a multi-core server for processing, and sent back for storage, which was expensive and caused a bottleneck on data transfer. With file and table storage like [MapR-XD](#) and MapR-DB, data is distributed across a cluster.



Technologies like [Apache Spark](#) speed up parallel processing of distributed data with [iterative algorithms](#) by caching data in memory across iterations and using lighter weight threads.



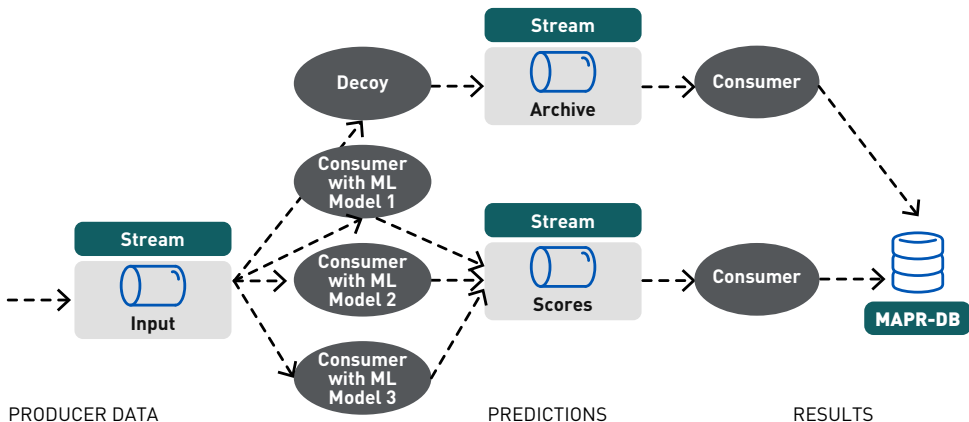
MapR Event Streams, a distributed messaging system for streaming event data at scale, combined with stream processing like Apache Spark Streaming, speed up parallel processing of real-time events with machine learning models.



Event Streams and Machine Learning Logistics

Combining event streams with machine learning can handle the logistics of machine learning in a flexible way by:

- Making input and output data available to independent consumers
- Managing and evaluating multiple models and easily deploying new models



Architectures for these types of applications are discussed in more detail in the [ebooks Machine Learning Logistics, Streaming Architecture, and Microservices and Containers](#).

Graphical Processing Units (GPUs) have sped up multi-core servers for parallel processing. A GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously, whereas a CPU consists of a few cores optimized for sequential serial processing. [In terms of potential performance, the evolution from the Cray-1 to today's clusters with lots of GPUs is roughly a million times what was once the fastest computer on the planet at a tiny fraction of the cost.](#)

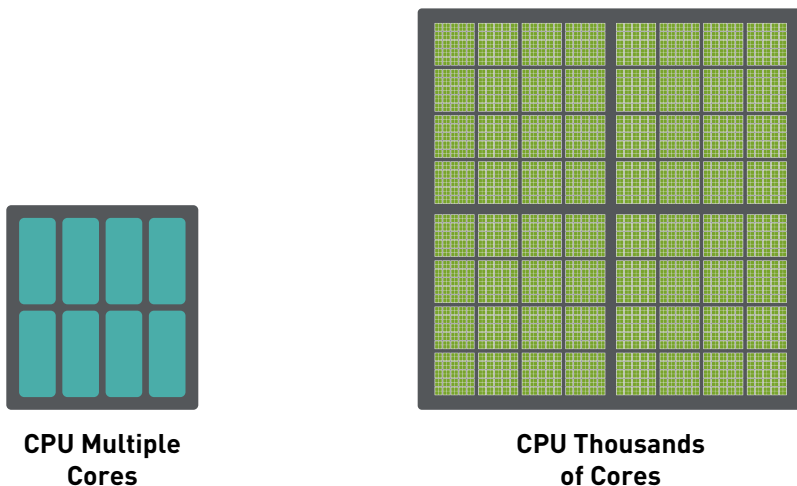
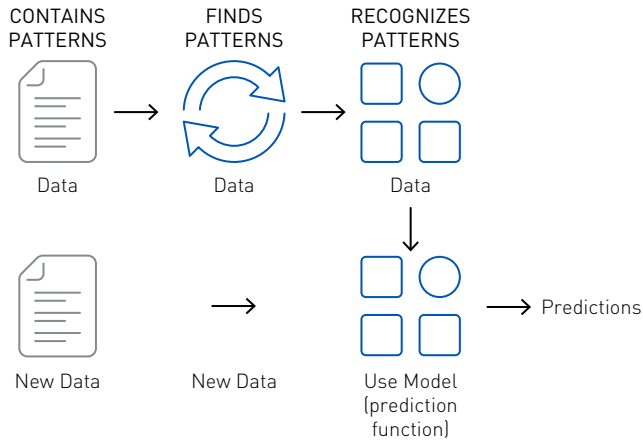


Image reference: <http://www.nvidia.com/object/what-is-gpu-computing.html>

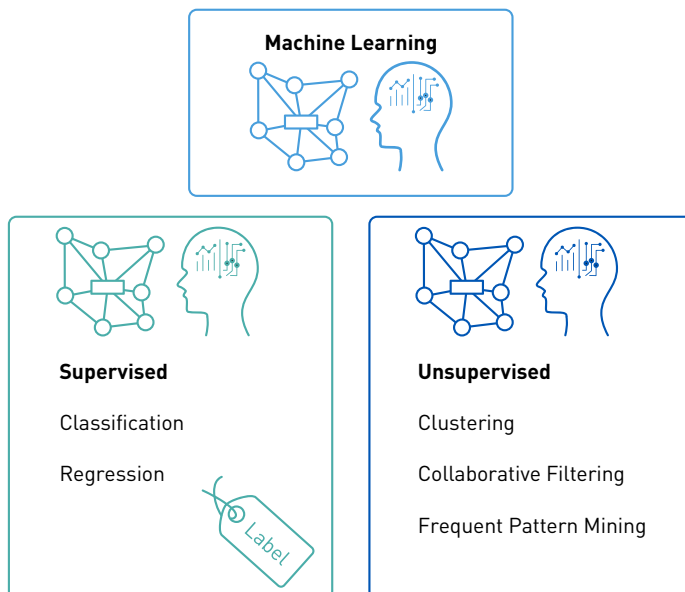
Improved technologies for parallel processing of distributed data, streaming analytics, and machine learning have enabled faster machine learning predictions and recommendations, even approaching real time in many cases.

What is Machine Learning?

Machine learning uses algorithms to find patterns in data, and then uses a model that recognizes those patterns to make predictions on new data.



In general, machine learning may be broken down into two types: supervised, unsupervised, and in between those two. Supervised learning algorithms use labeled data; unsupervised learning algorithms find patterns in unlabeled data. Semi-supervised learning uses a mixture of labeled and unlabeled data. Reinforcement learning trains algorithms to maximize rewards based on feedback.



Three Common Categories of Techniques for Machine Learning

Three common categories of machine learning techniques are Classification, Clustering, and Collaborative Filtering.

Classification

<
>
↺

<input type="checkbox"/>	Mr. Norman	Accept My Donation
<input type="checkbox"/>	Lending	Simple Loans
<input type="checkbox"/>	election time	Please Help My Campaign
<input type="checkbox"/>	Hi friend	Limited time offer
<input type="checkbox"/>	confirm	Confirmation Needed Now

Clustering

<
>
↺

Business
Technology
Entertainment
Heath
Sports
Science

Lorem ipsum dolor sit amet, consectetur qui adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna dignissim blandit

Veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in

Vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio

Collaborative Filtering

(Recommendation)

<
>
↺

Book 1

★★★★☆

Book 2

★★★★☆

Book 3

★★★★☆

Book 4

★★★★★

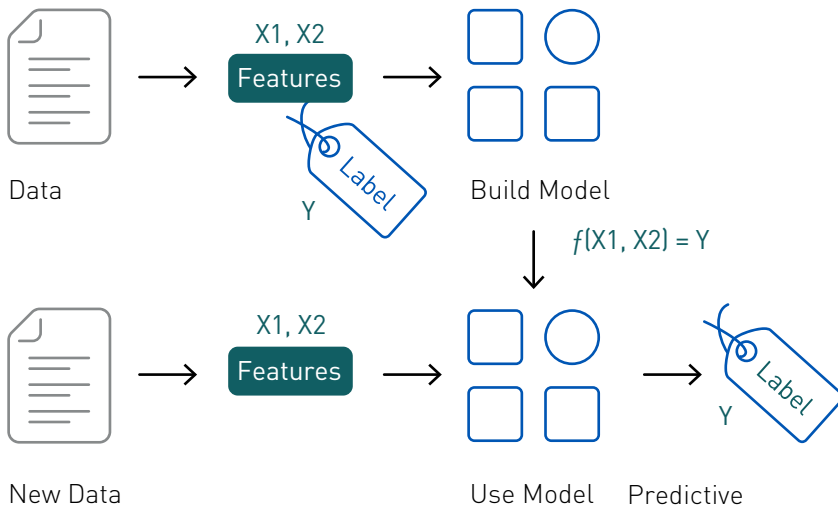
Classification: Gmail uses a machine learning technique called classification to designate if an email is spam or not, based on the data of an email: the sender, recipients, subject, and message body. Classification takes a set of data with known labels and learns how to label new records based on that information.

Clustering: Google News uses a technique called clustering to group news articles into different categories, based on title and content. Clustering algorithms discover groupings that occur in collections of data.

Collaborative Filtering: Amazon uses a machine learning technique called collaborative filtering (commonly referred to as recommendation) to determine which products users will like, based on their history and similarity to other users.

Supervised Learning: Classification and Regression

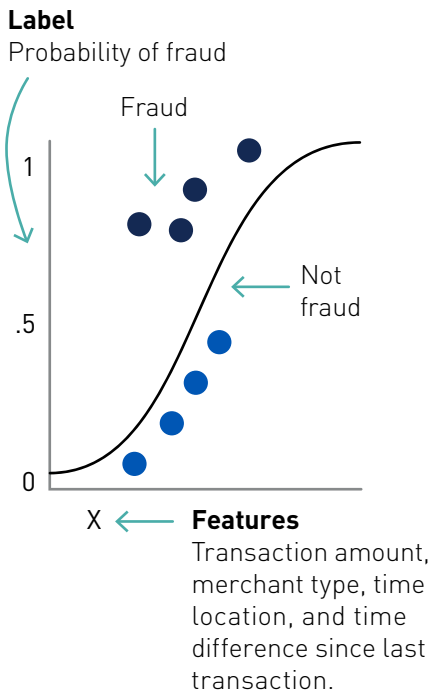
Supervised algorithms use labeled data in which both the input and target outcome, or label, are provided to the algorithm.



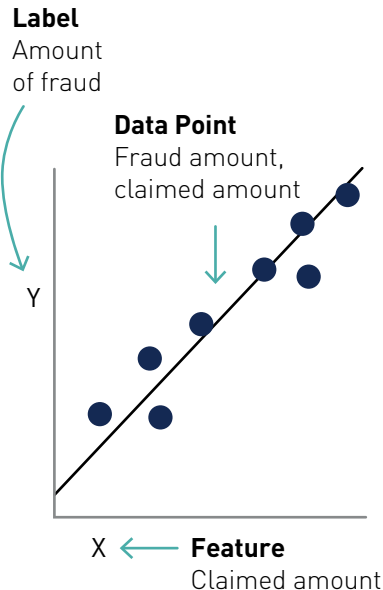
Supervised Learning is also called predictive modeling or predictive analytics, because you build a model that is capable of making predictions.

Some examples of predictive modeling are classification and regression. Classification identifies which category an item belongs to (e.g., whether a transaction is fraud or not fraud), based on labeled examples of known items (e.g., transactions known to be fraud or not). Logistic regression predicts a probability (e.g., the probability of fraud). Linear regression predicts a numeric value (e.g., the amount of fraud).

CREDIT CARD FRAUD LOGISTIC REGRESSION CLASSIFICATION EXAMPLE



CAR INSURANCE FRAUD REGRESSION EXAMPLE

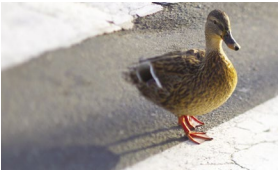

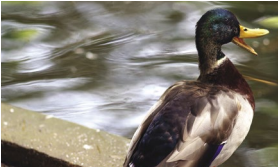


$$\text{AmntFraud} = \text{intercept} + \text{coefficient} \times \text{claimedAmnt}$$




Classification and Regression Example

Classification and Regression take a set of data with known labels and pre-determined features and learns how to label new records based on that information. Features are the “if questions” that you ask. The label is the answer to those questions.

If it walks/swims/quacks like a duck ... then it must be a duck.

Features	
Walks	
Swims	
Quacks	

Ducks

Features	
Walks	
Swims	
Quacks	

Not
Ducks

Regression Example

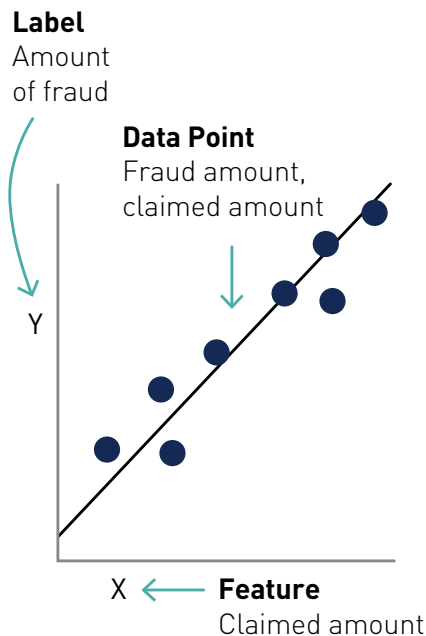
Let's go through an example of car insurance fraud:

What are we trying to predict?

- This is the Label: Amount of fraud

What are the “if questions” or properties that you can use to predict?

- These are the Features: to build a classifier model, you extract the features of interest that most contribute to the classification.
- In this simple example, we will use the claimed amount.



$$\text{AmntFraud} = \text{intercept} + \text{coefficient} \times \text{claimedAmnt}$$

Linear regression models the relationship between the Y “Label” and the X “Feature,” in this case the relationship between the amount of fraud and the claimed amount. The coefficient measures the impact of the feature, the claimed amount, on the label, the fraud amount.

Multiple linear regression models the relationship between two or more “Features” and a response “Label.” For example, if we wanted to model the relationship between the amount of fraud and the age of the claimant, the claimed amount, and the severity of the accident, the multiple linear regression function would look like this:

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

Amount Fraud = intercept + (coefficient1 * age) + (coefficient2 * claimed Amount) + (coefficient3 * severity) + error.

The coefficients measure the impact on the fraud amount of each of the features.

Some examples of linear regression include:

- Given historical car insurance fraudulent claims and features of the claims, such as age of the claimant, claimed amount, and severity of the accident, predict the amount of fraud.
- Given historical real estate sales prices and features of houses (square feet, number of bedrooms, location, etc.), predict a house’s price.
- Given historical neighborhood crime statistics, predict crime rate.

Classification Example

Let’s go through an example of Debit Card Fraud:

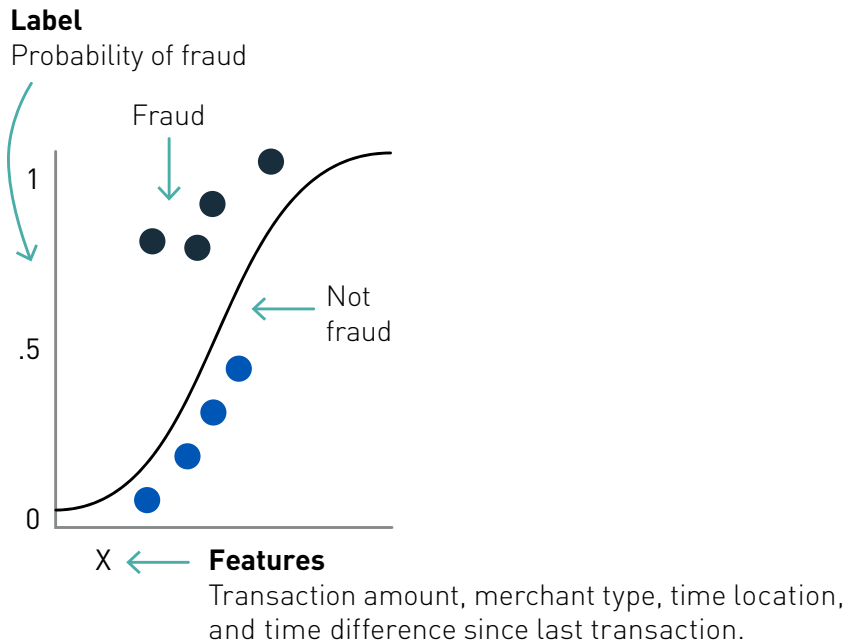
What are we trying to predict?

- This is the Label: Probability of fraud

What are the “if questions” or properties that you can use to make predictions?

- Is the amount spent today > historical average?
- Are there transactions in multiple countries today?
- Are the number of transactions today > historical average?
- Are the number of new merchant types today high compared to the last 3 months?
- Are there multiple purchases today from merchants with a category code of risk?
- Is there unusual signing activity today, compared to historically using pin?
- Are there new state purchases compared to the last 3 months?
- Are there foreign purchases today compared to the last 3 months?

To build a classifier model, you extract the features of interest that most contribute to the classification.



Logistic regression measures the relationship between the Y “Label” and the X “Features” by estimating probabilities using a [logistic function](#). The model predicts a probability, which is used to predict the label class.

Some examples of Classification include:

- Given historical car insurance fraudulent claims and features of the claims, such as age of the claimant, claimed amount, and severity of the accident, predict the probability of fraud.
- Given patient characteristics, predict the probability of congestive heart failure.
- Credit card fraud detection (fraud, not fraud)
- Credit card application (good credit, bad credit)
- Email spam detection (spam, not spam)
- Text sentiment analysis (happy, not happy)
- Predicting patient risk (high risk patient, low risk patient)
- Classifying a tumor (malignant, not malignant)

Spark Supervised Algorithms Summary

Classification

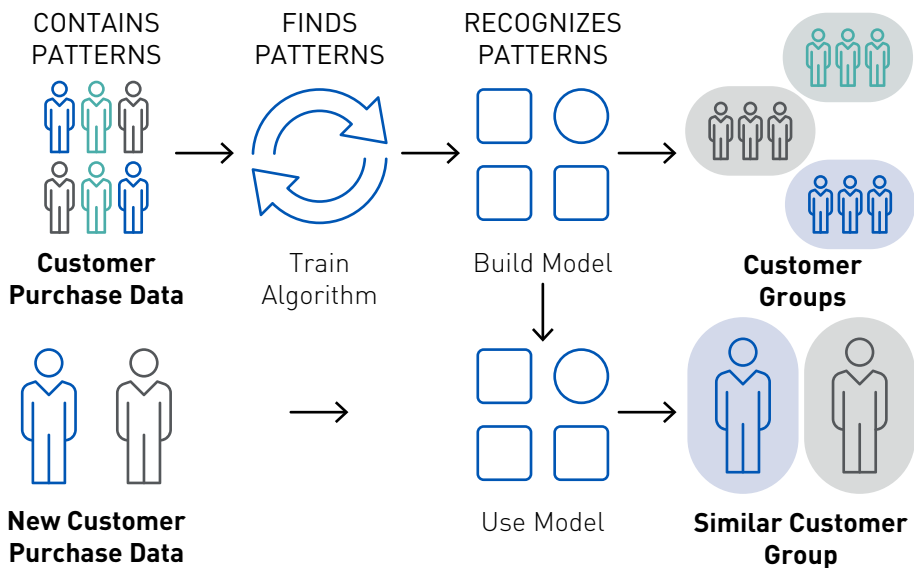
- Logistic regression
- Decision tree classifier
- Random forest classifier
- Gradient-boosted tree classifier
- Multilayer perception classifier
- Linear Support Vector Machine
- Naive Bayes

Regression

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

Unsupervised Learning

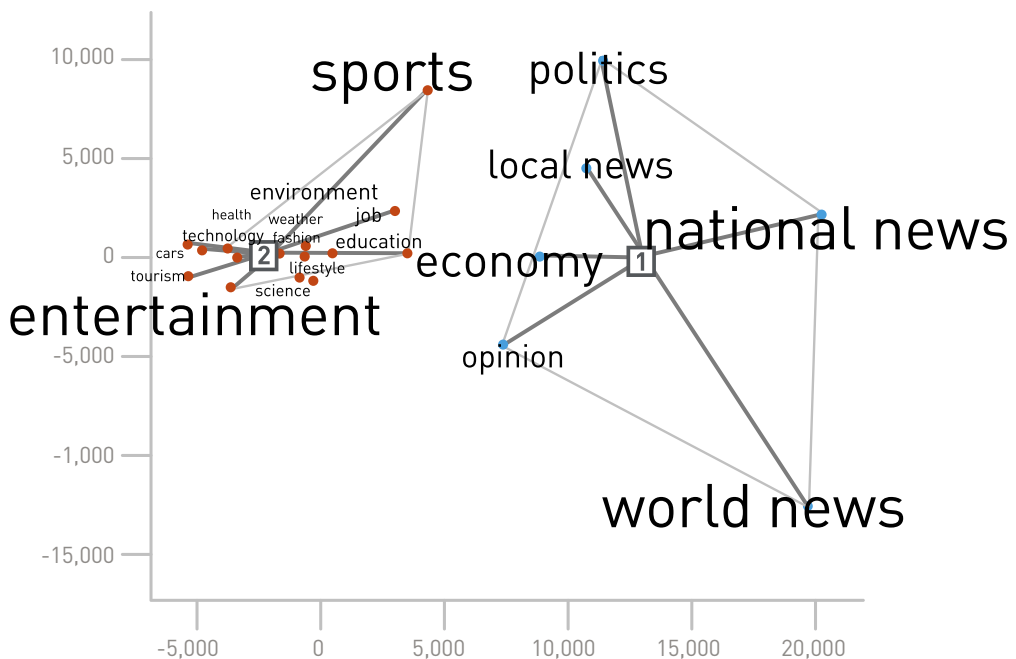
Unsupervised learning, also sometimes called descriptive analytics, does not have labeled data provided in advance. These algorithms discover similarities, or regularities, in the input data. An example of unsupervised learning is grouping similar customers, based on purchase data.



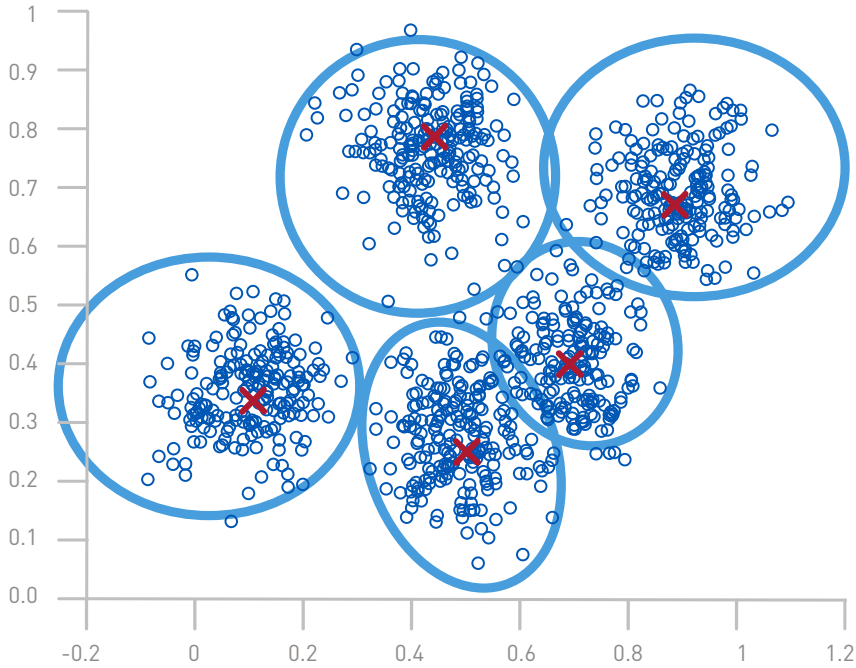
Clustering

In clustering, an algorithm classifies inputs into categories by analyzing similarities between input examples. Some clustering use cases include:

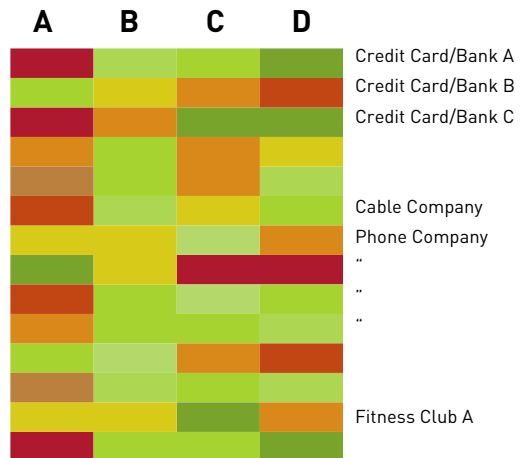
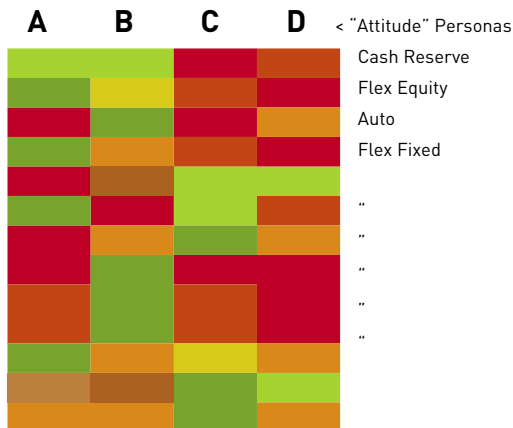
- Search results grouping
- Grouping similar customers
- Grouping similar patients
- Text categorization
- Network Security Anomaly detection (anomalies find what is not similar, which means the outliers from clusters)



The k -means algorithm groups observations into k clusters in which each observation belongs to the cluster with the nearest mean from its cluster center.



An example of clustering is a company that wants to segment its customers in order to better tailor products and offerings. Customers could be grouped on features such as demographics and purchase histories. Clustering with unsupervised learning is often combined with supervised learning in order to get more valuable results. For example, in this [banking customer 360](#) use case, customers were first clustered based on answers to a survey. The customer groups were analyzed and then labeled with customer personas. Next, the persona labels were linked by customer ID with customer features, such as types of accounts and purchases. Finally, supervised machine learning was applied and tested with the labeled customers, allowing it to link the survey customer personas with their banking actions and provide insights.



Frequent Pattern Mining, Association, Co-Occurrence, Market Basket Recommendations

Frequent pattern or association rule mining finds frequent co-occurring associations among a collection of items, such as products often purchased together. A famous story about association rule mining is the “beer and diaper” story. An analysis of behavior of grocery shoppers discovered that men who buy diapers often also buy beer.



Walmart mined their massive retail transaction database to see what their customers really wanted to buy prior to the arrival of a hurricane. They found one particular item which had an increase in sales by a factor of 7 over normal shopping days, a huge lift factor for a real-world case. The item was not bottled water, batteries, beer, flashlights, generators, or any of the usual things that you might imagine: it was [strawberry pop tarts](#)!



Another example is from Target, which analyzed that when a woman starts buying scent-free lotion, vitamin supplements, and a combination of some other items, it signals she could be pregnant. Unfortunately, Target sent a coupon for baby items to a teenager whose father questioned why she was receiving such coupons.



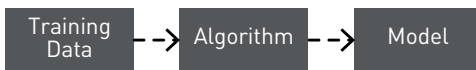
Co-occurrence analysis is useful for:

- Store layouts
- Determining which products to put on specials, promotions, coupons, etc.
- Identifying healthcare patients, like mine cohorts

Collaborative Filtering

Collaborative filtering algorithms recommend items (this is the filtering part) based on preference information from many users (this is the collaborative part). The collaborative filtering approach is based on similarity; people who liked similar items in the past will like similar items in the future. The goal of a collaborative filtering algorithm is to take preferences data from users and create a model that can be used for recommendations or predictions. Ted likes movies A, B, and C. Carol likes movies B and C. We take this data and run it through an algorithm to build a model. Then, when we have new data, such as Bob likes movie B, we use the model to predict that C is a possible recommendation for Bob.

Ted and Carol like movies B and C.



Bob likes movie B; what else might he like?



Bob likes movie B, so **predict movie C**.

User item rating matrix

	A	B	C
Ted	4	5	5
Carol		5	5
Bob		5	?

Spark Unsupervised Algorithms Summary

Clustering

- *k*-means
- Latent Dirichlet allocation (LDA)
- Gaussian mixture model (GMM)

Collaborative Filtering

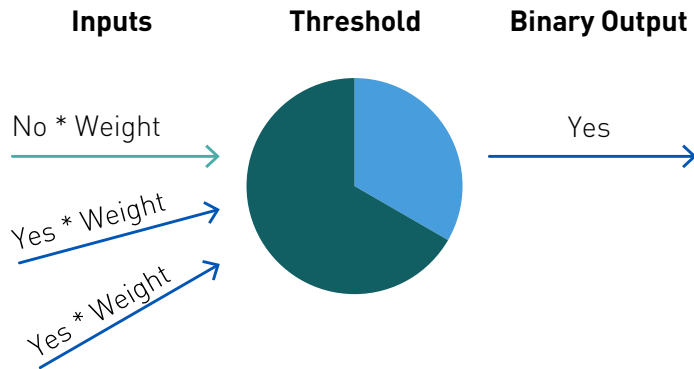
- Alternating least squares (ALS)

Frequent Pattern Mining

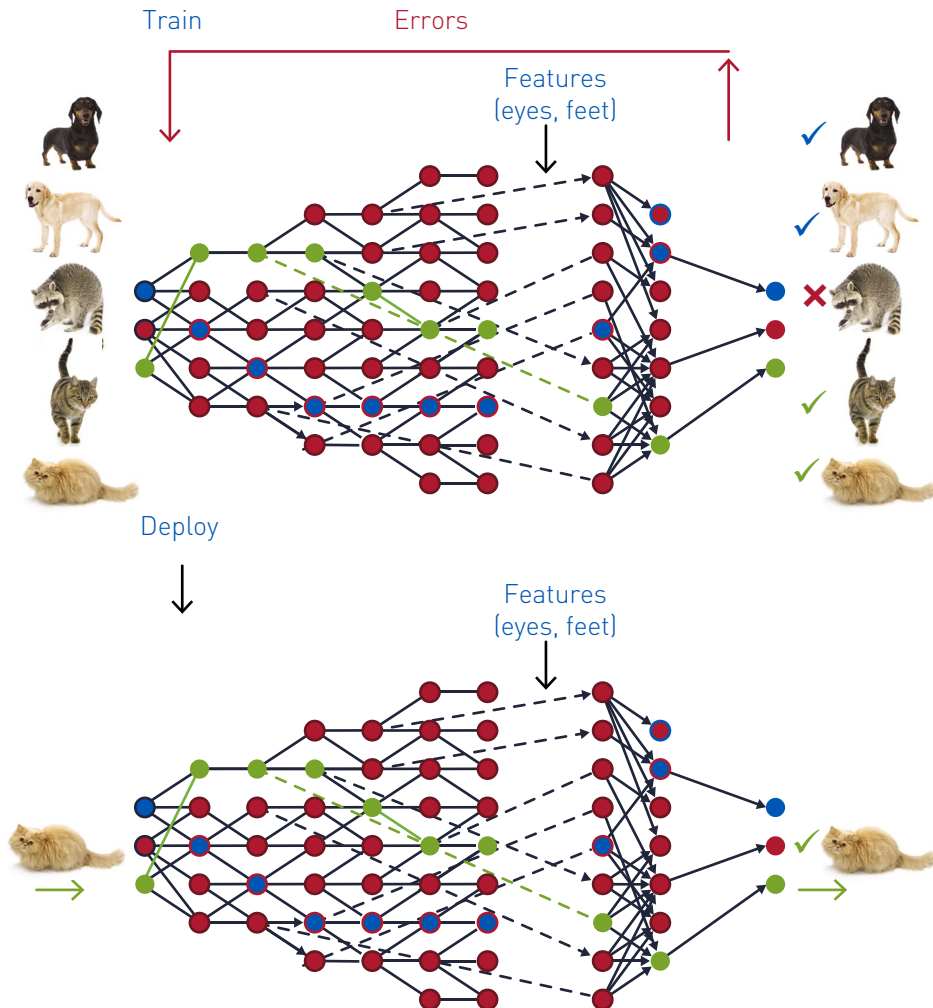
- FP-Growth Algorithm

Deep Learning

Deep learning is the name for multilayered neural networks, which are networks composed of several “hidden layers” of nodes between the input and output. There are many variations of neural networks, which you can learn more about on this [neural network cheat sheet](#). Improved algorithms, GPUs, and massively parallel processing (MPP) have given rise to networks with thousands of layers. Each node takes input data and a weight and outputs a confidence score to the nodes in the next layer, until the output layer is reached, where the error of the score is calculated.



With [backpropagation](#) inside of a process called [gradient descent](#), the errors are sent back through the network again and the weights are adjusted, improving the model. This process is repeated thousands of times, adjusting a model's weights in response to the error it produces, until the error can't be reduced any more.



During this process the layers learn the optimal features for the model, which has the advantage that features do not need to be predetermined. However, this has the disadvantage that the model's decisions are not explainable. Because explaining the decisions can be important, researchers are [developing new ways to understand the black box of deep learning](#).

There are different variations of deep learning algorithms, which can be used with the [Distributed Deep Learning Quick Start Solution from MapR](#) to build data-driven applications, such as the following:

Deep Neural Networks for improved traditional algorithms

- Finance: enhanced fraud detection through identification of more complex patterns
- Manufacturing: enhanced identification of defects, based on deeper anomaly detection

Convolutional Neural Networks for images

- Retail: in-store activity analysis of video to measure traffic
- Satellite images: labeling terrain, classifying objects
- Automotive: recognition of roadways and obstacles
- Healthcare: diagnostic opportunities from x-rays, scans, etc.
- Insurance: estimating claim severity, based on photographs

Recurrent Neural Networks for sequenced data

- Customer satisfaction: transcription of voice data to text for NLP analysis
- Social media: real-time translation of social and product forum posts
- Photo captioning: search archives of images for new insights
- Finance: Predicting behavior based on time series analysis (also enhanced recommendation systems)

Deep Learning with Spark

Deep learning libraries or frameworks that can be leveraged with Spark include:

BigDL	TensorFlowOnSpark	PyTorch
Spark Deep Learning	dist-keras	Caffe
Pipelines	H2O Sparkling Water	MXNet

Summary

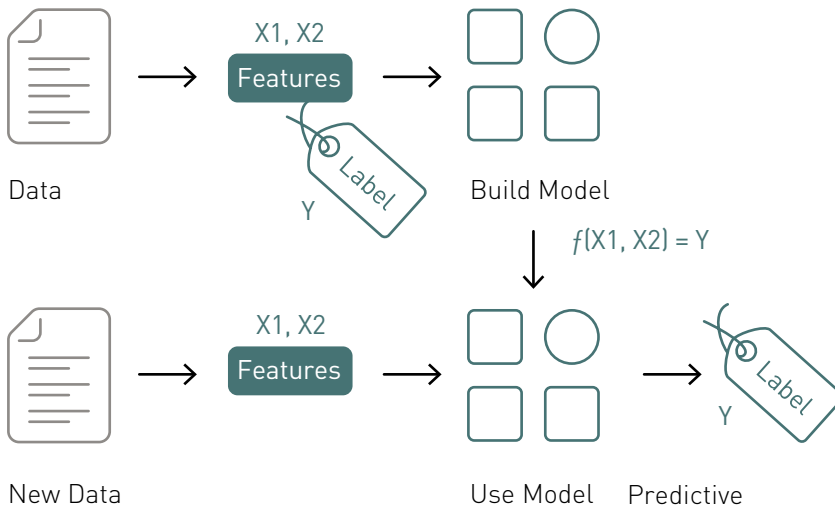
A confluence of several different technology shifts have dramatically changed machine learning applications. The combination of distributed computing, streaming analytics, and machine learning is accelerating the development of [next-generation intelligent applications](#), which are taking advantage of modern computational paradigms, powered by modern computational infrastructure. The MapR Data Platform integrates global event streaming, real-time database capabilities, and scalable enterprise storage with a collection of data processing and analytical engines to power this new generation of data processing pipelines and intelligent applications.

Predicting Flight Delays Using Apache Spark Machine Learning

Because flight delays create problems in scheduling, passenger inconvenience, and economic losses, there is growing interest in predicting flight delays beforehand in order to optimize operations and improve customer satisfaction. [Google Flights](#) uses historic flight status data with machine learning algorithms to find common patterns in late departures in order to predict flight delays and share the reasons for those delays. In this chapter, we will go over using Apache Spark's [ML pipelines](#) with a [Random Forest Classifier](#) to predict flight delays.

Classification

Classification is a family of supervised machine learning algorithms that identify which category an item belongs to, based on labeled examples of known items. Classification takes a set of data with known labels and pre-determined features and learns how to label new records, based on that information. Features are the “if questions” or properties that you can use to make predictions. To build a classifier model, you explore and extract the features that most contribute to the classification.



Let's go through an example for flight delays:

What are we trying to predict?

- Whether a flight will be delayed or not.
- Delayed is the Label: True or False

What are the “if questions” or properties that you can use to make predictions?

- What is the originating airport?
- What is the destination airport?
- What is the scheduled time of departure?
- What is the scheduled time of arrival?
- What is the day of the week?
- What is the airline carrier?

Decision Trees

Decision trees create a model that predicts the label (or class) 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. Below is an example of a simplified decision tree for flight delays:

Q1: If the scheduled departure time is < 10:15 AM

TQ2: If the originating airport is in the set {ORD, ATL, SFO}

T:Q3: If the day of the week is in the set {Monday, Sunday}

- T: Delayed=1
- F: Delayed=0

F: Q3: If the destination airport is in the set {SFO, ORD, EWR}

- T: Delayed=1
- F: Delayed=0

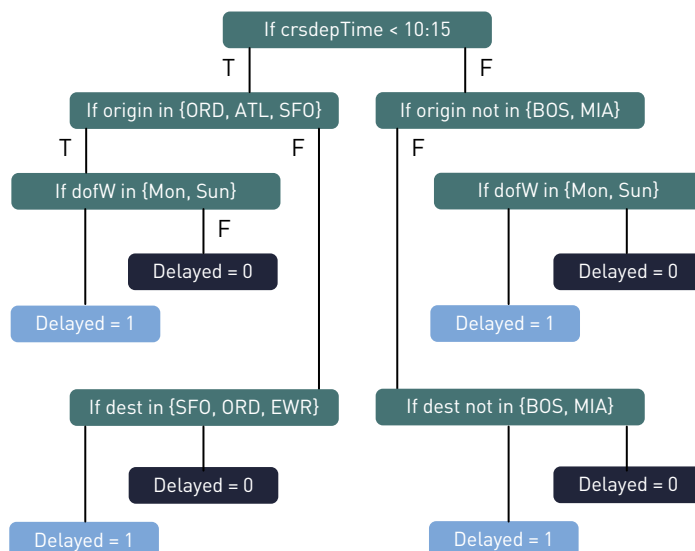
F: Q2: If the originating airport is not in the set {BOS, MIA}

T:Q3: If the day of the week is in the set {Monday, Sunday}

- T: Delayed=1
- F: Delayed=0

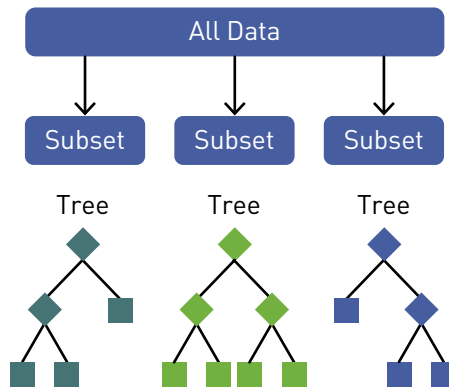
F: Q3: If the destination airport is not in the set {BOS, MIA}

- T: Delayed=1
- F: Delayed=0



Random Forests

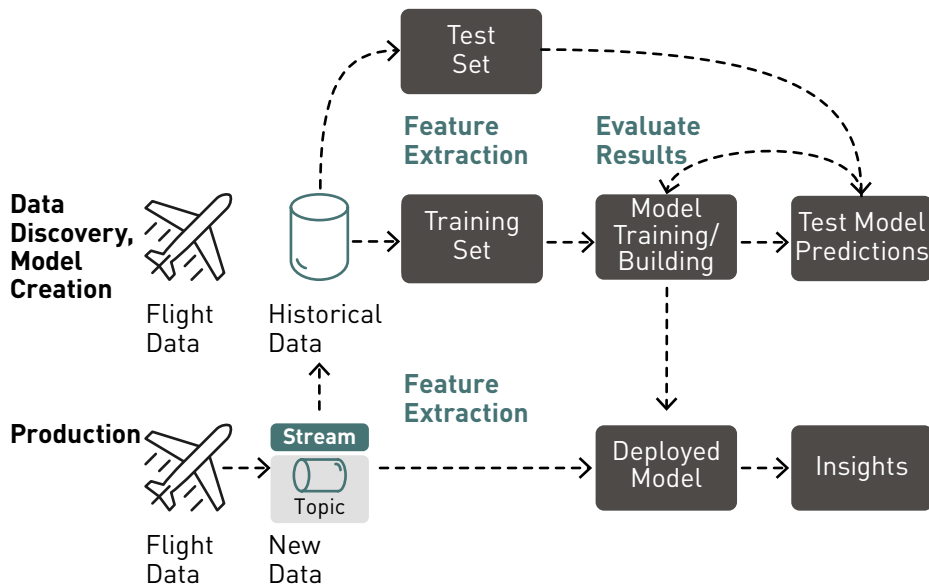
Ensemble learning algorithms combine multiple machine learning algorithms to obtain a better model. Random Forest is a popular ensemble learning method for classification and regression. The algorithm builds a model consisting of multiple decision trees, based on different subsets of data at the training stage. Predictions are made by combining the output from all of the trees, which reduces the variance and improves the predictive accuracy. For Random Forest classification, each tree's prediction is counted as a vote for one class. The label is predicted to be the class which receives the most votes.



Typical Machine Learning Workflow

Using machine learning in order to better understand your data and make predictions is an iterative process, which involves:

1. Data discovery and model creation:
 - Analysis of historical data
 - Identifying new data sources, which traditional analytics or databases are not using, due to the format, size, or structure
 - Collecting, correlating, and analyzing data across multiple data sources
 - Knowing and applying the right kind of machine learning algorithms to get value out of the data
 - Training, testing, and evaluating the results of machine learning algorithms to build a model.
2. Using the model in production to make predictions
3. Data discovery and updating the model with new data



Data Exploration and Feature Extraction

We will be using the Flight Data set that we explored in [chapter 2](#). To build a classifier model, you extract the features that most contribute to the classification. In this scenario, we will build a tree to predict the label of delayed or not, based on the following features:

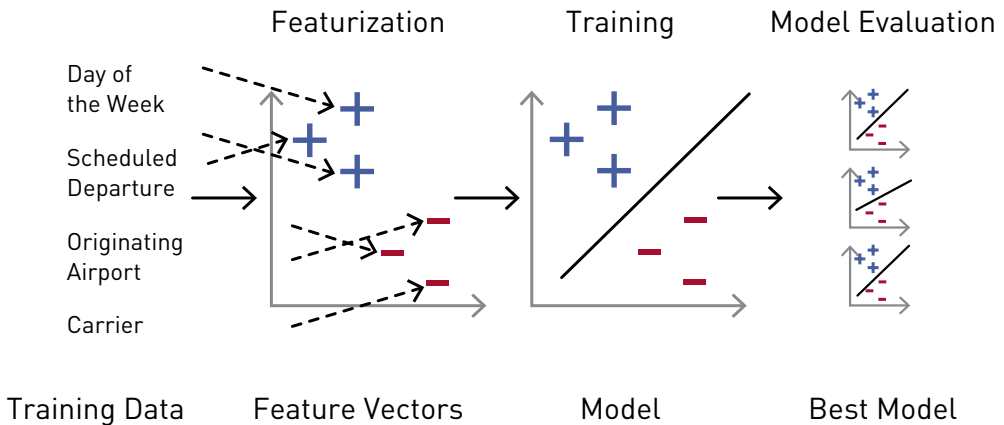
Label → delayed = 0

- Delayed = 1 if delay > 40 minutes

Features → {day of the week, scheduled departure time, scheduled arrival time, carrier, scheduled elapsed time, origin, destination, distance}

delayed	dofW	crsdepTime	crsArrTime	carrier	elapTime	origin	dest	dist
1.0/0.0	1	1015	1230	AA	385.0	JFK	LAX	2475.0

In order for the features to be used by a machine learning algorithm, they must be transformed and put into feature vectors, which are vectors of numbers representing the value for each feature.

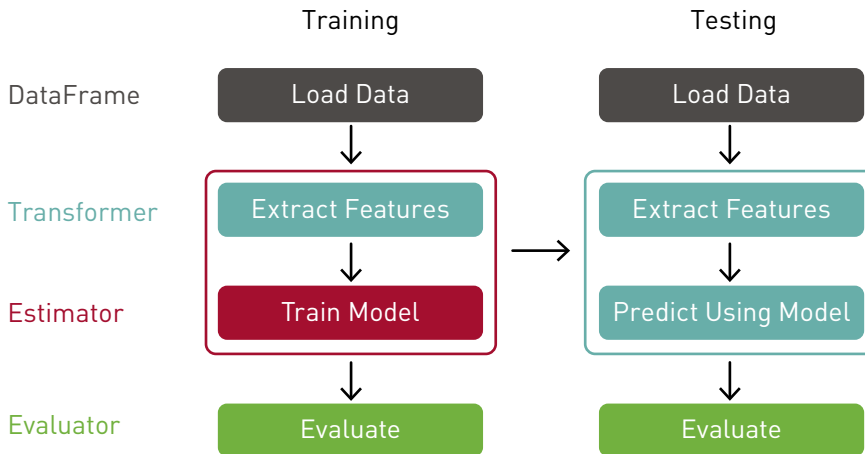


Reference: Learning Spark

Using The Spark ML Package

Spark ML provides a uniform set of high-level APIs, built on top of DataFrames with the goal of making machine learning scalable and easy. Having ML APIs built on top of DataFrames provides the scalability of partitioned data processing with the ease of SQL for data manipulation.

We will use an ML Pipeline to pass the data through transformers in order to extract the



features and an estimator to produce the model.

Transformer: A transformer is an algorithm that transforms one DataFrame into another DataFrame. We will use transformers to get 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 will use a an estimator to train a model, which can transform input data to get predictions.

Pipeline: A pipeline chains multiple transformers and estimators together to specify an ML workflow.

Load the Data from a File into a DataFrame

Load Data → DataFrame

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|_id|dofW|carrier|origin|dest|crsdephour|crsdeptime|depdelay|crsarrrtime|arrdelay|crselapsedtime|dist|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|AA_2017-01-01_ATL...|7|AA|ATL|LGA|17|1700.0|0.0|1912.0|0.0|132.0|762.0|
|AA_2017-01-01_LGA...|7|AA|LGA|ATL|13|1343.0|0.0|1620.0|0.0|157.0|762.0|
|AA_2017-01-01_MIA...|7|AA|MIA|ATL|9|939.0|0.0|1137.0|10.0|118.0|594.0|
```

The first step is to load our data into a DataFrame, like we did in [chapter 2](#). We use a Scala case class and StructType to define the schema, corresponding to a line in the JSON data file. Below, [we specify the data source, schema, and class to load into a Dataset](#). We load the data from January and February, which we will use for training and testing the model. (Note that specifying the schema when loading data into a DataFrame will give better performance than schema inference.)

```

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.classification._
import org.apache.spark.ml.evaluation._
import org.apache.spark.ml.tuning._
val schema = StructType(Array(
    StructField("_id", StringType, true),
    StructField("dofW", IntegerType, true),
    StructField("carrier", StringType, true),
    StructField("origin", StringType, true),
    StructField("dest", StringType, true),
    StructField("crsdephour", IntegerType, true),
    StructField("crsdeptime", DoubleType, true),
    StructField("depdelay", DoubleType, true),
    StructField("crsarrrtime", DoubleType, true),
    StructField("arrdelay", DoubleType, true),
    StructField("crselapsedtime", DoubleType, true),
    StructField("dist", DoubleType, true)
))

case class Flight(_id: String, dofW: Integer, carrier: String,
origin: String, dest: String, crsdephour: Integer, crsdeptime:
Double, depdelay: Double, crsarrrtime: Double, arrdelay: Double,
crselapsedtime: Double, dist: Double) extends Serializable

var file = "/path/flights20170102.json"

val df = spark.read.format("json").option("inferSchema", "false").
schema(schema).load(file).as[Flight]

df.createOrReplaceTempView("flights")

```

The DataFrame show method displays the first 20 rows or the specified number of rows:

```
scala> df.show(3)
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|_id|dofw|carrier|origin|dest|crsdephour|crsdeptime|depdelay|crsarrrtime|arrdelay|crselapsedtime| dist|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|ATL_BOS_2017-01-01..| 7| DL| ATL| BOS| 9| 859.0| 30.0| 1127.0| 11.0| 148.0|946.0|
|ATL_BOS_2017-01-01..| 7| DL| ATL| BOS| 11| 1141.0| 0.0| 1409.0| 0.0| 148.0|946.0|
|ATL_BOS_2017-01-01..| 7| WN| ATL| BOS| 13| 1335.0| 0.0| 1600.0| 0.0| 145.0|946.0|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

In the code below, using the DataFrame with column transformation, we add a column “orig_dest” for the origination->destination, in order to use this as a feature. Then we query to get the count of departure delays by origin_destination. The routes ORD->SFO and DEN->SFO have the highest delays, possibly because of weather in January and February. Adding weather to this Dataset would give better results.

```
import org.apache.spark.sql.functions.{concat, lit}

val df1 = df.withColumn("orig_dest", concat($"origin",lit("_"),
$"dest"))

df1.select($"orig_dest", $"depdelay")
  .filter($"depdelay" > 40)
  .groupBy("orig_dest")
  .count
  .orderBy(desc("count" )) .show(5)
```

result:

```
+-----+-----+
|orig_dest|count|
+-----+-----+
| DEN_SFO| 172|
| ORD_SFO| 168|
| ATL_LGA| 155|
| ATL_EWR| 141|
| SFO_DEN| 134|
+-----+-----+

```

Summary Statistics

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

```
df.describe("dist", "depdelay", "arrdelay", "crselapsedtime").show
```

result:

summary	depdelay	arrdelay	crselapsedtime
count	41348	41348	41348
mean	15.018719164167553	14.806907226468027	186.26264873754474
stddev	44.529632044361385	44.223705132666396	68.38149648990024
min	0.0	0.0	64.0
max	1440.0	1442.0	423.0

In the code below, a Spark [Bucketizer](#) is used to split the Dataset into delayed and not delayed flights with a delayed 0/1 column. Then the resulting total counts are displayed. Grouping the data by the delayed field and counting the number of instances in each group shows that there are roughly 8 times as many not delayed samples as delayed samples.

```
val delaybucketizer = new Bucketizer().setInputCol("depdelay")
    .setOutputCol("delayed").setSplits(Array( 0.0, 15.0 , Double.
PositiveInfinity))
```

```
val df2 = delaybucketizer.transform(df1)
```

```
df2.createOrReplaceTempView("flights")
```

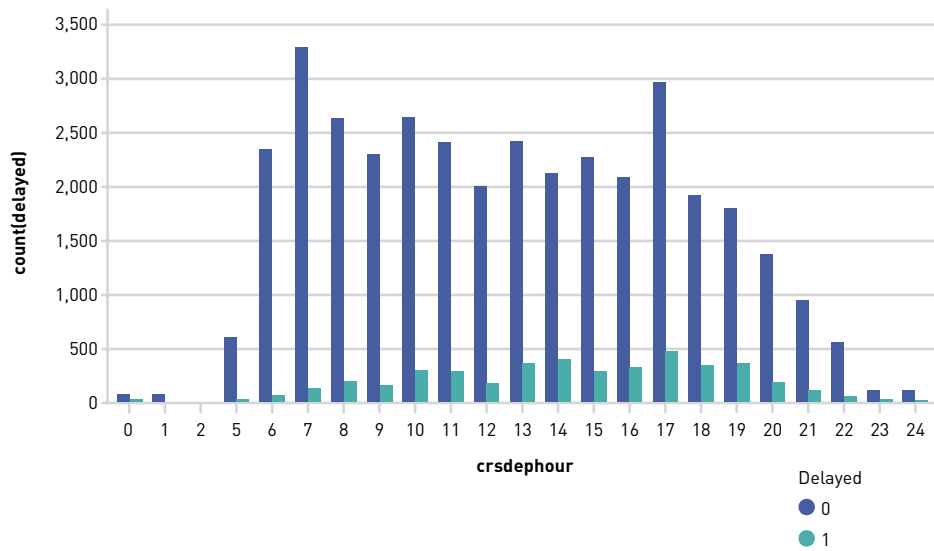
```
df2.groupBy("delayed").count().show
```

result:

delayed	count
0.0	36790
1.0	4558

In the query below we see the count of not delayed (0=dark blue) and delayed (1= turquoise) flights by departure hour.

```
%sql select crsdephour, delayed, count(delayed) from flights group by crsdephour, delayed order by crsdephour
```



Stratified Sampling

In order to ensure that our model is sensitive to the delayed samples, we can put the two sample types on the same footing using stratified sampling. The DataFrames `sampleBy()` function does this when provided with fractions of each sample type to be returned. Here, we're keeping all instances of delayed, but downsampling the not delayed instances to 13%, then displaying the results.

```
val fractions = Map(0.0 -> .13, 1.0 -> 1.0)
val strain = df2.stat.sampleBy("delayed", fractions, 36L)
val Array(trainingData, testData) = strain
    .randomSplit(Array(0.7, 0.3), 5043)

strain.groupBy("delayed").count.show
```

result:

```
+-----+-----+
|delayed|count|
+-----+-----+
|      0.0| 4766|
|      1.0| 4558|
+-----+-----+
```

Feature Extraction and Pipelining

The ML package needs the label and feature vector to be added as columns to the input DataFrame. We set up a pipeline to pass the data through transformers in order to extract the features and label. We will use Spark `StringIndexers` in the pipeline to encode a column of string values to a column of number indices for those values (the indices are proportional to the occurrence of the values in the dataset). An example of `StringIndexing`, encoding a column of string values to a column of number indices for carrier, is shown below:

```
+-----+-----+
|carrier|carrierIndexed|
+-----+-----+
|      UA|           0.0|
|      DL|           1.0|
|      WN|           3.0|
|      AA|           2.0|
+-----+-----+
```

The code below sets up StringIndexers for all of the categorical columns. Later, we will put these StringIndexers in the pipeline.

```
// column names for string types
val categoricalColumns = Array("carrier", "origin", "dest", "dofW",
"orig_dest")

// used to encode string columns to number indices
// Indices are fit to dataset
val stringIndexers = categoricalColumns.map { colName =>
    new StringIndexer()
        .setInputCol(colName)
        .setOutputCol(colName + "Indexed")
        .fit(strain)
}
```

A Bucketizer will be used in the pipeline to add a label of delayed 0/1, with 0 for delays less than 40 minutes and 1 for delays greater than 40 minutes.

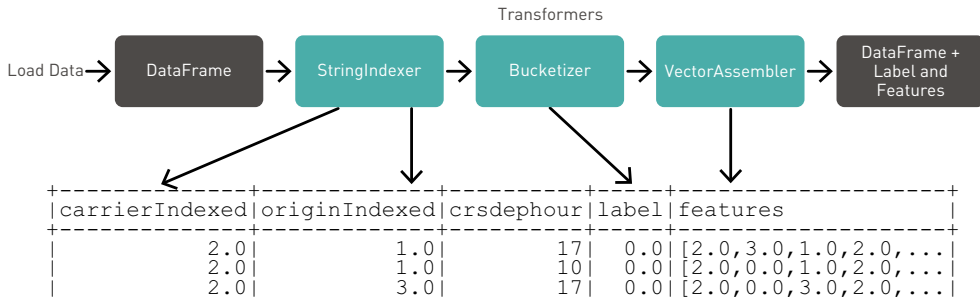
```
// add a label column based on departure delay
val labeler = new Bucketizer().setInputCol("depdelay")
    .setOutputCol("label")
    .setSplits(Array(0.0, 40.0, Double.PositiveInfinity))
```

The VectorAssembler is used in the pipeline to combine a given list of columns into a single feature vector column.

```
// list of feature columns
val featureCols = Array("carrierIndexed", "destIndexed",
"originIndexed","dofWIndexed","orig_destIndexed",
"crsdephour", "crsdeptime", "crsarrrtime",
"crselapsedtime","dist" )

// combines a list of feature columns into a vector column
val assembler = new VectorAssembler()
    .setInputCols(featureCols)
    .setOutputCol("features")
```


The result of running these transformers in a pipeline will be to add a label and features column to the dataset as shown below.

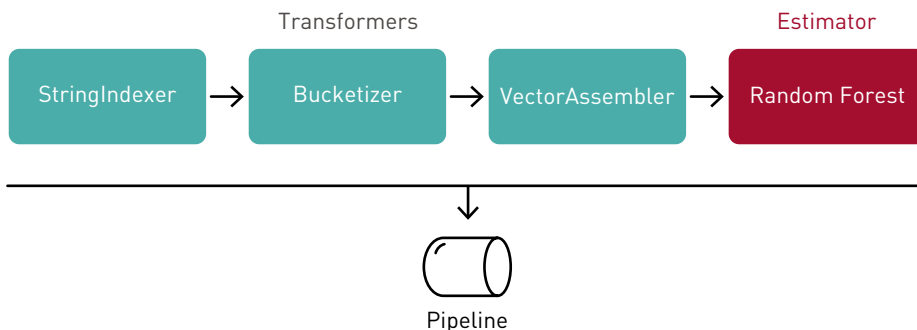


The final element in our pipeline is an estimator (a Random Forest Classifier), which will train on the vector of labels and features and return a (transformer) model.

```
val rf = new RandomForestClassifier()
    .setLabelCol("label")
    .setFeaturesCol("features")
)
```

Below, we put the StringIndexers, VectorAssembler, and Random Forest Classifier in an pipeline. A pipeline chains multiple transformers and estimators together to specify an ML workflow for training and using a model.

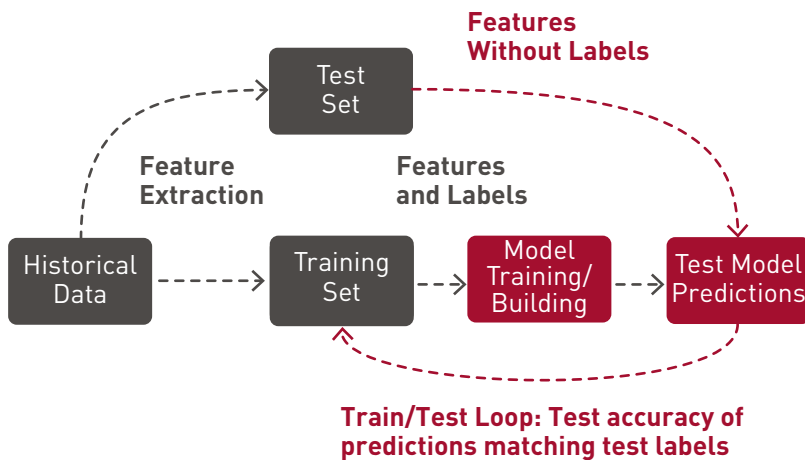
```
val steps = stringIndexers ++ Array(labeler, assembler, rf)
val pipeline = new Pipeline().setStages(steps)
```



Train The Model

We would like to determine which parameter values of the Random Forest Classifier produce the best model. A common technique for model selection is k -fold cross-validation, where the data is randomly split into k partitions. Each partition is used once as the testing data set, 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 performance measurements. The model parameters leading to the highest performance metric produce the best model.

ML Cross-Validation Process



Spark ML supports k -fold cross-validation with a transformation/estimation pipeline to try out different combinations of parameters, using a process called grid search, where you set up the parameters to test and a cross validation evaluator to construct a model selection workflow.

Below, we use a `ParamGridBuilder` to construct the parameter grid for the model training. We define an evaluator, which will evaluate the model by comparing the test label column with the test prediction column. We use a `CrossValidator` for model selection. The `CrossValidator` uses the pipeline, the parameter grid, and the classification evaluator to fit the training data set and returns a model.

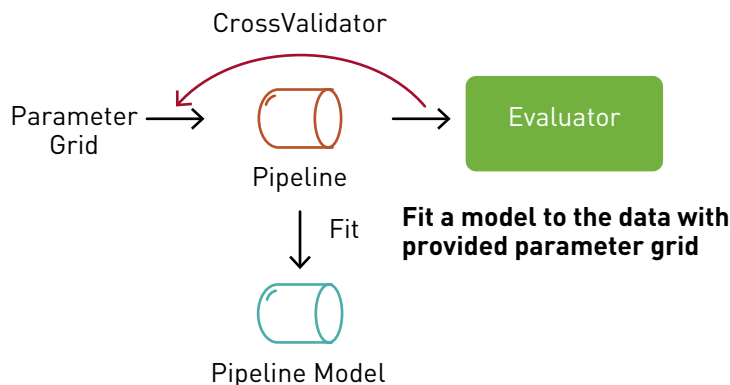
```
val paramGrid = new ParamGridBuilder()
  .addGrid(rf.maxBins, Array(100, 200))
  .addGrid(rf.maxDepth, Array(2, 4, 10))
  .addGrid(rf.numTrees, Array(5, 20))
  .addGrid(rf.impurity, Array("entropy", "gini"))
  .build()

val evaluator = new BinaryClassificationEvaluator()

// Set up 3-fold cross validation with paramGrid
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)
```

The `CrossValidator` uses the `ParamGridBuilder` to iterate through the `maxDepth`, `maxBins`, and `numTrees` parameters of the Random Forest Classifier and to evaluate the models, repeating 3 times per parameter value for reliable results.



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

Next, we can get the best model, in order to print out the feature importances. The results show that the scheduled departure time and the orig->dest are the most important features.

```
val featureImportances = pipelineModel
    .bestModel.asInstanceOf[PipelineModel]
    .stages(stringIndexers.size + 2)
    .asInstanceOf[RandomForestClassificationModel]
    .featureImportances

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

result:

```
feature: crsdeptime, importance: 0.2954321019748874
feature: orig_destIndexed, importance: 0.21540676913162476
feature: crsarrrtime, importance: 0.1594826730807351
feature: crsdephour, importance: 0.11232750835024508
feature: destIndexed, importance: 0.07068851952515658
feature: carrierIndexed, importance: 0.03737067561393635
feature: dist, importance: 0.03675114205144413
feature: dofWIndexed, importance: 0.030118527912782744
feature: originIndexed, importance: 0.022401521272697823
feature: crselapsedtime, importance: 0.020020561086490113
```

We find that the best random forest model produced, using the cross-validation process, is one with a depth of 4, 20 trees and 100 bins.

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

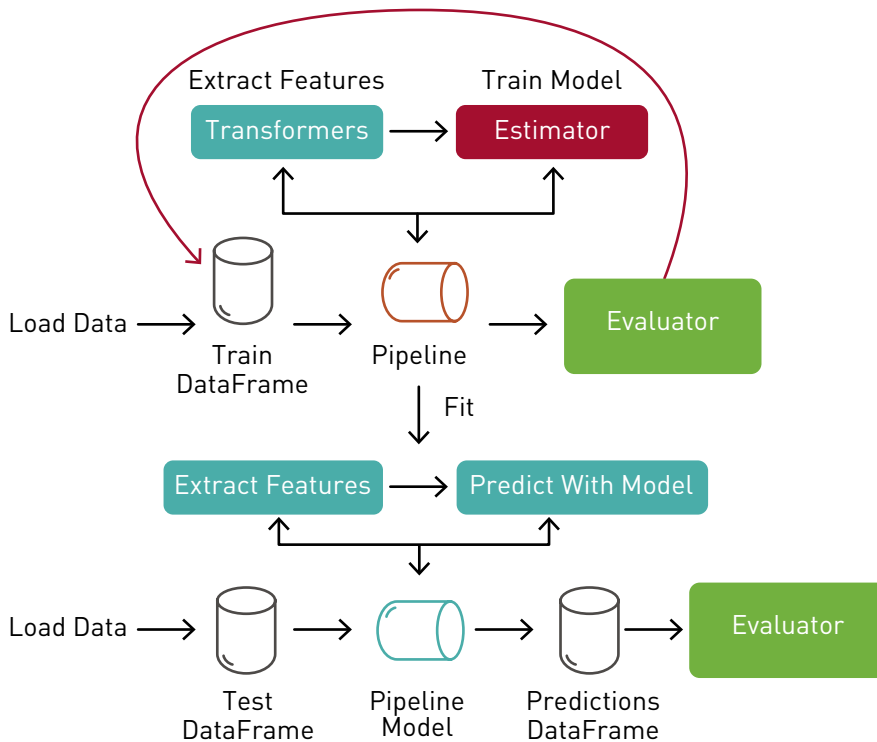
result:

```
Best params: {
rfc-impurity: gini,
rfc-maxBins: 100,
rfc-maxDepth: 4,
rfc-numTrees: 20 }
```

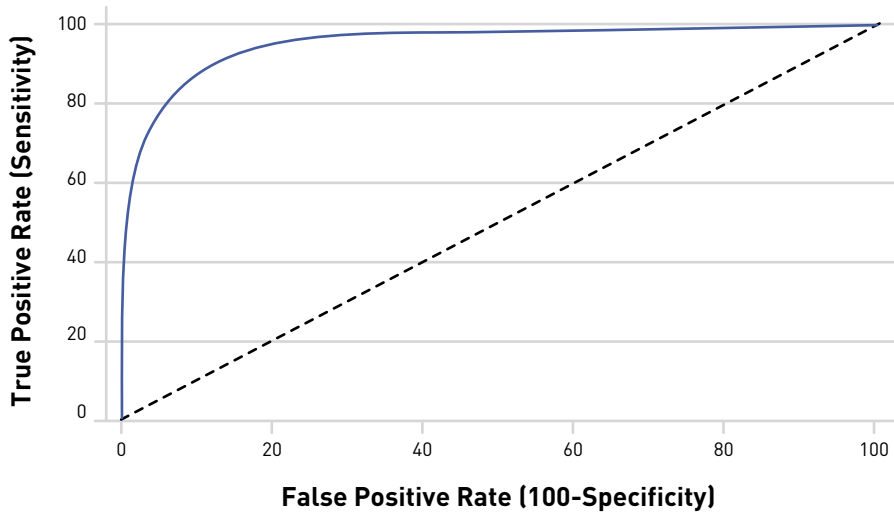
Predictions and Model Evaluation

The performance of the model can be determined using the test data set that has not been used for any training or cross-validation activities. We transform the test DataFrame with the pipeline model, which will pass the test data, according to the pipeline steps, through the feature extraction stage, estimate with the random forest model chosen by model tuning, and then return the label predictions in a column of a new DataFrame.

```
val predictions = pipelineModel.transform(testData)
```



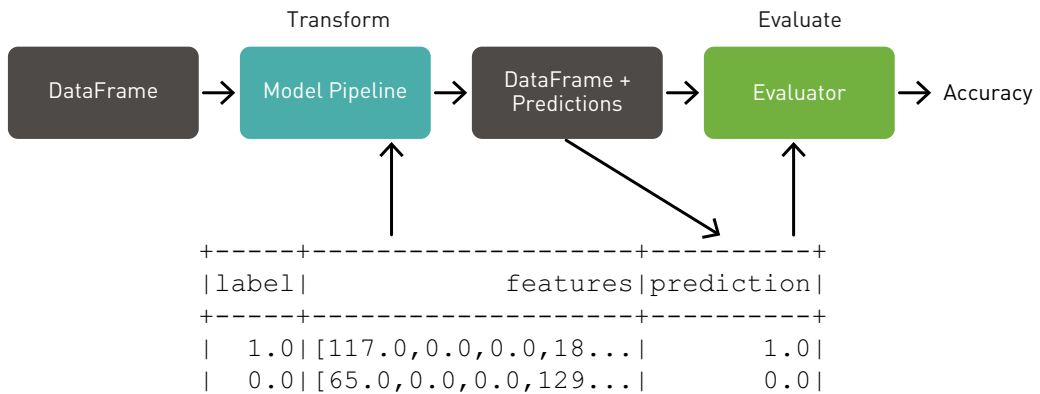
The `BinaryClassificationEvaluator` provides a metric to measure how well a fitted model does on the test data. The default metric for this evaluator is the area under the ROC curve. The area measures the ability of the test to correctly classify true positives from false positives. A random predictor would have .5. The closer the value is to 1, the better its predictions are.



Below, we pass the predictions DataFrame (which has a predictions column and a label column) to the BinaryClassificationEvaluator, which returns .69 as the area under the ROC curve. We could get better flight delay predictions with more data sources, such as weather, holidays, incoming flight information, and current or incoming airport operations problems.

```
val areaUnderROC = evaluator.evaluate(predictions)
```

```
result: 0.69
```



```
val predictions = cvModel.transform(test)  
val accuracy = evaluator.evaluate(predictions)
```

Below, we calculate some more metrics. The number of false/true positives and negative predictions is also useful:

- True positives are how often the model correctly predicted delayed flights.
- False positives are how often the model incorrectly predicted delayed flights.
- True negatives indicate how often the model correctly predicted not delayed flights.
- False negatives indicate how often the model incorrectly predicted not delayed flights.

```
val lp = predictions.select("label", "prediction")
val counttotal = predictions.count()
val correct = lp.filter($"label" === $"prediction").count()
val wrong = lp.filter(not($"label" === $"prediction")).count()
val ratioWrong = wrong.toDouble / counttotal.toDouble
val ratioCorrect = correct.toDouble / counttotal.toDouble

val truep = lp.filter($"prediction" === 0.0)
  .filter($"label" === $"prediction").count() /
  counttotal.toDouble

val truen = lp.filter($"prediction" === 1.0)
  .filter($"label" === $"prediction").count() /
  counttotal.toDouble

val falsep = lp.filter($"prediction" === 0.0)
  .filter(not($"label" === $"prediction")).count() /
  counttotal.toDouble

val falsen = lp.filter($"prediction" === 1.0)
  .filter(not($"label" === $"prediction")).count() /
  counttotal.toDouble
```

Results:

```
counttotal: Long = 2744
correct: Long = 1736
wrong: Long = 1008
ratioWrong: Double = 0.3673469387755102
ratioCorrect: Double = 0.6326530612244898
truep: Double = 0.3079446064139942
truen: Double = 0.32470845481049565
falsep: Double = 0.15998542274052477
falsen: Double = 0.20736151603498543
```


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

There are plenty of great tools to build classification models. Apache Spark provides an excellent framework for building solutions to business problems that can extract value from massive, distributed datasets.

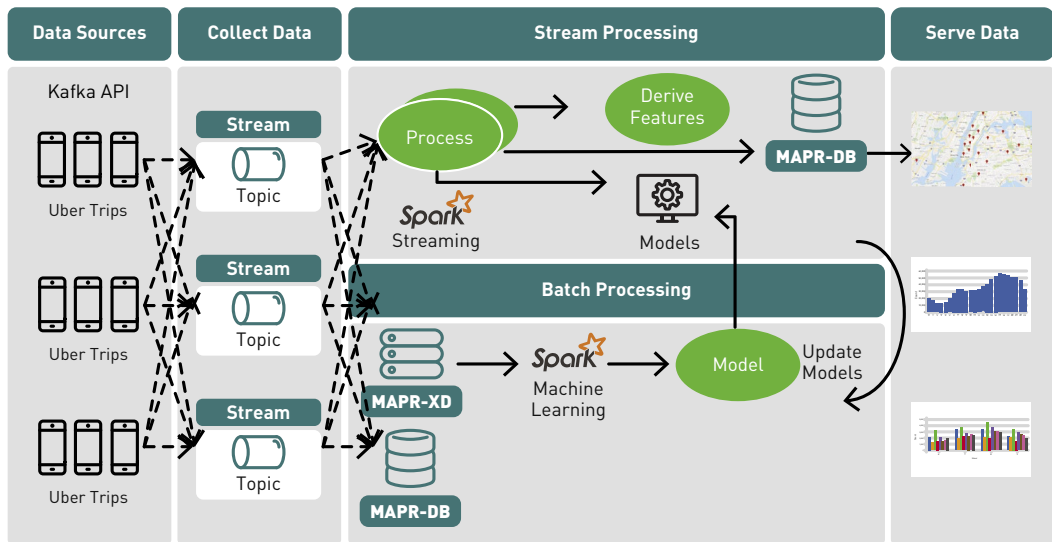
Machine learning algorithms cannot answer all questions perfectly. But they do provide evidence for humans to consider when interpreting results, assuming the right question is asked in the first place. In this example, we could get better flight delay predictions with more timely information, such as weather, holidays, incoming flight delays, and airport problems.

All of the data and code to train the models and make your own conclusions, using Apache Spark, are located in GitHub. Refer to the Appendix for the links to the GitHub and more information about running the code.

Cluster Analysis on Uber Event Data to Detect and Visualize Popular Uber Locations

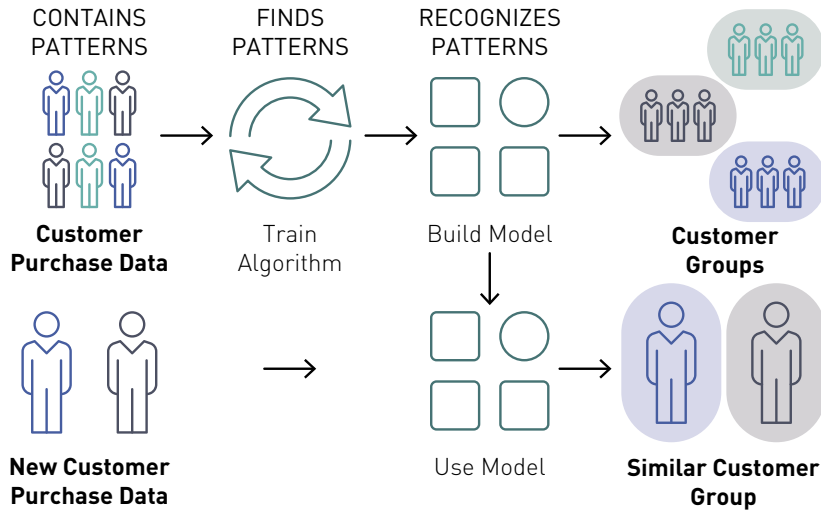
According to Bernard Marr, [one of the 10 major areas](#) in which big data is being used to excellent advantage is in improving cities. The analysis of location and behavior patterns within cities allows optimization of traffic, better planning decisions, and smarter advertising. For example, the analysis of GPS car data can allow 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 a large metropolitan area. The application of machine learning to geolocation data is being used in telecom, travel, marketing, and manufacturing to identify patterns and trends, for services such as recommendations, anomaly detection, and fraud.

Uber is using [Apache Spark and big data to perfect its processes](#), from calculating Uber's "surge pricing" to finding the optimal positioning of cars to maximize profits. In this chapter, we are going to use public Uber trip data to discuss cluster analysis on Uber event data to detect and visualize popular Uber locations. We start with a review of clustering and the k -means algorithm and then explore the use case. In the next chapter, we will use the saved k -means model with streaming data. (Note the code in this example is not from Uber, only the data.)

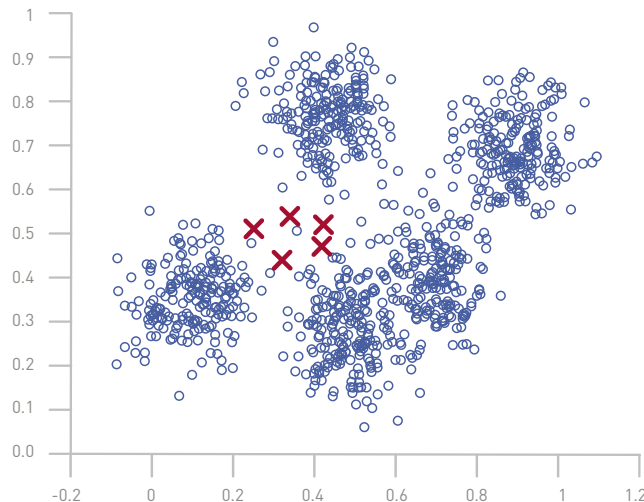


Clustering

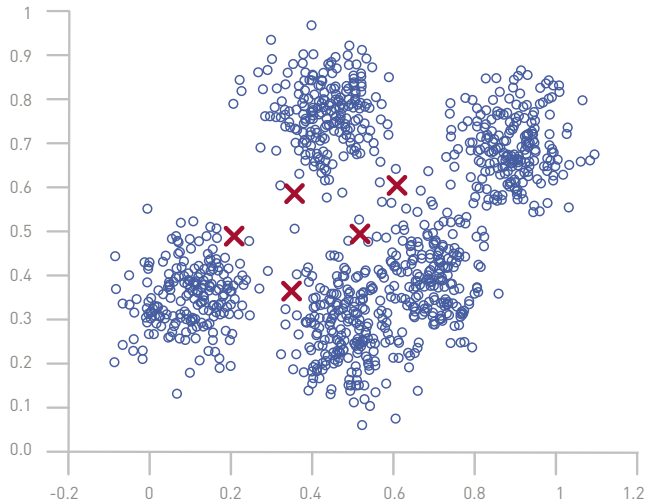
Clustering is a family of unsupervised machine learning algorithms that discover groupings that occur in collections of data by analyzing similarities between input examples. Some examples of clustering uses include customer segmentation and text categorization.



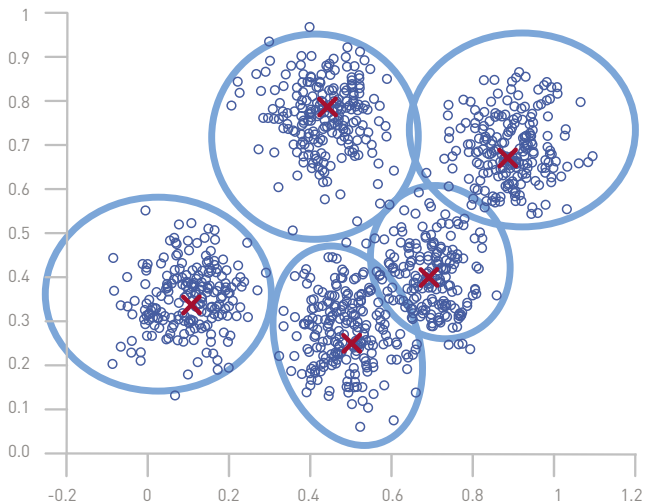
K-means is one of the most commonly used clustering algorithms that clusters the data points into a predefined number of clusters (k). Clustering using the k -means algorithm begins by initializing all the coordinates to k number of centroids.



With every pass of the algorithm, each point is assigned to its nearest centroid, based on some distance metric, usually Euclidean distance. The centroids are then updated to be the “centers” of all the points assigned to it in that pass.



This repeats until there is a minimum change in the centers.



Example Use Case Dataset

The example Dataset is [Uber trip data](#), which [FiveThirtyEight](#) obtained from the [NYC Taxi & Limousine Commission](#). In this example, we will discover the clusters of Uber data based on the longitude and latitude, then we will analyze the cluster centers by date/time, using Spark SQL. The Dataset has the following schema:

Field	Definition
Date/Time	The date and time of the Uber pickup
Lat	The latitude of the Uber pickup
Lon	The longitude of the Uber pickup
Base	The TLC base company code affiliated with the Uber pickup

The Data Records are in CSV format; an example line is shown below:

```
2014-08-01 00:00:00,40.729,-73.9422,B02598
```

Load the Data from a File into a DataFrame

First, we import the packages needed for Spark ML clustering and SQL.

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.types._
import org.apache.spark.sql._
import org.apache.spark.ml.clustering._
import org.apache.spark.ml._
import org.apache.spark.ml.feature._
```

We specify the schema with a Spark [StructType](#) and a Scala case class.

```
case class Uber(dt: java.sql.Timestamp, lat: Double,
  lon: Double, base: String) extends Serializable

val schema = StructType(Array(
  StructField("dt", TimestampType, true),
  StructField("lat", DoubleType, true),
  StructField("lon", DoubleType, true),
  StructField("base", StringType, true)
))
```

Next we load the data from a CSV file into a Spark DataFrame, specifying the datasource and schema to load into the DataFrame, as shown below. (Note: if you are using a notebook, then you do not have to create the SparkSession.)

Load Data → DataFrame

```
+-----+-----+-----+-----+
|          dt|      lat|      lon|   base|
+-----+-----+-----+-----+
|2014-08-01 07:00:...| 40.729|-73.9422|B02598|
|2014-08-01 07:00:...|40.7476|-73.9871|B02598|
|2014-08-01 07:00:...|40.7424|-74.0044|B02598|
|2014-08-01 07:00:...| 40.751|-73.9869|B02598|
|2014-08-01 07:00:...|40.7406|-73.9902|B02598|
+-----+-----+-----+-----+
```

```
val spark: SparkSession = SparkSession.builder()
  .appName("uber").getOrCreate()

import spark.implicits._

// path to dataset file
var file: String = "/mapr/demo.mapr.com/data/uber.csv"

val df: Dataset[Uber] = spark.read
  .option("inferSchema", "false")
  .schema(schema)
  .csv(file).as[Uber]
```

DataFrame `printSchema()` prints the schema to the console in a tree format, shown below:

```
df.printSchema

result:
root
 |-- dt: timestamp (nullable = true)
 |-- lat: double (nullable = true)
 |-- lon: double (nullable = true)
 |-- base: string (nullable = true)
```

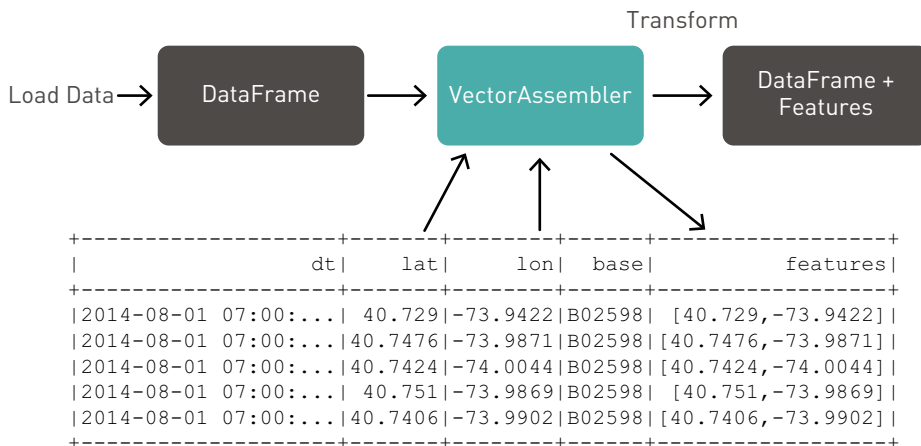
DataFrame show(5) displays the first 5 rows:

```
df.show(5)

result:
+-----+-----+-----+-----+
|          dt|      lat|      lon|   base|
+-----+-----+-----+-----+
|2014-08-01 00:00:00| 40.729|-73.9422|B02598|
|2014-08-01 00:00:00|40.7476|-73.9871|B02598|
|2014-08-01 00:00:00|40.7424|-74.0044|B02598|
|2014-08-01 00:00:00| 40.751|-73.9869|B02598|
|2014-08-01 00:00:00|40.7406|-73.9902|B02598|
+-----+-----+-----+-----+
```

Define Features Array

In order for the features to be used by a machine learning 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 the input columns lat, lon in a vector features column. The df2 DataFrame with the features column is cached, since it will be used iteratively by the *k*-means estimator to create a model.




```
// input column names
val featureCols = Array("lat", "lon")

// create transformer
val assembler = new VectorAssembler()
  .setInputCols(featureCols)
  .setOutputCol("features")

// transform method adds features column
val df2 = assembler.transform(df)

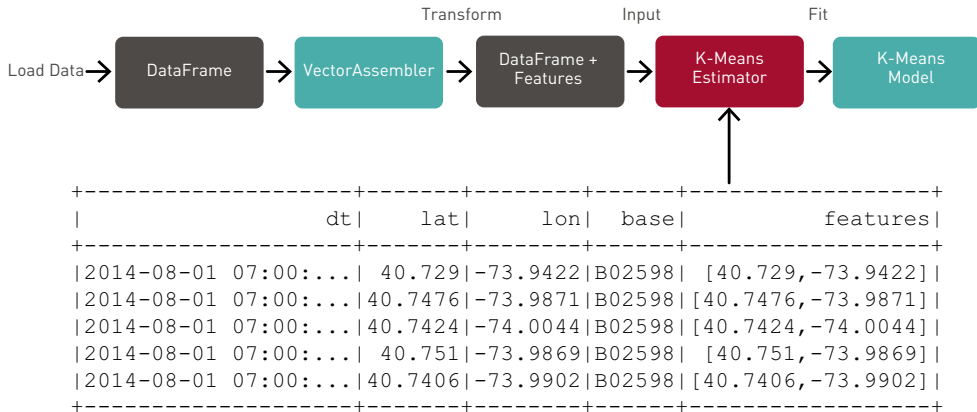
// cache transformed DataFrame
df2.cache

df2.show(5)
```

result:

dt	lat	lon	base	features
2014-08-01 00:00:00	40.729	-73.9422	B02598	[40.729,-73.9422]
2014-08-01 00:00:00	40.7476	-73.9871	B02598	[40.7476,-73.9871]
2014-08-01 00:00:00	40.7424	-74.0044	B02598	[40.7424,-74.0044]
2014-08-01 00:00:00	40.751	-73.9869	B02598	[40.751,-73.9869]
2014-08-01 00:00:00	40.7406	-73.9902	B02598	[40.7406,-73.9902]

Next, we create a *k*-means estimator; we set the parameters to define the number of clusters and the column name for the cluster IDs. Then we use the *k*-means estimator fit method, on the VectorAssembler transformed DataFrame, to train and return a *k*-means model.



```
// create the estimator
val kmeans: KMeans = new KMeans()
.setK(20)
.setFeaturesCol("features")
.setPredictionCol("cid")
.setSeed(1L)

// use the estimator to fit (train) a KMeans model
val model: KMeansModel = kmeans.fit(df2)

// print out the cluster center latitude and longitude
println("Final Centers: ")
val centers = model.clusterCenters
centers.foreach(println)

result:
Final Centers:
[40.77486503453673, -73.95529530005687]
[40.71471849886388, -74.01021744470336]
[40.77360039001209, -73.86783834670749]
[40.68434684712066, -73.98492349953315]
...
```

Below, the 20 cluster centers are displayed on a Google Map:



Below, the 20 cluster centers and 5000 trip locations are displayed on a Google Heatmap:



We use the *k*-means model summary and *k*-means model summary predictions methods, which return the clusterIDs added as a column in a new DataFrame, in order to further analyze the clustering. Then we register the DataFrame as a temporary table in order to run SQL statements on the table.

```
// get the KMeansModelSummary from the KMeansModel
val summary : KMeansModelSummary = model.summary

// get the cluster centers in a dataframe column from the summary
val clusters : Dataframe = summary.predictions

// register the DataFrame as a temporary table
clusters.createOrReplaceTempView("uber")
clusters.show(5)
```

result:

dt	lat	lon	base	features	cid
2014-08-01 00:00:00	40.729	-73.9422	B02598	[40.729,-73.9422]	14
2014-08-01 00:00:00	40.7476	-73.9871	B02598	[40.7476,-73.9871]	10
2014-08-01 00:00:00	40.7424	-74.0044	B02598	[40.7424,-74.0044]	16
2014-08-01 00:00:00	40.751	-73.9869	B02598	[40.751,-73.9869]	10
2014-08-01 00:00:00	40.7406	-73.9902	B02598	[40.7406,-73.9902]	10

Now we can ask questions like:

Which clusters had the highest number of pickups?

```
clusters.groupBy("cid").count().orderBy(desc("count")).show(5)
```

result:

cid	count
4	101566
10	95560
11	93224
15	77019
16	75563

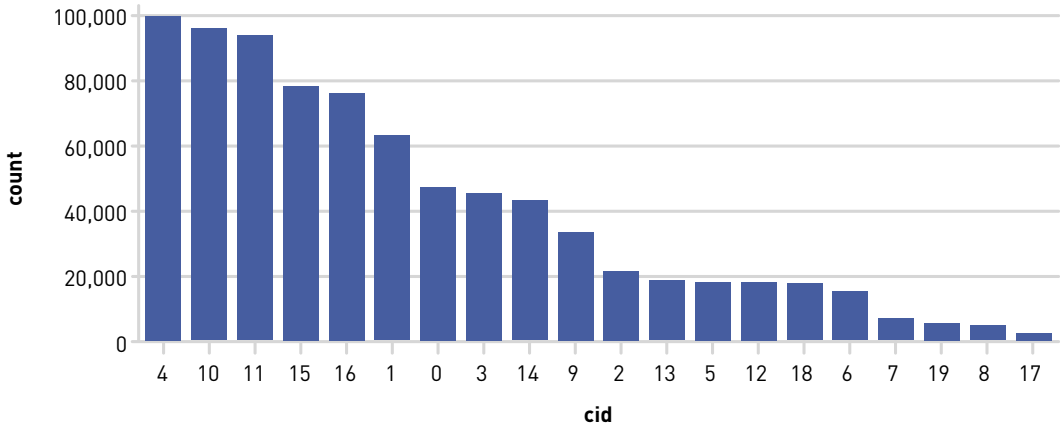
Below, the top cluster centers are displayed on a Google Map:



Which clusters had the highest number of pickups? (in Spark SQL)

```
%sql
select cid, count(cid) as count from uber group by cid order by
count desc
```

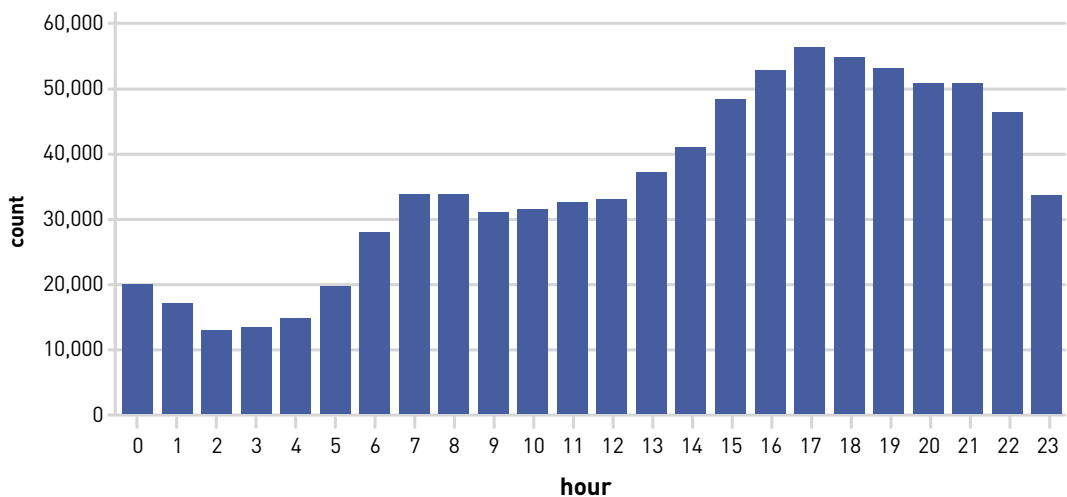
With a notebook, we can also display query results in bar charts or graphs. Below, the x axis is the cluster ID, and the y axis is the count.



Which hours of the day had the highest number of pickups?

```
%sql SELECT hour(uber.dt) as hr,count(cid) as ct FROM uber group By
hour(uber.dt) order by hour(uber.dt)
```

(Below, the x axis is the hour, and the y axis is the count.)



Which hours of the day and which cluster had the highest number of pickups?

```
clusters.select(hour($"dt").alias("hour"), $"cid")
  .groupBy("hour", "cid").agg(count("cid")
    .alias("count")).orderBy(desc("count"), $"hour").show(5)
```

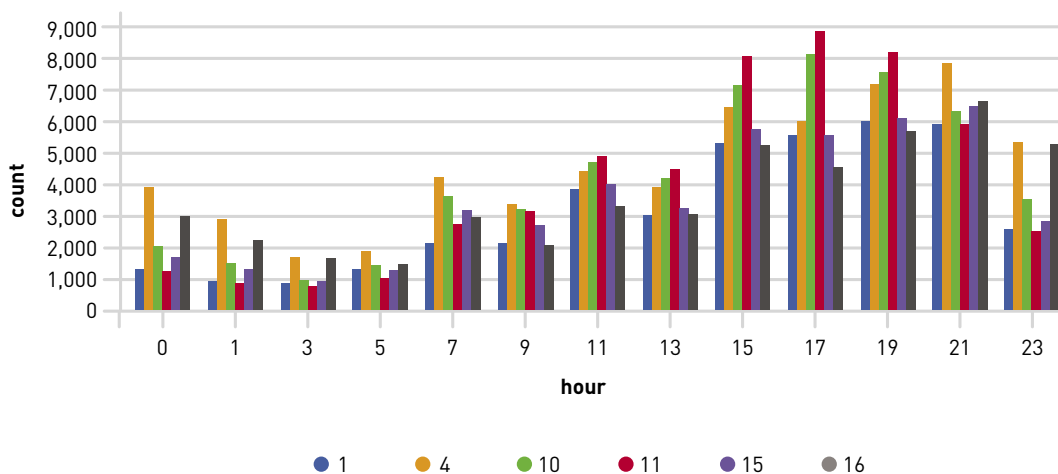
result:

```
+-----+-----+-----+
|hour|cid|count|
+-----+-----+-----+
| 16| 11| 8563|
| 17| 11| 8535|
| 17| 10| 8111|
| 18| 11| 7833|
| 18| 10| 7546|
+-----+-----+-----+
```

in Spark SQL:

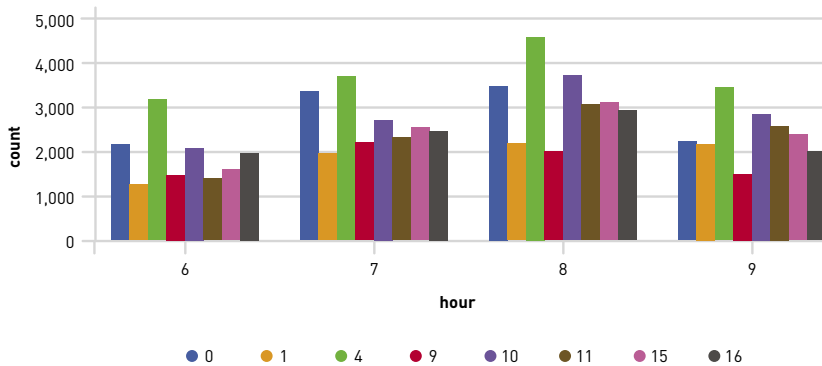
```
%sql SELECT hour(uber.dt) as hr, cid, count(cid) as ct FROM uber
WHERE cid IN (1,4, 10,11,16,15) group By hour(uber.dt), cid order by
hr, cid
```

(Below, the x axis is the hour, the y axis is the count, and the grouping is the cluster ID.)



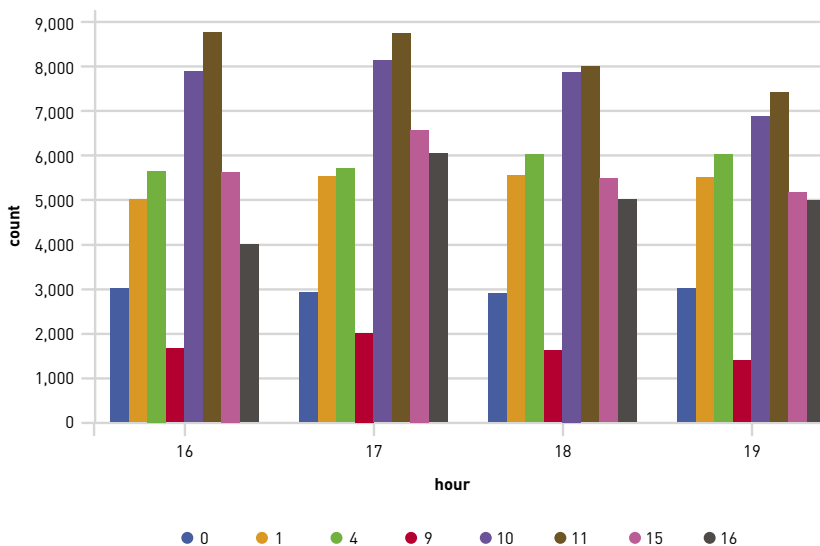
Which clusters had the highest number of pickups during morning rush hour?

```
%sql SELECT hour(uber.dt) as hr, cid, count(cid) as ct FROM uber
WHERE cid IN (0,1,4,9, 10,11,16,15) and hour(uber.dt) IN (6,7,8,9)
group By hour(uber.dt), cid order by hr, cid
```



Which clusters had the highest number of pickups during evening rush hour?

```
%sql SELECT hour(uber.dt) as hr, cid, count(cid) as ct FROM
uber WHERE cid IN (0,1,4,9, 10,11,16,15) and hour(uber.dt) IN
(16,17,18,19) group By hour(uber.dt), cid order by hr, cid
```



Save The Model

The model can be persisted to disk as shown below, in order to use later (for example, with Spark Streaming).

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

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 = KMeansModel.load(savedirectory)
```

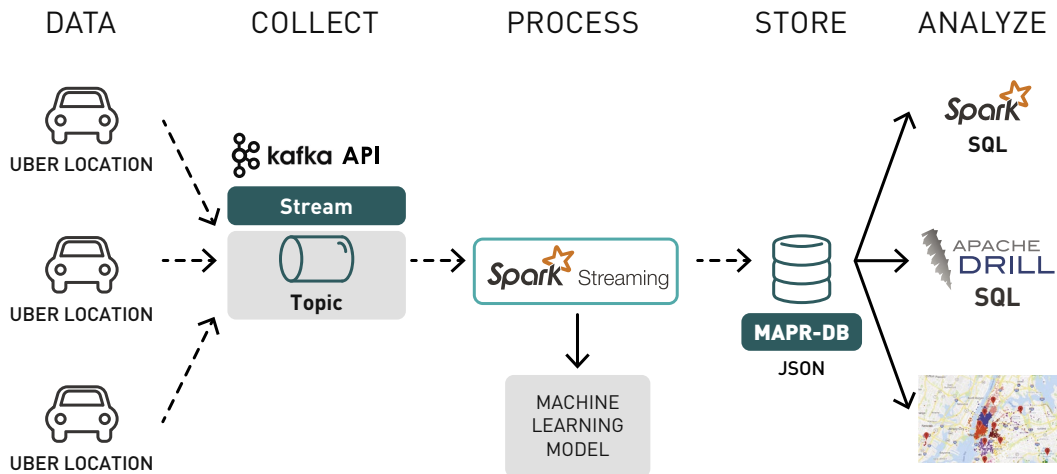
Summary

In this chapter, you learned how to use Spark ML's *k*-means clustering for analysis of Uber event data to detect and visualize popular Uber locations. In the next chapter, we will use the saved *k*-means model with streaming data.

Real-Time Analysis of Popular Uber Locations using Apache APIs: Spark Structured Streaming, Machine Learning, Kafka, and MapR-DB

According to Gartner, [20.8 billion connected things will be in use worldwide by 2020](#). Examples of connected things include cars and devices as well as applications used for healthcare, telecom, manufacturing, retail, and finance. Connected vehicles are projected to generate 25 GB of data per hour, which can be analyzed to provide real-time monitoring and apps and will lead to new concepts of mobility and vehicle usage. Leveraging the huge amounts of data coming from these devices requires processing events in real time, applying machine learning to add value, and providing scalable, fast storage. Architectures for these types of applications are usually an event-driven microservices architecture.

This chapter will discuss using the saved *k*-means model from the previous chapter with Apache Spark Structured Streaming in a data processing pipeline for cluster analysis on Uber event data to detect and visualize popular Uber locations.



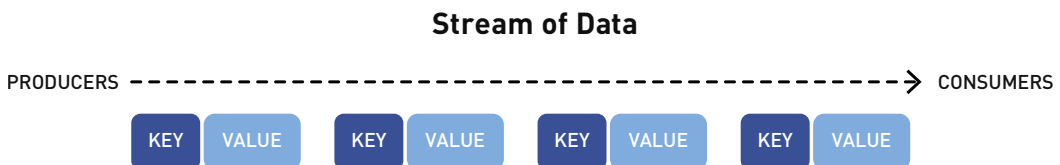
We start with a review of several Structured Streaming concepts, then explore the end-to-end use case. (Note the code in this example is not from Uber, only the data.)

Streaming Concepts

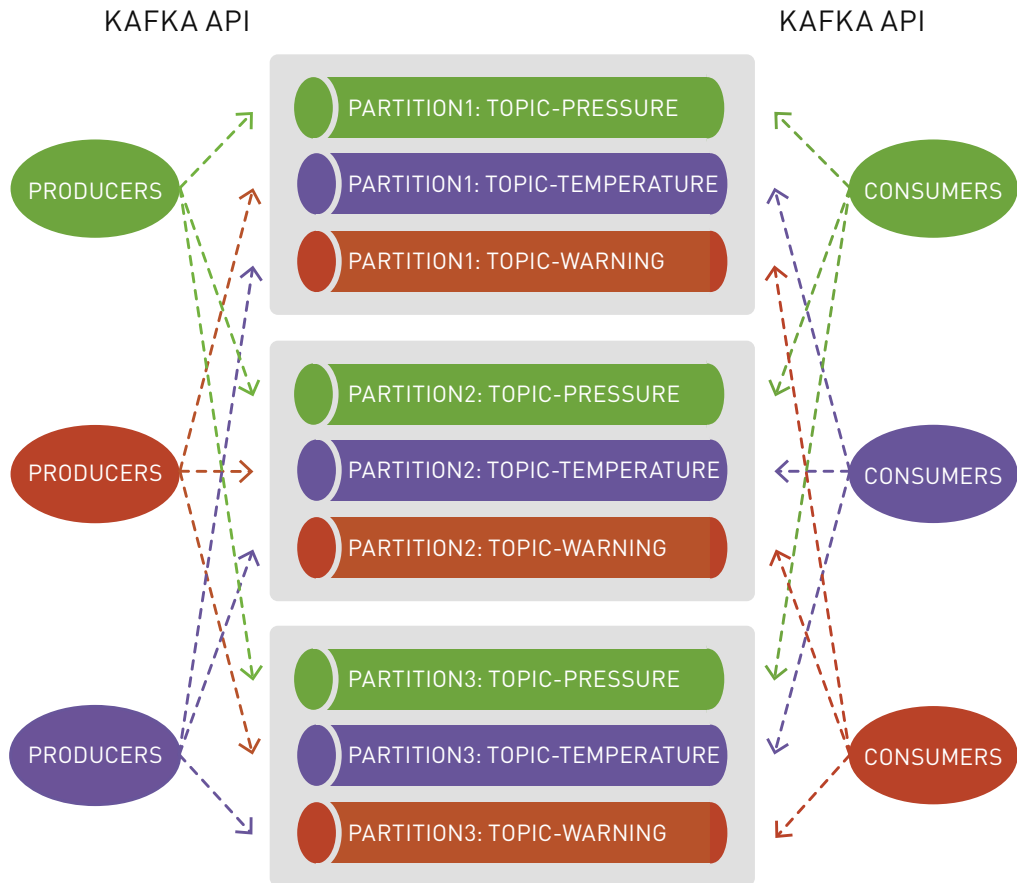
Publish-Subscribe Event Streams with MapR-ES

[MapR-ES](#) is a distributed publish-subscribe event streaming system that enables producers and consumers to exchange events in real time in a parallel and fault-tolerant manner via the Apache Kafka API.

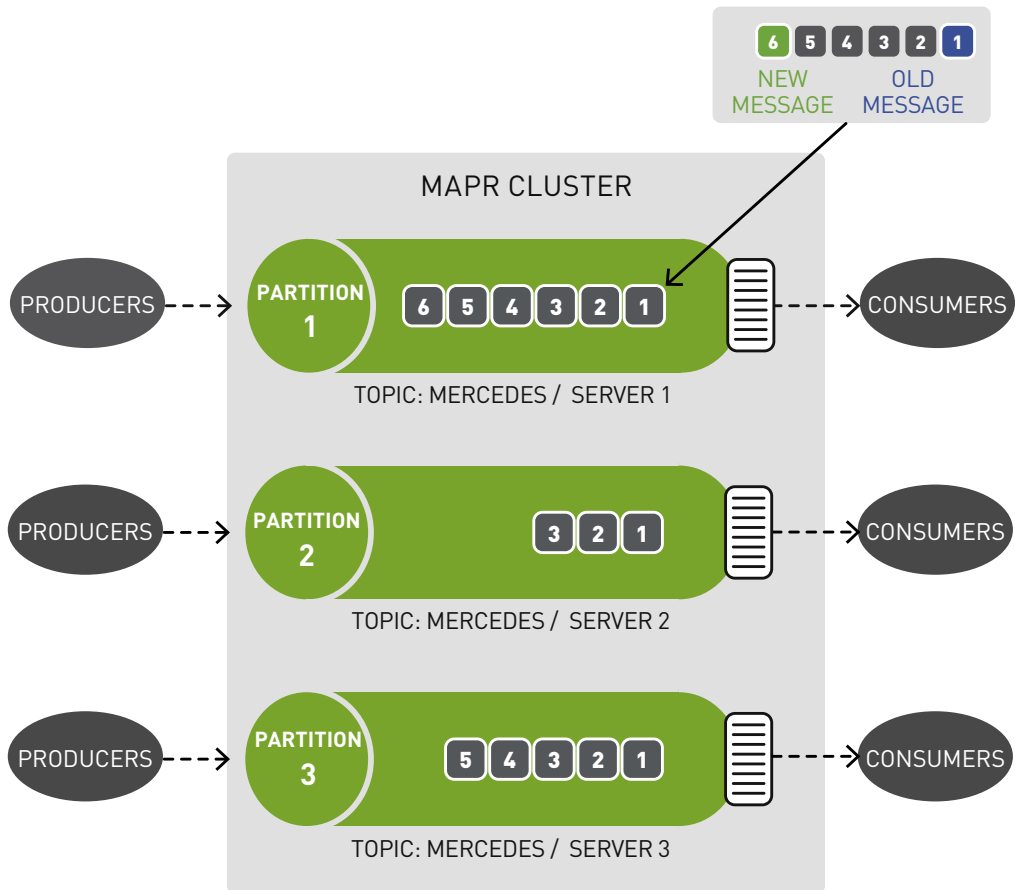
A stream represents a continuous sequence of events that goes from producers to consumers, where an event is defined as a key-value pair.



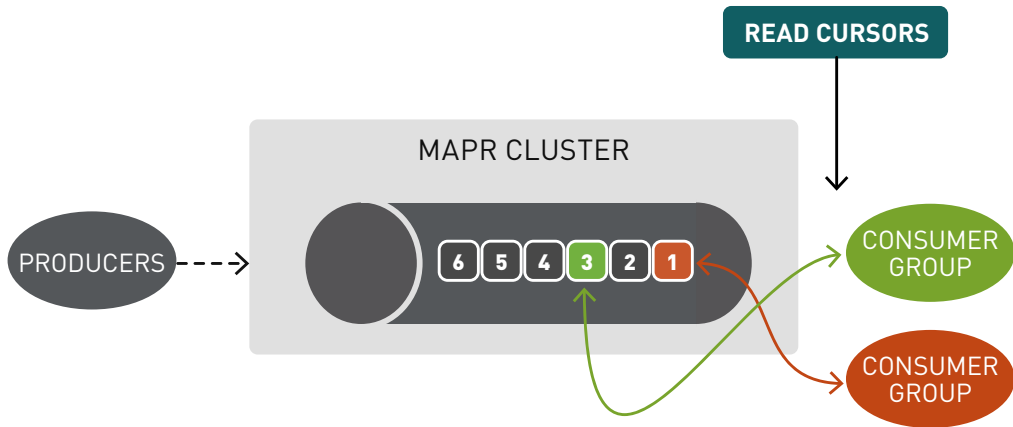
Topics are a logical stream of events. Topics organize events into categories and decouple producers from consumers. Topics are partitioned for throughput and scalability. MapR-ES can scale to very high throughput levels, easily delivering millions of messages per second using very modest hardware.



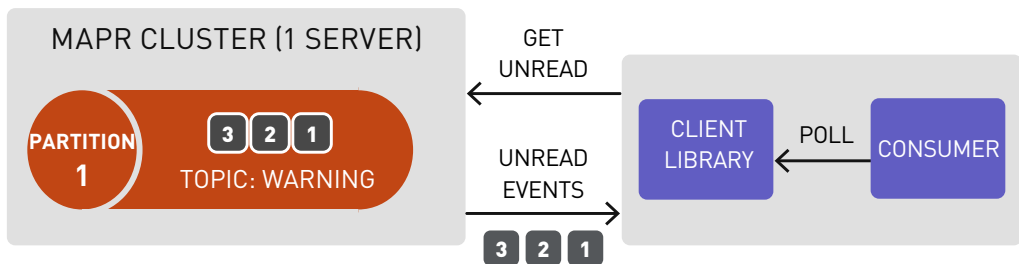
You can think of a partition like an event log: new events are appended to the end and are assigned a sequential ID number called the *offset*.



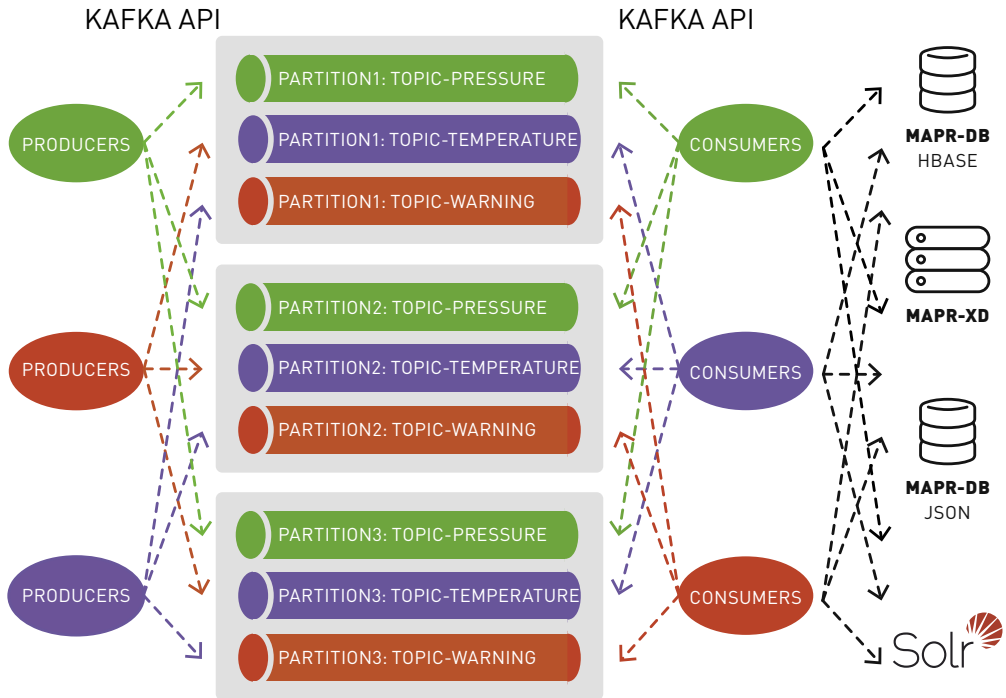
Like a queue, events are delivered in the order they are received.



Unlike a queue, however, messages are not deleted when read. They remain on the partition available to other consumers. Messages, once published, are immutable and can be retained forever.

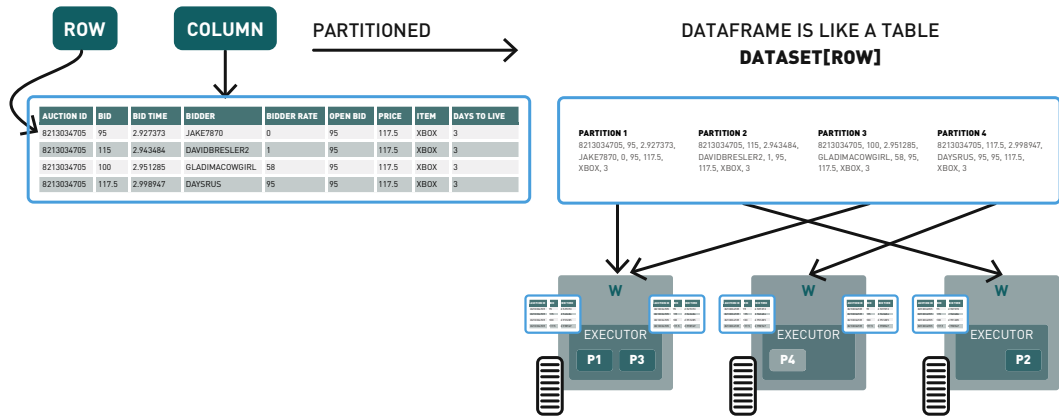


Not deleting messages when they are read allows for high performance at scale and also for processing of the same messages by different consumers for different purposes such as [multiple views with polyglot persistence](#).



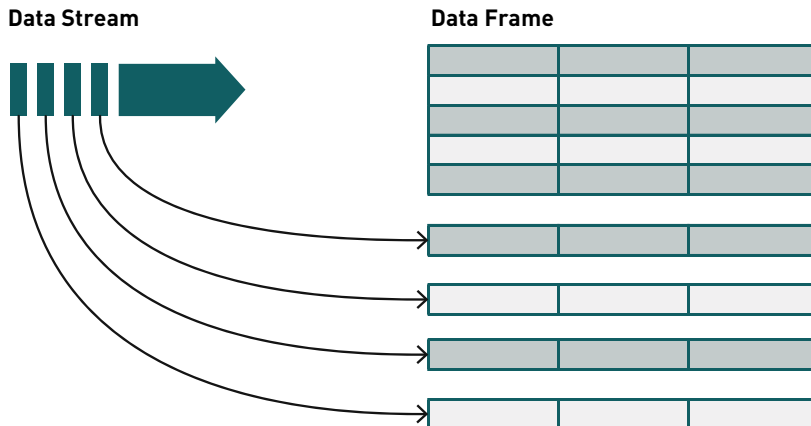
Spark Dataset, DataFrame, SQL

A Spark Dataset is a distributed collection of typed objects partitioned across multiple nodes in a cluster. A Dataset can be manipulated using functional transformations (map, flatMap, filter, etc.) and/or Spark SQL. A DataFrame is a Dataset of Row objects and represents a table of data with rows and columns.

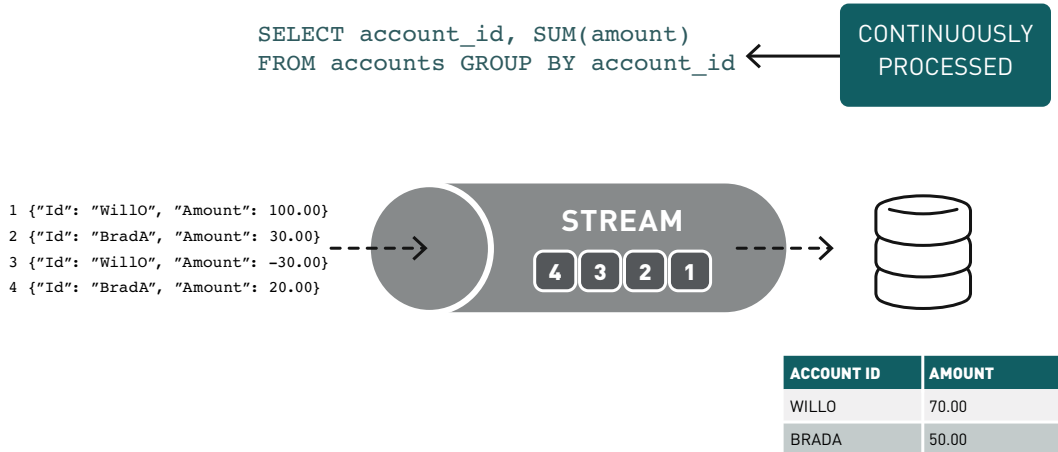


Spark Structured Streaming

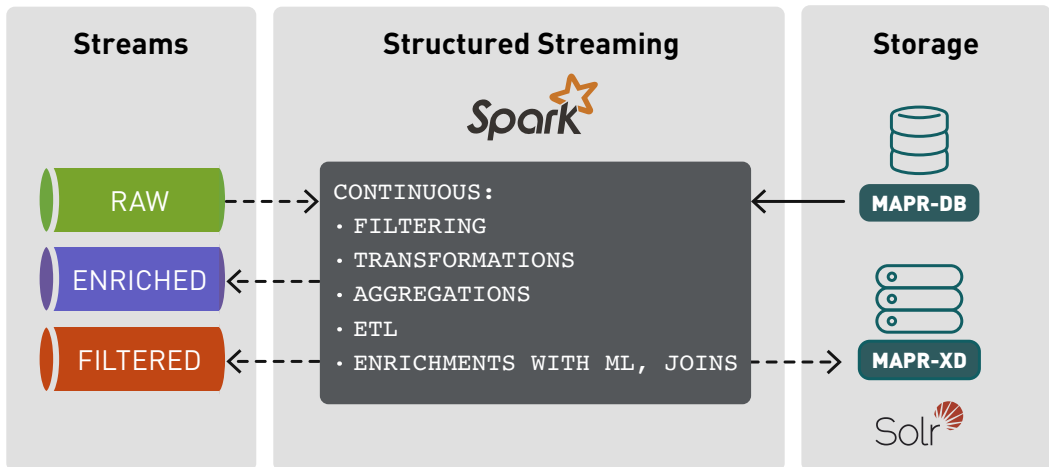
Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. Structured Streaming enables you to view data published to Kafka as an unbounded DataFrame and process this data with the same DataFrame, Dataset, and SQL APIs used for batch processing.



As streaming data continues to arrive, the Spark SQL engine incrementally and continuously processes it and updates the final result.

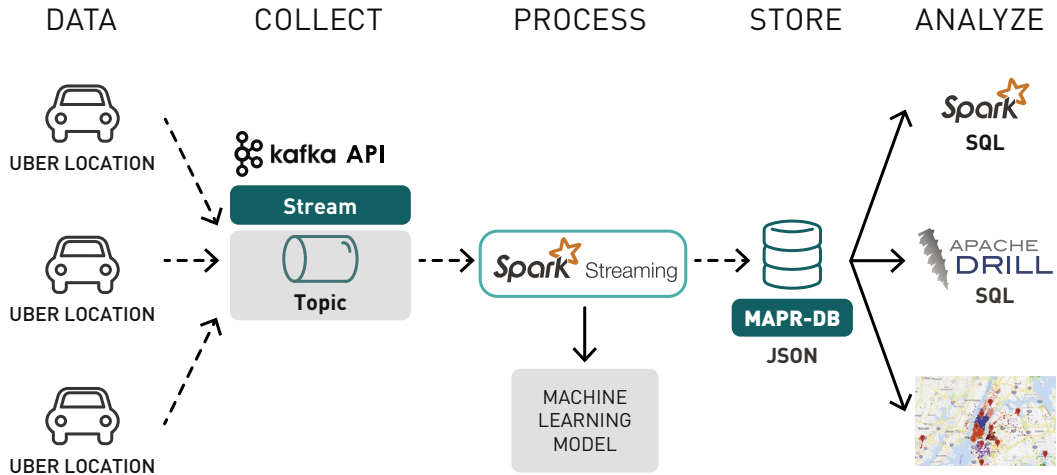


Stream processing of events is useful for real-time ETL, filtering, transforming, creating counters and aggregations, correlating values, enriching with other data sources or machine learning, persisting to files or Database, and publishing to a different topic for pipelines.

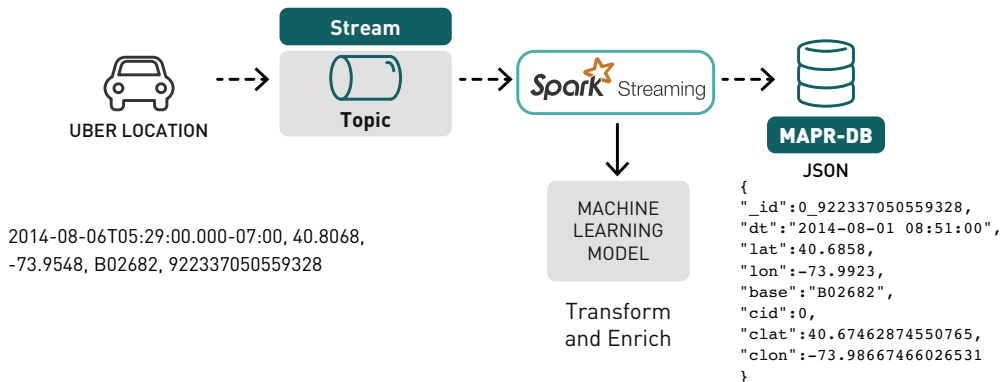


Spark Structured Streaming Use Case Example Code

Below is the data processing pipeline for this use case of cluster analysis on Uber event data to detect popular pickup locations.



1. Uber trip data is published to a MapR-ES topic using the Kafka API.
2. A Spark Streaming application subscribed to the topic:
 - Ingests a stream of Uber trip data
 - Uses a deployed machine learning model to enrich the trip data with a cluster ID and cluster location
 - Stores the transformed and enriched data in MapR-DB JSON



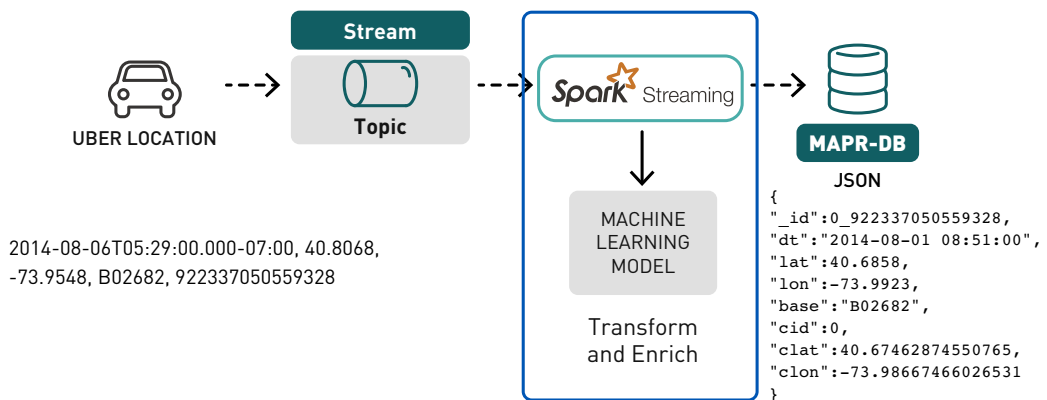
Example Use Case Data

The example data set is Uber trip data from [chapter 6](#). The incoming data is in CSV format; an example is shown below, with the header:

```
date/time, latitude, longitude, base, reverse time stamp  
2014-08-06T05:29:00.000-07:00, 40.7276, -74.0033, B02682,  
9223370505593280605
```

We enrich this data with the cluster ID and location, then transform it into the following JSON object:

```
{  
  "_id":0_922337050559328,  
  "dt":"2014-08-01 08:51:00",  
  "lat":40.6858,  
  "lon":-73.9923,  
  "base":"B02682",  
  "cid":0,  
  "clat":40.67462874550765,  
  "clon":-73.98667466026531  
}
```



Loading the K-Means Model

The Spark `KMeansModel` class is used to load a *k*-means model, which was [fitted on the historical Uber trip data and then saved to the MapR-XD cluster](#). Next, a Dataset of Cluster Center IDs and location is created to join later with the Uber trip locations.

```
// load the saved model from the distributed file system
val model = KMeansModel.load(savedirectory)

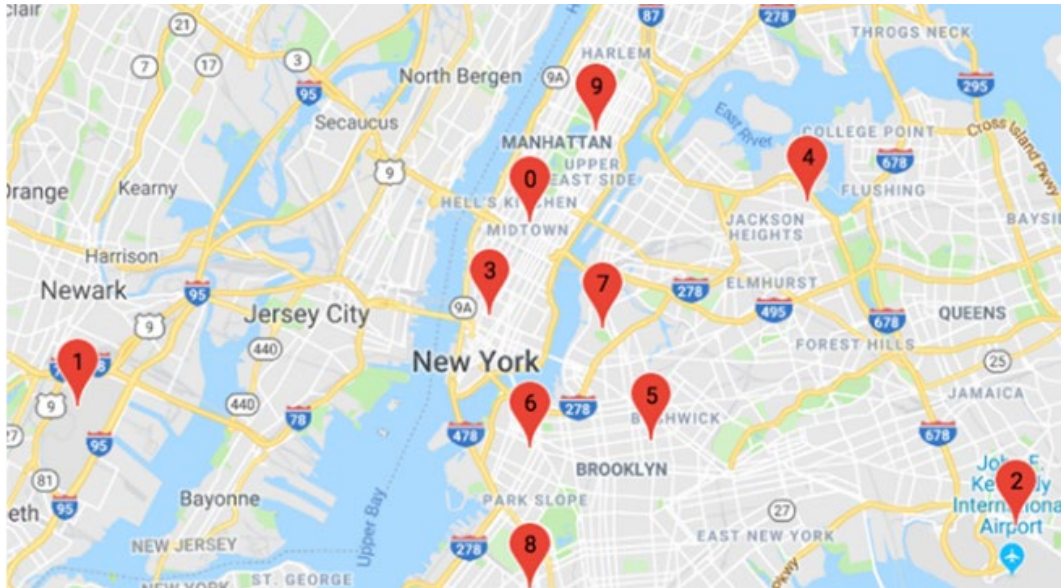
// create a Dataset with cluster id and location
case class Center(cid: Integer, clat: Double,
clon: Double)

var ac = new Array[Center](10)
var index: Int = 0
model.clusterCenters.foreach(x => {
    ac(index) = Center(index, x(0), x(1));
    index += 1;
})

val ccdf = spark.createDataset(ac)
ccdf.show(3)
```

cid	clat	clon
0	40.7564201526695	-73.98253669425347
1	40.69774864372469	-74.1746190485833
2	40.65913663371848	-73.77616609142027

Below the cluster centers are displayed on a Google Map in a Zeppelin notebook:



(Note: In the previous chapter, we used 20 cluster IDs; in this example, we are using 10. Also, when you run this model, the cluster IDs will be different, but the locations will be about the same, depending on how many iterations you run. When you save and reload the same model, the cluster IDs will be the same.)

Reading Data from Kafka Topics

In order to read from Kafka, we must first specify the stream format, topic, and offset options. For more information on the configuration parameters, [see the MapR Streams documentation](#).

```
var topic: String = "/apps/uberstream:ubers"

val df1 = spark.readStream.format("kafka")
    .option("kafka.bootstrap.servers", "maprdemo:9092")
    .option("subscribe", topic)
    .option("group.id", "testgroup")
    .option("startingOffsets", "earliest")
    .option("failOnDataLoss", false)
    .option("maxOffsetsPerTrigger", 1000)
    .load()
```

This returns a DataFrame with the following schema:

```
df1.printSchema()

result:
root
 |-- key: binary (nullable = true)
 |-- value: binary (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)
```

The next step is to parse and transform the binary values column into a Dataset of Uber objects.

Parsing the Message Values into a Dataset of Uber Objects

A Scala Uber case class defines the schema corresponding to the CSV records. The `parseUber` function parses a comma separated value string into an Uber object.

```
case class Uber(dt: String, lat: Double, lon: Double, base: String,
rdt: String)

// Parse string into Uber case class
def parseUber(str: String): Uber = {
  val p = str.split(",")
  Uber(p(0), p(1).toDouble, p(2).toDouble, p(3), p(4))
}
```

In the code below, we register a user-defined function (UDF) to deserialize the message value strings using the `parseUber` function. Then we use the UDF in a select expression with a String Cast of the `df1` column value, which returns a DataFrame of Uber objects.

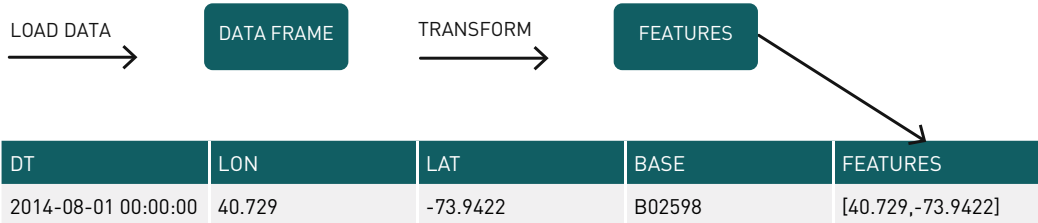
```
import spark.implicits._

spark.udf.register("deserialize",
  (message: String) => parseUber(message))

val df2 = df1.selectExpr("""deserialize(CAST(value as STRING)) AS
message""").select($"message".as[Uber])
```

Enriching the Dataset of Uber Objects with Cluster Center IDs and Location

A VectorAssembler is used to transform and return a new DataFrame with the latitude and longitude feature columns in a vector column.

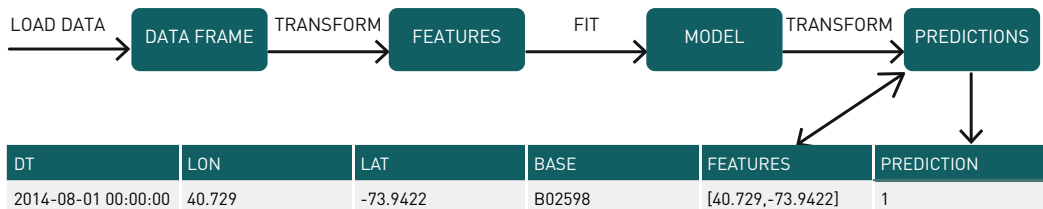


```
val featureCols = Array("lat", "lon")

val assembler = new VectorAssembler()
  .setInputCols(featureCols)
  .setOutputCol("features")

val df3 = assembler.transform(df2)
```

The k -means model is used to get the clusters from the features with the model transform method, which returns a DataFrame with the cluster ID (labeled predictions). This resulting Dataset is joined with the cluster center Dataset created earlier (ccdf) to create a Dataset of UberC objects, which contain the trip information combined with the cluster Center ID and location.



```
//use model to get the clusters from the features
val clusters1 = model.transform(df3)

val temp= clusters1.select($"dt".cast(TimestampType),
    $"lat", $"lon", $"base", $"rdt", $"prediction"
    .alias("cid"))

// Uber class with Cluster id,lat lon
case class UberC(dt: java.sql.Timestamp, lat: Double,
    lon: Double, base: String, rdt: String, cid: Integer,
    clat: Double, clon: Double) extends Serializable

val clusters = temp.join(ccdf, Seq("cid")).as[UberC]
```

The final Dataset transformation is to add a unique ID to our objects for storing in MapR-DB JSON. The createUberwId function creates a unique ID, consisting of the cluster ID and the reverse timestamp. Since MapR-DB partitions and sorts rows by the ID, the rows will be sorted by cluster ID with the most recent first. This function is used with a map to create a Dataset of UberwId objects.

```
// Uber with unique Id and Cluster id and cluster lat lon
case class UberwId(_id: String, dt: java.sql.Timestamp,
    base: String, cid: Integer,
    clat: Double, clon: Double)

// enrich with unique id for Mapr-DB
def createUberwId(uber: UberC): UberwId = {
    val id = uber.cid + "_" + uber.rdt
    UberwId(id, uber.dt, uber.lat, uber.lon, uber.base,
    uber.cid , uber.clat, uber.clon)
}

val cdf: Dataset[UberwId] = clusters.map(uber =>
    createUberwId(uber))
```


Writing to a Memory Sink

We have now set up the enrichments and transformations on the streaming data. Next, for debugging purposes, we can start receiving data and storing the data in memory as an in-memory table, which can then be queried.

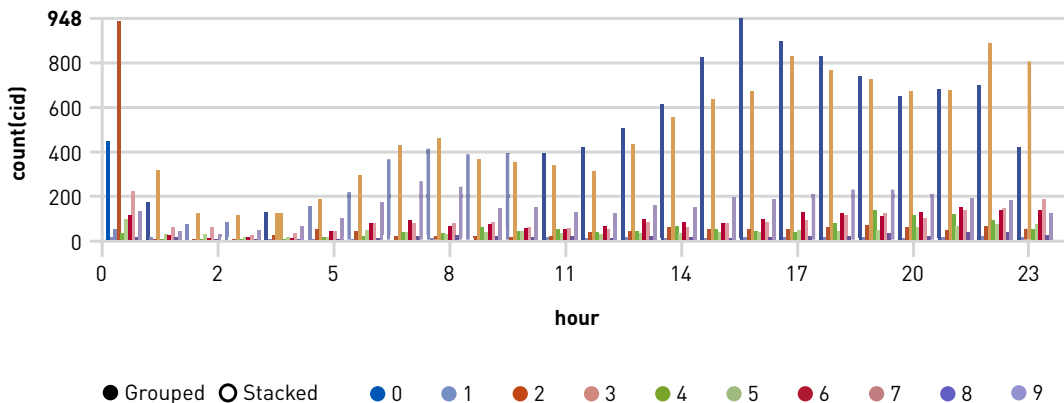
```
val streamingquery = cdf
  .writeStream
  .queryName("uber")
  .format("memory")
  .outputMode("append")
  .start
```

Here is example output from `%sql select * from Uber limit 10:`

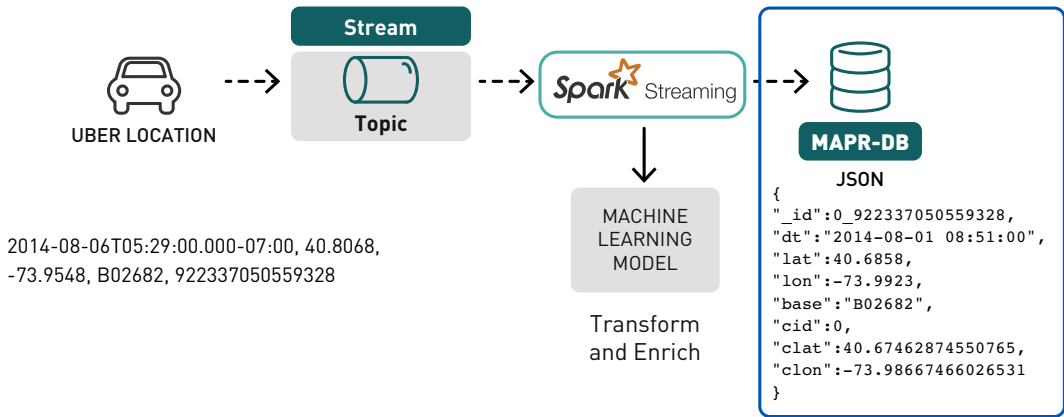
_ID	DT	LAT	LON	BASE	CID	CLAT	CLON
0_9223370505601502336	2014-08-01 00:00:00	40.7476	-73.9871	B02598	0	40.7564201526695	-73.98253669425347
3_9223370505601500625	2014-08-01 00:00:00	40.7424	-74.0044	B02598	3	40.727167721391965	-73.99996932251409
0_9223370505601500564	2014-08-01 00:00:00	40.751	-73.9869	B02598	0	40.7564201526695	-73.98253669425347

Now we can query the streaming data to ask questions, like: which hours and clusters have the highest number of pickups? (Output is shown in a Zeppelin notebook.)

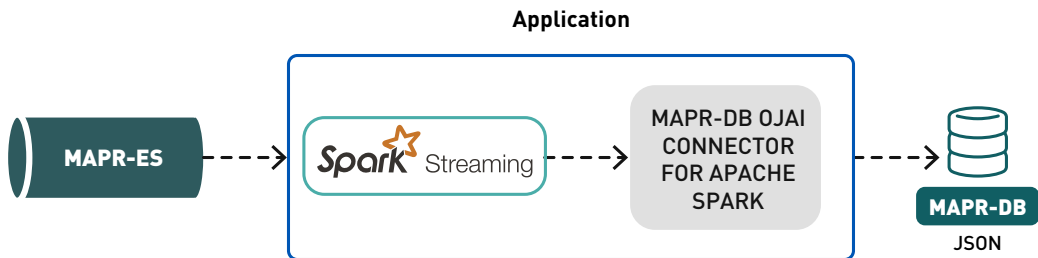
```
%sql
SELECT hour(uber.dt) as hr,cid, count(cid) as ct FROM uber group By
hour(uber.dt), cid
```



Spark Streaming Writing to MapR-DB



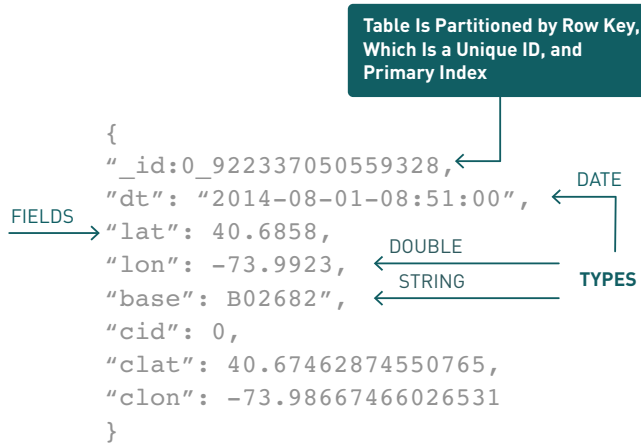
The MapR-DB Connector for Apache Spark enables you to use MapR-DB as a sink for Spark Structured Streaming or Spark Streaming.



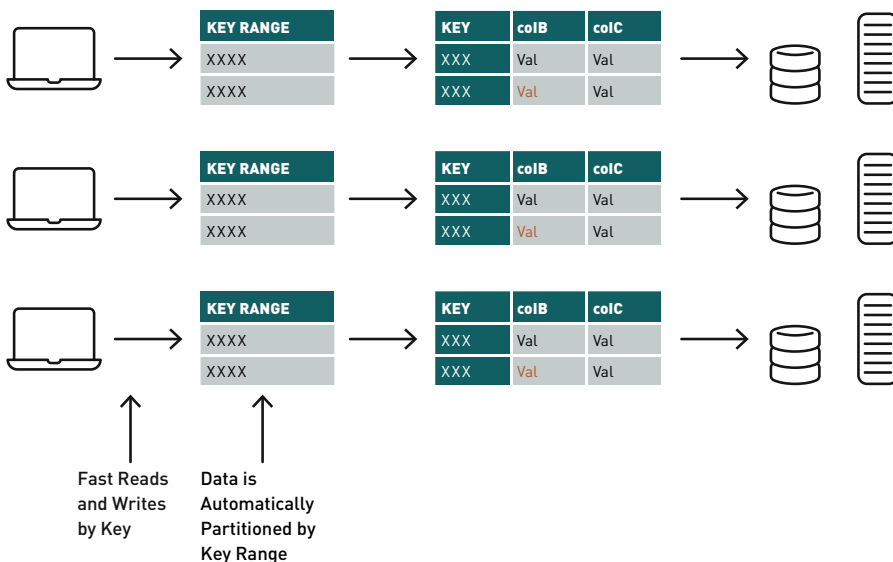
One of the challenges when you are processing lots of streaming data is: where do you want to store it? For this application, [MapR-DB JSON](#), a high performance NoSQL database, was chosen for its scalability and flexible ease of use with JSON.

JSON Schema Flexibility

MapR-DB supports JSON documents as a native data store. MapR-DB makes it easy to store, query, and build applications with JSON documents. The Spark connector makes it easy to build real-time or batch pipelines between your JSON data and MapR-DB and leverage Spark within the pipeline.



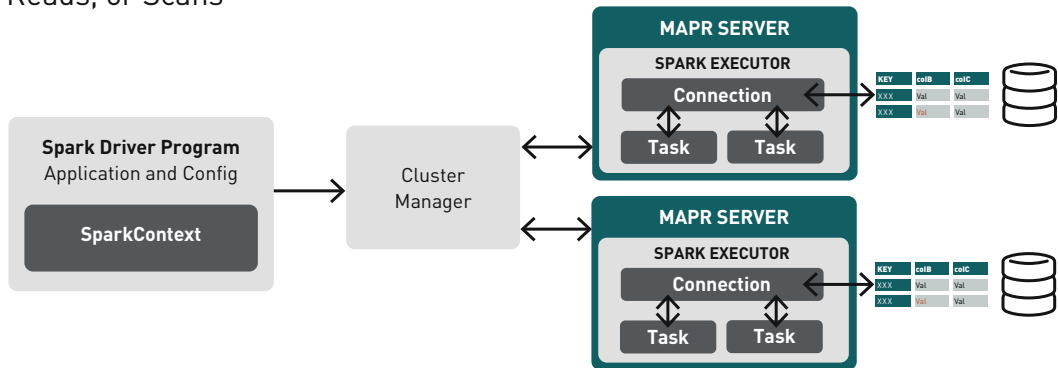
With MapR-DB, a table is automatically partitioned into tablets across a cluster by key range, providing for scalable and fast reads and writes by row key. In this use case, the row key, the `_id`, consists of the cluster ID and reverse timestamp, so the table is automatically partitioned and sorted by cluster ID with the most recent first.



The Spark MapR-DB Connector leverages the Spark [DataSource API](#). The connector architecture has a connection object in every Spark Executor, allowing for distributed parallel writes, reads, or scans with MapR-DB tablets (partitions).

Connection in Every Spark Executor

Allowing for Distributed Parallel Writes,
Reads, or Scans



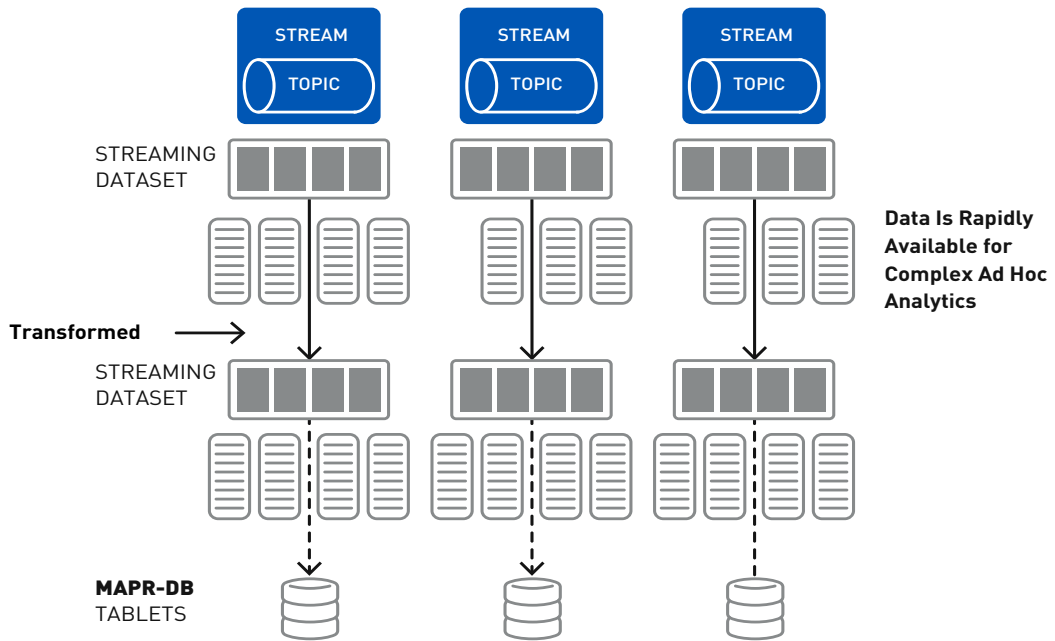
Writing to a MapR-DB Sink

To write a Spark Stream to MapR-DB, specify the [format with the tablePath, idFieldPath, createTable, bulkMode, and sampleSize parameters](#). The following example writes out the cdf DataFrame to MapR-DB and starts the stream.

```
import com.mapr.db.spark.impl._
import com.mapr.db.spark.streaming._
import com.mapr.db.spark.sql._
import com.mapr.db.spark.streaming.MapRDBSourceConfig

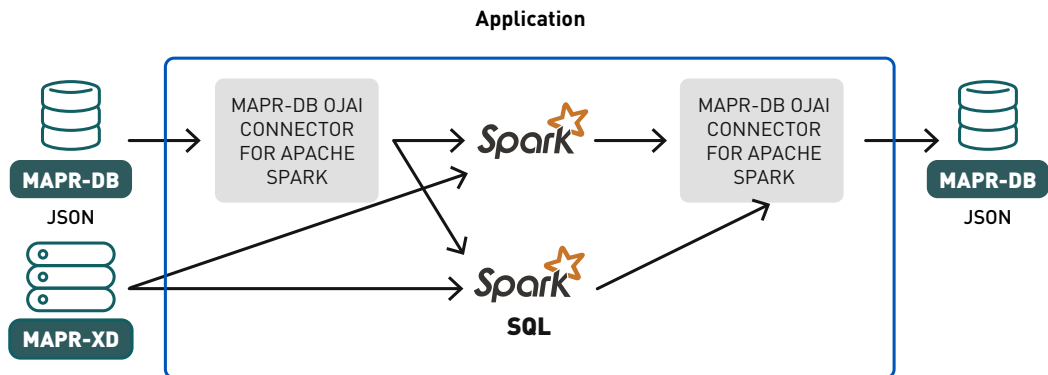
var tableName: String = "/apps/ubertable"
val writedb = cdf.writeStream
  .format(MapRDBSourceConfig.Format)
  .option(MapRDBSourceConfig.TablePathOption, tableName)
  .option(MapRDBSourceConfig.IdFieldPathOption, "_id")
  .option(MapRDBSourceConfig.CreateTableOption, false)
  .option("checkpointLocation", "/tmp/uberdb")
  .option(MapRDBSourceConfig.BulkModeOption, true)
  .option(MapRDBSourceConfig.SampleSizeOption, 1000)

writedb.start()
```



Querying MapR-DB JSON with Spark SQL

The Spark MapR-DB Connector enables users to perform complex SQL queries and updates on top of MapR-DB using a Spark Dataset, while applying critical techniques such as projection and filter pushdown, custom partitioning, and data locality.



Loading Data from MapR-DB into a Spark Dataset

To load data from a MapR-DB JSON table into an Apache Spark Dataset, we invoke the `loadFromMapRDB` method on a `SparkSession` object, providing the `tableName`, `schema`, and `case class`. This returns a Dataset of `UberwId` objects:

```
case class UberwId(_id: String, dt: java.sql.Timestamp,
    lat: Double, lon: Double, base: String, cid: Integer,
    clat: Double, clon: Double) extends Serializable

val schema = StructType(Array(
    StructField("_id", StringType, true),
    StructField("dt", TimestampType, true),
    StructField("lat", DoubleType, true),
    StructField("lon", DoubleType, true),
    StructField("base", StringType, true),
    StructField("cid", IntegerType, true),
    StructField("clat", DoubleType, true),
    StructField("clon", DoubleType, true)
))

var tableName: String = "/apps/ubertable"
val df: Dataset[UberwId] = spark
    .loadFromMapRDB[UberwId](tableName, schema)
    .as[UberwId]

df.createOrReplaceTempView("uber")
```

Explore and Query the Uber Data with Spark SQL

Now we can query the data that is continuously streaming into MapR-DB to ask questions with the Spark DataFrames domain-specific language or with Spark SQL.

Below, we use the DataFrames `show` method to display the first rows in tabular format. (Note how the rows are partitioned and sorted by the `_id`, which is composed of the cluster ID and reverse timestamp; the reverse timestamp sorts most recent first.)

```
df.show
```

	_id	dt	lat	lon	base	cid	clat	clon
0_922337050559328...	[2014-08-06 05:28	0:00	40.7663	-73.9915	B02598	0	40.7564201526695	-73.98253669425347
0_922337050559328...	[2014-08-06 05:27	0:00	40.7674	-73.9848	B02682	0	40.7564201526695	-73.98253669425347
0_922337050559328...	[2014-08-06 05:27	0:00	40.7564	-73.9975	B02617	0	40.7564201526695	-73.98253669425347
0_922337050559328...	[2014-08-06 05:26	0:00	40.768	-73.9833	B02617	0	40.7564201526695	-73.98253669425347
0_922337050559328...	[2014-08-06 05:26	0:00	40.7656	-73.9636	B02598	0	40.7564201526695	-73.98253669425347
0_922337050559328...	[2014-08-06 05:25	0:00	40.7499	-73.9895	B02764	0	40.7564201526695	-73.98253669425347

What are the top 5 pickup cluster locations?

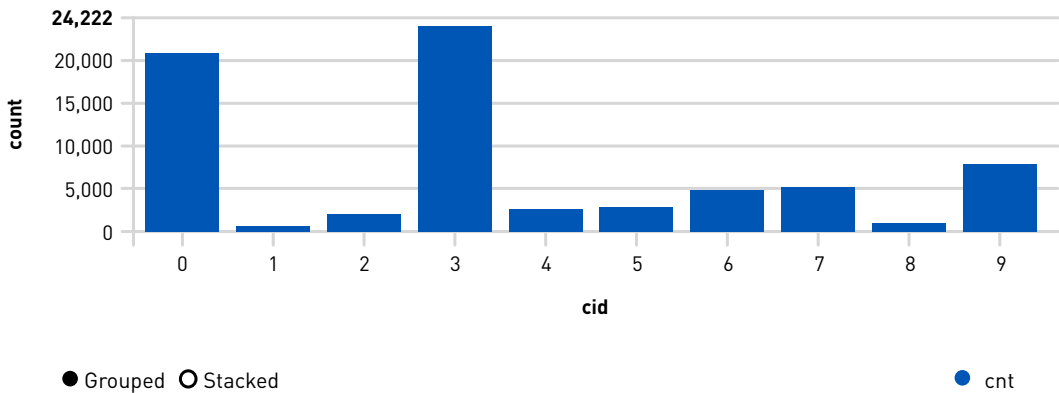
```
df.groupBy("cid").count().orderBy(desc("count")).show(5)
```

result:

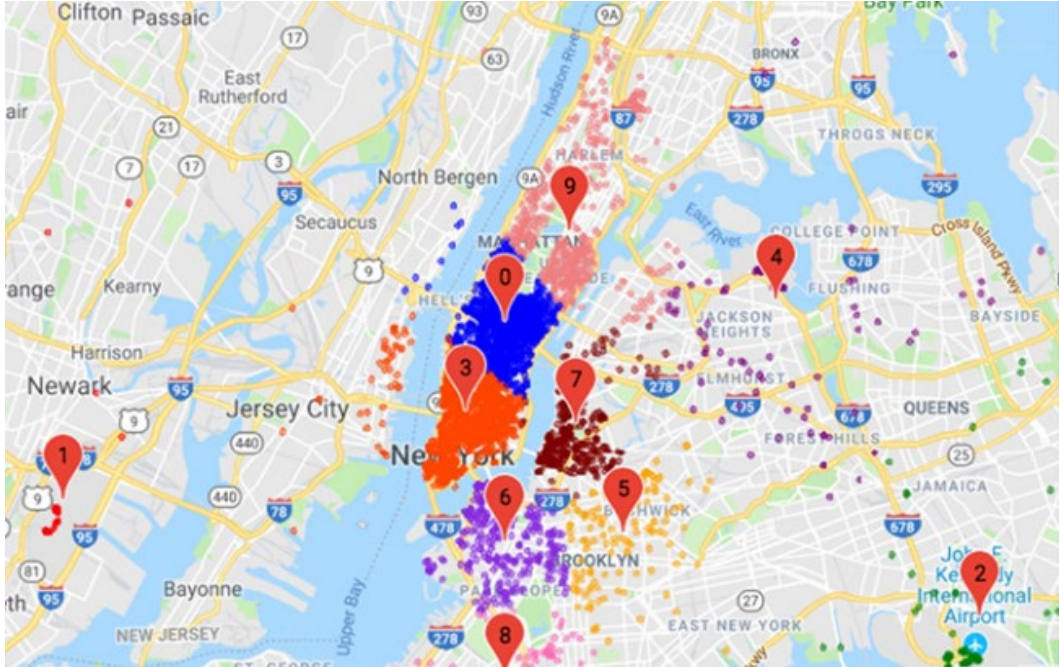
```
+---+-----+
|cid|count|
+---+-----+
|  3|43544|
|  0|43301|
|  9|15182|
|  6| 8411|
|  7| 8324|
+---+-----+
```

or with Spark SQL:

```
%sql SELECT COUNT(cid), cid FROM uber GROUP BY cid ORDER BY
COUNT(cid) DESC
```

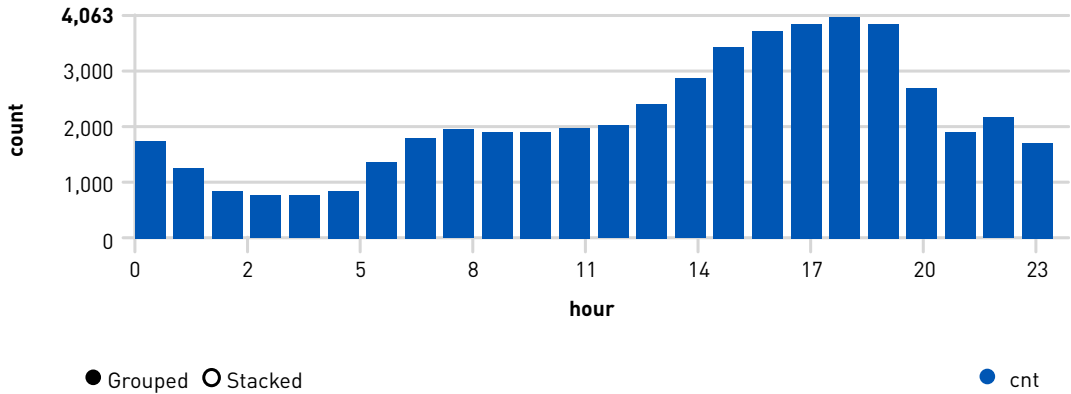


With Angular and Google Maps script in a Zeppelin notebook, we can display cluster center markers and the latest 5000 trip locations on a map, which shows that the most popular locations – 0, 3, and 9 – are in Manhattan.



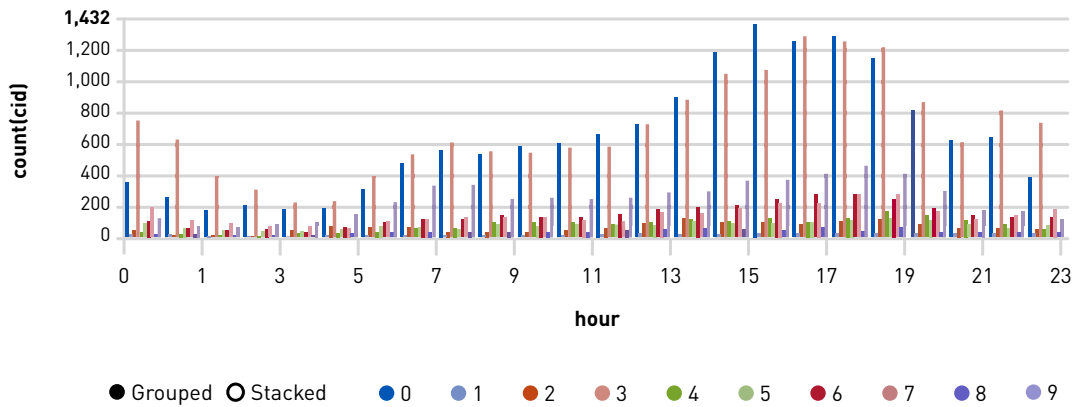
Which hours have the highest number of pickups for cluster 0?

```
df.filter($"_id" <= "1")  
  .select(hour($"dt").alias("hour"), $"cid")  
  .groupBy("hour", "cid").agg(count($"cid")  
  .alias("count")).show
```

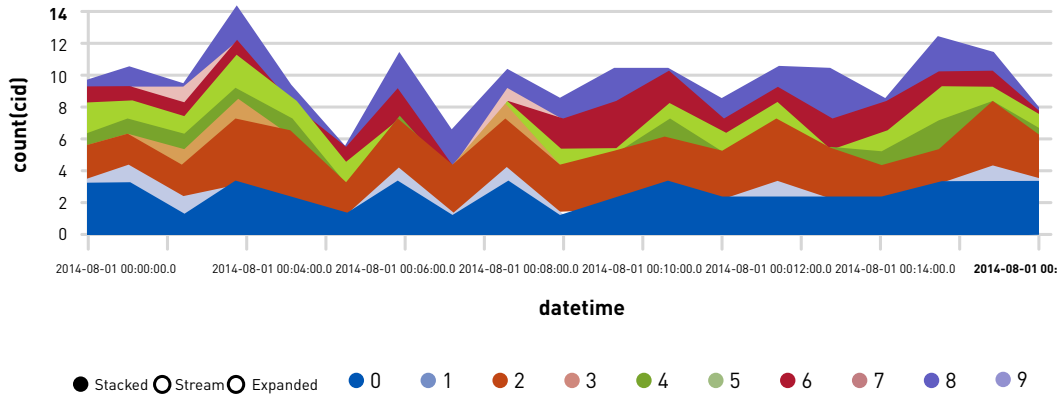
Which hours of the day and which cluster had the highest number of pickups?

```
%sql SELECT hour(uber.dt), cid, count(cid) FROM uber GROUP BY  
hour(uber.dt), cid
```



Display cluster counts for Uber trips by datetime:

```
%sql select cid, dt, count(cid) as count from uber group by dt, cid  
order by dt, cid limit 100
```



Summary

In this chapter, you learned how to use the following:

- A Spark machine learning model in a Spark Structured Streaming application
- Spark Structured Streaming with MapR-ES to ingest messages using the Kafka API
- Spark Structured Streaming to persist to MapR-DB for continuously and rapidly available SQL analysis

All of the components of the use case architecture we just discussed can run on the same cluster with the MapR Data Platform.

Predicting Forest Fire Locations with *K*-Means in Spark

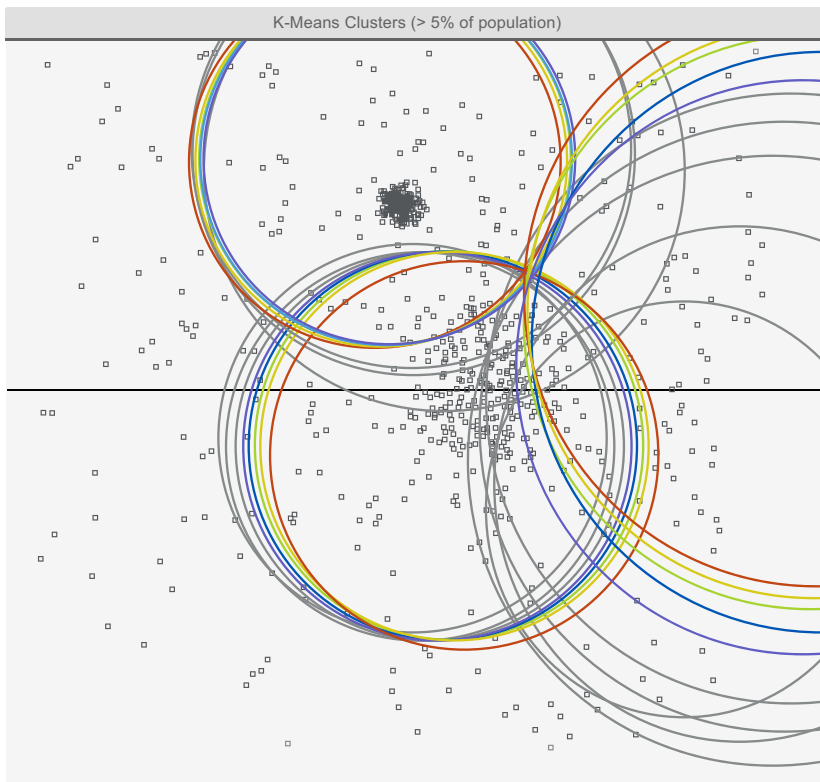
Contributed by: Ian Downard

Every summer, wildfires become front-of-mind for thousands of people who live in the Western United States. In recent years, the business of firefighting has been setting records. [Wildland fire suppression costs exceeded \\$2 billion in 2017](#), making it the most expensive year on record for the United States Forest Service. The Forest Service is the primary agency of the United States government responsible for wildfire management. Part of their mission is to provide the public with a wide variety of information about past and current wildfires. This includes datasets that describe wildfires which have occurred in Canada and the United States since the year 2000. That data can be downloaded from <https://fsapps.nwcg.gov/gisdata.php>.

Anytime you have lat/long coordinates, you have an opportunity to do data science with *k*-means clustering and visualization on a map. Let's look at one small way in which *k*-means could be applied within the context of wildland firefighting to reduce costs and incident response time.

Problem Statement

Fires have a tendency to explode in size. It's not unusual for fires to grow by 40,000 acres in one day when winds are high and the terrain is steep. This is why it's so important to respond to fires as quickly as possible when they start. However, the logistics of moving firefighting equipment is one of the major factors limiting incident response time. By strategically staging equipment where fires are prone to occur, it may be possible to improve incident response time, but how do we pinpoint those locations?



The Solution

One way to estimate where wildfires are prone to occur is to partition the locations of past burns into clusters. The centroids for those clusters could conceivably help wildland management agencies optimally place heavy wildfire suppression equipment, such as water tanks, as near as possible to where fires are likely to occur. The k -means clustering algorithm is perfectly suited for this purpose.

The first step in solving this problem is to download the dataset containing locations for past burns. Here is how to do that with Bash:

```
mkdir -p data/fires
cd data/fires
curl -s --remote-name https://fsapps.nwcg.gov/afm/data/fireptdata/modis_fire_2016_365_conus_shapefile.zip
curl -s --remote-name https://fsapps.nwcg.gov/afm/data/fireptdata/modis_fire_2015_365_conus_shapefile.zip
curl -s --remote-name https://fsapps.nwcg.gov/afm/data/fireptdata/modis_fire_2014_365_conus_shapefile.zip
curl -s --remote-name https://fsapps.nwcg.gov/afm/data/fireptdata/modis_fire_2013_365_conus_shapefile.zip
curl -s --remote-name https://fsapps.nwcg.gov/afm/data/fireptdata/modis_fire_2012_366_conus_shapefile.zip
curl -s --remote-name https://fsapps.nwcg.gov/afm/data/fireptdata/modis_fire_2011_365_conus_shapefile.zip
curl -s --remote-name https://fsapps.nwcg.gov/afm/data/fireptdata/modis_fire_2010_365_conus_shapefile.zip
curl -s --remote-name https://fsapps.nwcg.gov/afm/data/fireptdata/modis_fire_2009_365_conus_shapefile.zip
curl -s --remote-name https://fsapps.nwcg.gov/afm/data/fireptdata/mcd14ml_2008_005_01_conus_shp.zip
curl -s --remote-name https://fsapps.nwcg.gov/afm/data/fireptdata/mcd14ml_2007_005_01_conus_shp.zip
curl -s --remote-name https://fsapps.nwcg.gov/afm/data/fireptdata/mcd14ml_2006_005_01_conus_shp.zip
curl -s --remote-name https://fsapps.nwcg.gov/afm/data/fireptdata/mcd14ml_2005_005_01_conus_shp.zip
curl -s --remote-name https://fsapps.nwcg.gov/afm/data/fireptdata/mcd14ml_2004_005_01_conus_shp.zip
curl -s --remote-name https://fsapps.nwcg.gov/afm/data/fireptdata/mcd14ml_2003_005_01_conus_shp.zip
curl -s --remote-name https://fsapps.nwcg.gov/afm/data/fireptdata/mcd14ml_2002_005_01_conus_shp.zip
curl -s --remote-name https://fsapps.nwcg.gov/afm/data/fireptdata/mcd14ml_2001_005_01_conus_shp.zip
find modis*.zip | xargs -I {} unzip {} modis*.dbf
find mcd*.zip | xargs -I {} unzip {} mcd*.dbf
```

Raw data is hardly ever suitable for machine learning without cleansing. The process of cleaning and unifying messy datasets is called “data wrangling,” and it frequently comprises the bulk of the effort involved in real-world machine learning.

The dataset used for this study also requires a bit of wrangling. It’s provided in an inconvenient [shapefile](#) format, which we’ll transform into CSV in order to make it more easily usable in Spark. Also, the records after 2008 have a different schema than prior years, so after converting the shapefiles to CSV, they’ll need to be ingested into Spark using separate user-defined schemas.

The following Python code will reformat shapefiles into CSV:

```
%python
import csv
from dbfpy import dbf
import os
import sys
DATADIR='data/fires/'

for filename in os.listdir(DATADIR):

    if filename.endswith('.dbf'):
        print "Converting %s to csv" % filename
        csv_fn = DATADIR+filename[:-4]+ ".csv"
        with open(csv_fn, 'wb') as csvfile:
            in_db = dbf.Dbf(DATADIR+filename)
            out_csv = csv.writer(csvfile)
            names = []
            for field in in_db.header.fields:
                names.append(field.name)
            out_csv.writerow(names)
            for rec in in_db:
                out_csv.writerow(rec.fieldData)
            in_db.close()
            print "Done..."
```

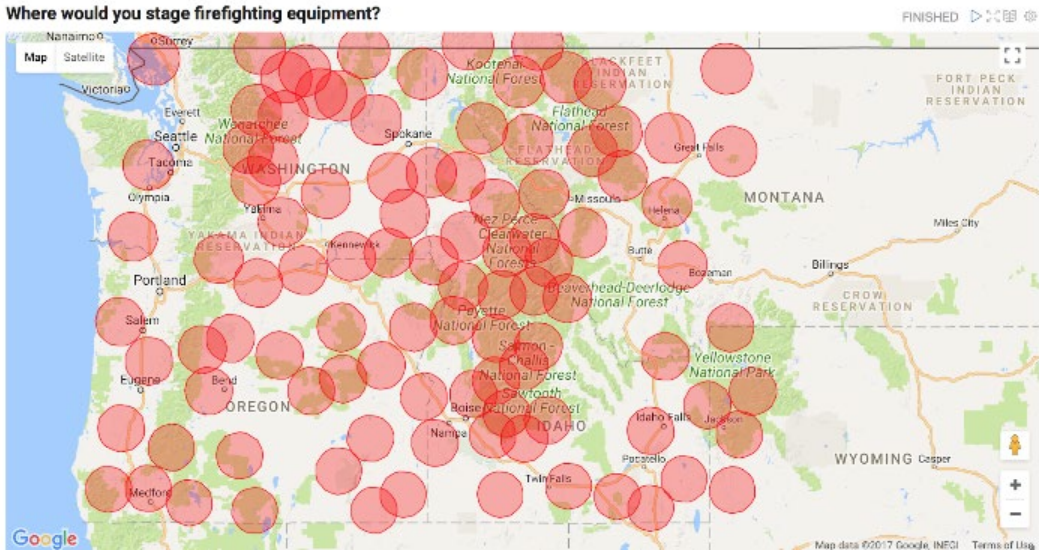
The following Spark Scala code will ingest the CSV files and train a *k*-means model with Spark libraries:

```
import org.apache.spark._
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.types._
import org.apache.spark.sql.functions._
import org.apache.spark.sql._
import org.apache.spark._
import org.apache.spark.ml.feature.StringIndexer
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.clustering.KMeans
import org.apache.spark.ml.clustering.KMeansModel
import org.apache.spark.mllib.linalg.Vectors
import sqlContext.implicits._
import sqlContext._

val schema = StructType(Array(
  StructField("area", DoubleType, true),
  StructField("perimeter", DoubleType, true),
  StructField("firenum", DoubleType, true),
  StructField("fire_id", DoubleType, true),
  StructField("lat", DoubleType, true),
  StructField("lon", DoubleType, true),
  StructField("date", TimestampType, true),
  StructField("julian", IntegerType, true),
  StructField("gmt", IntegerType, true),
  StructField("temp", DoubleType, true),
  StructField("spix", DoubleType, true),
  StructField("tpix", DoubleType, true),
  StructField("src", StringType, true),
  StructField("sat_src", StringType, true),
  StructField("conf", IntegerType, true),
  StructField("frp", DoubleType, true)
))

val df_all = sqlContext.read.format("com.databricks.spark.csv").
  option("header", "true").schema(schema).load("data/fires/modis*.csv")
// Include only fires with coordinates in the Pacific Northwest
val df = df_all.filter($"lat" > 42).filter($"lat" < 50).filter($"lon" >
-124).filter($"lon" < -110)
val featureCols = Array("lat", "lon")
val assembler = new VectorAssembler().setInputCols(featureCols).
  setOutputCol("features")
val df2 = assembler.transform(df)
val Array(trainingData, testData) = df2.randomSplit(Array(0.7, 0.3), 5043)
val kmeans = new KMeans().setK(400).setFeaturesCol("features").setMaxIter(5)
val model = kmeans.fit(trainingData)
println("Final Centers: ")
model.clusterCenters.foreach(println)
// Save the model to disk
model.write.overwrite().save("data/save_fire_model")
```

The resulting cluster centers are shown below. Where would you stage fire fighting equipment?



These centroids were calculated by analyzing the locations for fires that have occurred in the past. These points can be used to help stage fire fighting equipment as near as possible to regions prone to burn, but how do we know which staging area should respond when a new forest fire starts? We can use our previously saved model to answer that question. The Scala code for that would look like this:

```
val test_coordinate = Seq((42.3, -112.2)).toDF("latitude",
"longitude")
val df3 = assembler.transform(test_coordinate)
val categories = model.transform(df3)
val centroid_id = categories.select("prediction").rdd.map(r =>
r(0)).collect()(0).asInstanceOf[Int]
println(model.clusterCenters(centroid_id))
```

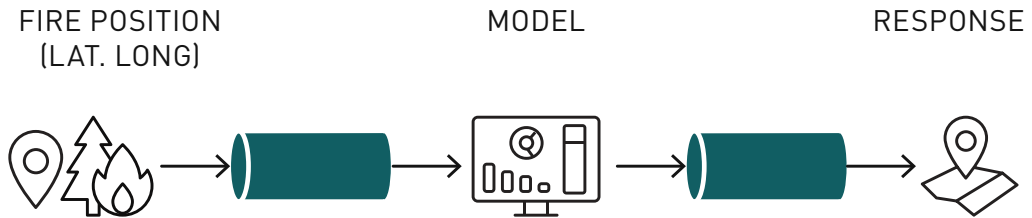

Data scientists have embraced web-based notebooks, such as Apache Zeppelin, because they allow you to interactively transform and visually explore datasets without the burden of compiling and running a full program. To view the processes described above for data wrangling, *k*-means modeling, and centroid visualization on a map, see the following Zeppelin notebook:



<https://www.zepl.com/viewer/notebooks/bm90ZTovL2lhbmRvdy8zNmNmU10DJkNGQ0ZWVmYjg5MzI4MzUzYTBJNmViYi9ub3RlLmpzb24>

Operationalizing This Model as a Real-Time “Fire Response” App

The previous code excerpt shows how the model we developed could be used to identify which fire station (i.e., centroid) should be assigned to a given wildfire. We could operationalize this as a real-time fire response application with the following ML pipeline:



Streams ensure requests and responses are saved, replicated, and replayable.

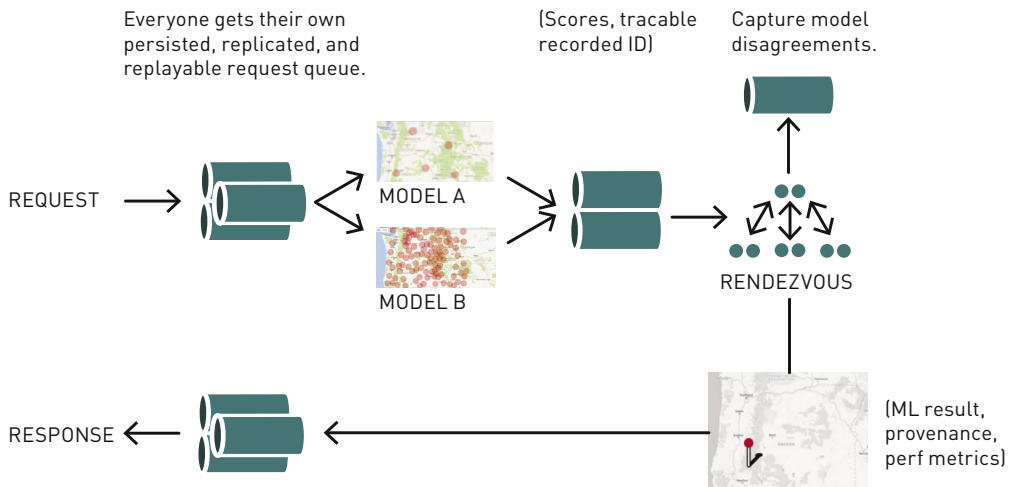
Most machine learning applications are initially architected with a synchronous pipeline like the one shown above, but there are limitations to this simplistic approach. Since it is only architected for a single model, your options are limited when it comes to the following:

- How do you A/B test different versions of your model?
- How do you load balance inference requests?
- How do you process inference requests with multiple models optimized for different objectives (e.g., speed versus accuracy)?

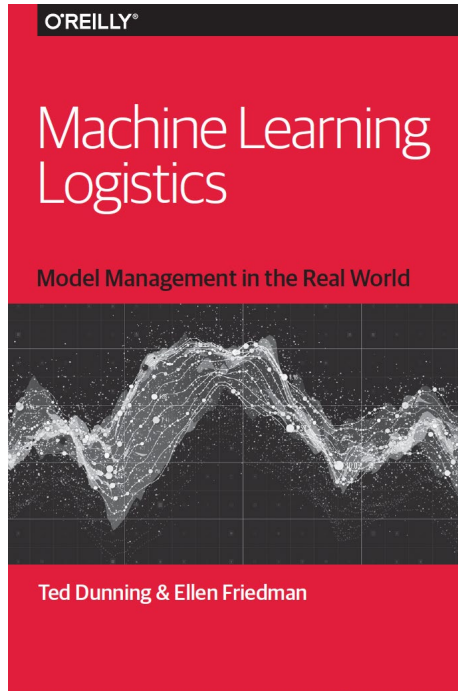
In order to do these things, the model must be a modular component in the pipeline, and model results should rendezvous at a point where they can be compared, monitored, and selected based upon user-defined criteria. This design pattern can be achieved with an architecture called the rendezvous architecture.

The Rendezvous Architecture

The rendezvous architecture is a machine learning pipeline that allows multiple models to process inference requests and rendezvous at a point where user-defined logic can be applied to choose which ML result to return to the requester. Such logic could say, “Give me the fastest result,” or “give me the highest confidence score after waiting 10 seconds.” The rendezvous point also gives us a point where models can be monitored and requests can be captured when model results significantly disagree with each other.



Note the emphasis on streams. Streams buffer requests in an infinite, resilient, and replayable queue. This makes it easy to hot swap models and scale ML executors in a microservices fashion. It also guarantees traceability for every inference request and response.



If you'd like to learn more about the rendezvous architecture, read the highly recommended *Machine Learning Logistics* by Ted Dunning and Ellen Friedman, which is available as a [free downloadable ebook](#).

Conclusion

This was a story about how I used geolocation data with k -means clustering that relates to a topic which deeply affects a lot of people – wildfires! Anytime you have lat/long coordinates, you have an opportunity to do data science with k -means clustering and visualization on a map. I hope this example illustrates the basics of k -means clustering and also gives some perspective on how machine learning models can be operationalized in production scenarios using streaming interfaces.

Using Apache Spark GraphFrames to Analyze Flight Delays and Distances

This chapter will help you get started using Apache Spark GraphFrames. We will begin with an overview of Graph and GraphFrames concepts, then we will analyze the flight dataset from previous chapters for flight distances and delays.

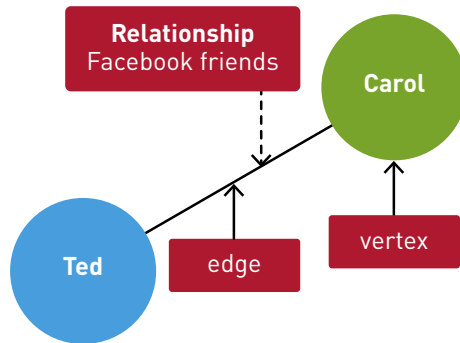
Graphs provide a powerful way to analyze the connections in a Dataset. GraphX is the Apache Spark component for graph-parallel and data-parallel computations, built upon a branch of mathematics called graph theory. It is a distributed graph processing framework that sits on top of the Spark core. GraphX brings the speed and scalability of parallel, iterative processing to graphs for big datasets. It partitions graphs that are too large to fit in the memory of a single computer among multiple computers in a cluster. In addition, GraphX partitions vertices independently of edges, which avoids the load imbalance often suffered when putting all the edges for a vertex onto a single machine.

GraphFrames extends Spark GraphX to provide the DataFrame API, making the analysis easier to use and, for some queries, more efficient with the Spark SQL Catalyst optimizer.

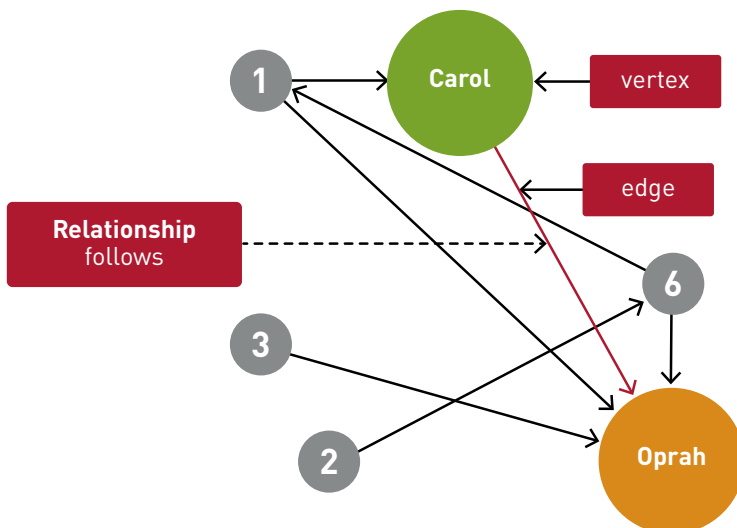
Overview of Some Graph Concepts

A graph is a mathematical structure used to model relations between objects. A graph is made up of vertices and edges that connect them. The vertices are the objects, and the edges are the relationships between them.

A **regular graph** is a graph where each vertex has the same number of edges. An example of a regular graph is Facebook friends. If Ted is a friend of Carol, then Carol is also a friend of Ted.



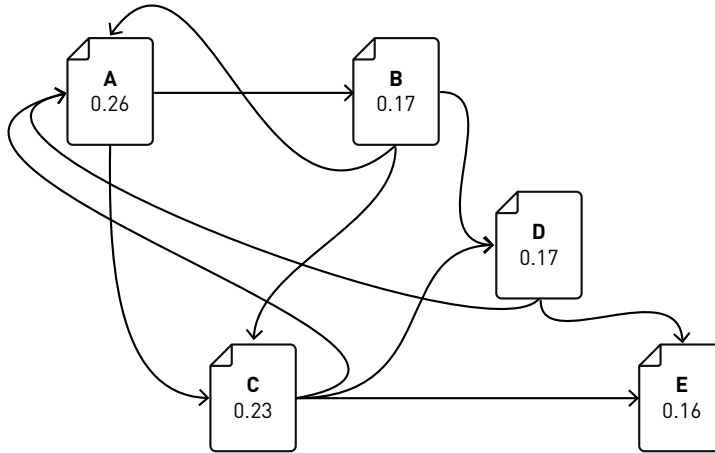
A **directed graph** is a graph where the edges have a direction associated with them. An example of a directed graph is a Twitter follower. Carol can follow Oprah without implying that Oprah follows Carol.



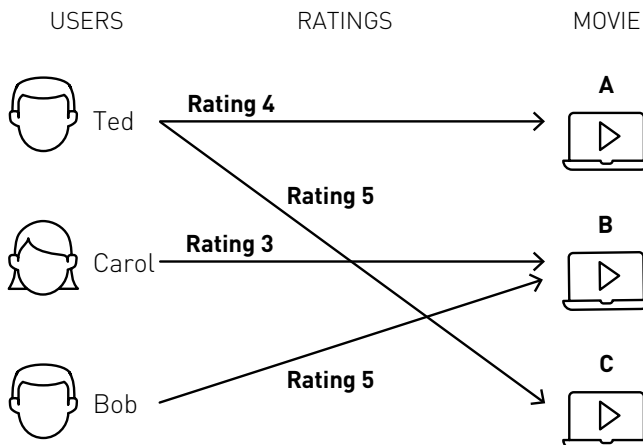
Graph Examples

Examples of connected data that can be represented by graphs include:

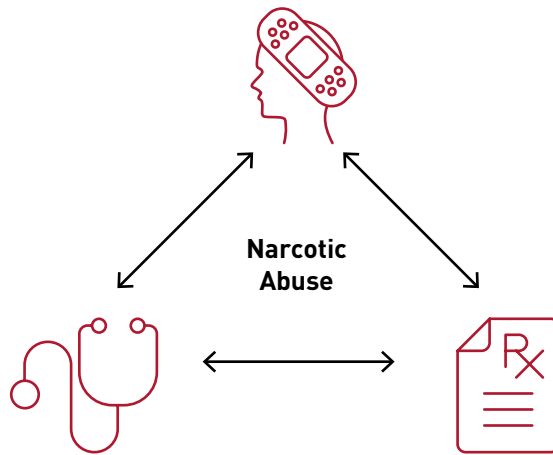
Websites: The breakthrough for the creators of the Google search engine was to create the PageRank graph algorithm, which represents pages as nodes and links as edges and measures the importance of each page by the number of links to a page and the number of links to each of the linking pages.



Recommendation Engines: Recommendation algorithms can use graphs where the nodes are the users and products, and their respective attributes and the edges are the ratings or purchases of the products by users. Graph algorithms can calculate weights for how similar users rated or purchased similar products.



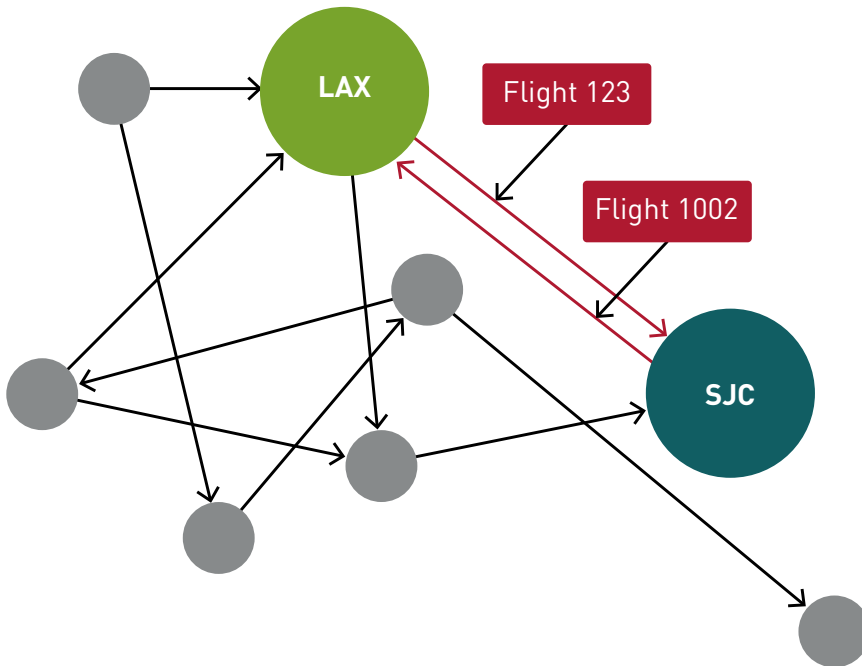
Fraud: Graphs are useful for fraud detection algorithms in banking, healthcare, and network security. In healthcare, graph algorithms can explore the connections between patients, doctors, and pharmacy prescriptions. In banking, graph algorithms can explore the relationship between credit card applicants and phone numbers and addresses or between credit cards customers and merchant transactions. In network security, graph algorithms can explore data breaches.



These are just some examples of the uses of graphs. Next, we will look at a specific example, using Spark GraphFrames.

GraphFrame Property Graph

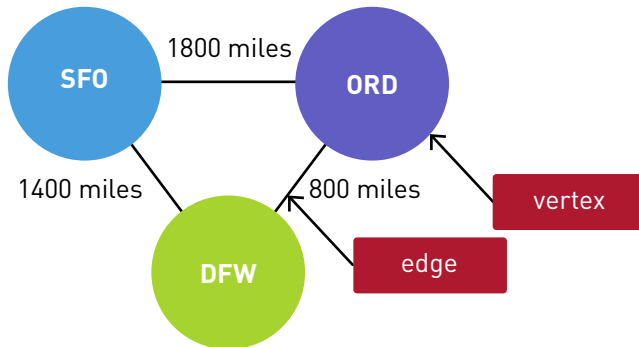
Spark GraphX supports graph computation with a distributed property graph. A property graph is a directed multigraph, which can have multiple edges in parallel. Every edge and vertex has user defined properties associated with it. The parallel edges allow multiple relationships between the same vertices.



With GraphFrames, vertices and edges are represented as DataFrames, which adds the advantages of querying with Spark SQL and support for DataFrame data sources like Parquet, JSON, and CSV.

Example Flight Scenario

As a starting simple example, we will analyze 3 flights; for each flight, we have the following information:



Originating Airport	Destination Airport	Distance
SFO	ORD	1800 miles
ORD	DFW	800 miles
DFW	SFO	1400 miles

In this scenario, we are going to represent the airports as vertices and flight routes as edges. For our graph, we will have three vertices, each representing an airport. The vertices each have the airport code as the ID, and the city as a property:

Vertex Table for Airports

id	city
SFO	San Francisco
ORD	Chicago
DFW	Texas

The edges have the Source ID, the Destination ID, and the distance as a property.

Edges Table for Routes

src	dst	distance	delay
SFO	ORD	1800	40
ORD	DFW	800	0
DFW	SFO	1400	10

Launching the Spark Interactive Shell with GraphFrames

Because GraphFrames is a separate package from Spark, start the Spark shell, specifying the GraphFrames package as shown below:

```
$SPARK_HOME/bin/spark-shell --packages
graphframes:graphframes:0.6.0-spark2.3-s_2.11
```

Define Vertices

First, we will import the DataFrames, GraphX, and GraphFrames packages.

```
import org.apache.spark._
import org.apache.spark.graphx._
import org.apache.spark.sql._
import org.apache.spark.sql.functions._
import org.apache.spark.sql.types._
import org.apache.spark.sql.types.StructType
import org.graphframes._
import spark.implicits._
```

We define airports as vertices. A vertex DataFrame must have an ID column and may have multiple attribute columns. In this example, each airport vertex consists of:

- Vertex ID → id
- Vertex Property → city

Vertex Table for Airports

id	city
SFO	San Francisco

We define a `DataFrame` with the above properties, which will be used for the vertices in the `GraphFrame`.

```
// create vertices with ID and Name
case class Airport(id: String, city: String) extends
  Serializable

val airports=Array(Airport("SFO","San
Francisco"),Airport("ORD","Chicago"),Airport("DFW","Dallas Fort
Worth"))

val vertices = spark.createDataset(airports).toDF
vertices.show
```

result:

```
+---+-----+
| id|          city|
+---+-----+
|SFO|    San Francisco|
|ORD|         Chicago|
|DFW|Dallas Fort Worth|
+---+-----+
```

Define Edges

Edges are the flights between airports. An edge `DataFrame` must have `src` and `dst` columns and may have multiple relationship columns. In our example, an edge consists of:

- Edge origin ID → `src`
- Edge destination ID → `dst`
- Edge property distance → `dist`
- Edge property delay → `delay`

Edges Table for Flights

id	src	src	src	delay
AA_2017-01-01_SFO_ORD	SFO	SFO	SFO	40

We define a `DataFrame` with the above properties, which will be used for the edges in the `GraphFrame`.

```
// create flights with srcid, destid, distance
case class Flight(id: String, src: String, dst: String, dist:
Double, delay: Double)

val flights=Array(Flight("SFO_ORD_2017-01-01_AA","SFO","ORD",1800,
40),Flight("ORD_DFW_2017-01-01_UA","ORD","DFW",800,
0),Flight("DFW_SFO_2017-01-01_DL","DFW","SFO",1400, 10))

val edges = spark.createDataset(flights).toDF
edges.show
```

result:

id	src	dst	dist	delay
SFO_ORD_2017-01-01-AA	SFO	ORD	1800.0	40.0
ORD_DFW_2017-01-01-UA	ORD	DFW	800.0	0.0
DFW_SFO_2017-01-01-DL	DFW	SFO	1400.0	10.0

Create the GraphFrame

Below, we create a GraphFrame by supplying a vertex DataFrame and an edge DataFrame. It is also possible to create a GraphFrame with just an edge DataFrame; then the vertices will be inferred.

```
// define the graph
val graph = GraphFrame(vertices, edges)

// show graph vertices
graph.vertices.show
```

id	name
SFO	San Francisco
ORD	Chicago
DFW	Dallas Fort Worth

```
// show graph edges
graph.edges.show
```

result:

	id	src	dst	dist	delay
SFO_ORD_2017-01-0...	SFO	ORD	1800.0	40.0	
ORD_DFW_2017-01-0...	ORD	DFW	800.0	0.0	
DFW_SFO_2017-01-0...	DFW	SFO	1400.0	10.0	

Querying the GraphFrame

Now we can query the GraphFrame to answer the following questions:

How many airports are there?

```
// How many airports?
graph.vertices.count

result: Long = 3
```

How many flights are there between airports?

```
// How many flights?
graph.edges.count

result: = 3
```

Which flight routes are greater than 1000 miles in distance?

```
// routes > 1000 miles distance?
graph.edges.filter("dist > 800").show

+-----+-----+-----+-----+
|          id|src|dst|  dist|delay|
+-----+-----+-----+-----+
|SFO_ORD_2017-01-0...|SFO|ORD|1800.0| 40.0|
|DFW_SFO_2017-01-0...|DFW|SFO|1400.0| 10.0|
+-----+-----+-----+-----+
```

The GraphFrames triplets put all of the edge, src, and dst columns together in a DataFrame.

```
// triplets
graph.triplets.show

result:
```

src	edge	dst
[SFO, San Francisco]	[SFO_ORD_2017-01-...	[ORD, Chicago]
[ORD, Chicago]	[ORD_DFW_2017-01-...	[DFW, Dallas Fort ...]
[DFW, Dallas Fort ...]	[DFW_SFO_2017-01-...	[SFO, San Francisco]

What are the longest distance routes?

```
// print out longest routes
graph.edges
  .groupBy("src", "dst")
  .max("dist")
  .sort(desc("max(dist)")).show
```

src	dst	max(dist)
SFO	ORD	1800.0
DFW	SFO	1400.0
ORD	DFW	800.0

Analyze Real Flight Data with GraphFrames Scenario

Now, we will analyze flight delays and distances, using the real flight data that we explored in [chapter 2](#). For each airport, we have the following information:

Vertex Table for Airports

id	city	state
SFO	San Francisco	CA

For each flight, we have the following information

Edges Table for Flights

id	src	dst	dist	dist
AA_2017-01-01_SFO_ORD	SFO	ORD	1800	1800

Again, in this scenario, we are going to represent the airports as vertices and flights as edges. We are interested in analyzing airports and flights to determine the busiest airports, their flight delays, and distances.

First, we will import the needed packages.

```
import org.apache.spark._
import org.apache.spark.graphx._
import org.apache.spark.sql._
import org.apache.spark.sql.functions._
import org.apache.spark.sql.types._
import org.apache.spark.sql.types.StructType
import org.graphframes._
import spark.implicits._
```


Below, we define the flight schema, corresponding to the JSON data file.

```
// define the Flight Schema
val schema = StructType(Array(
  StructField("_id", StringType, true),
  StructField("dofW", IntegerType, true),
  StructField("carrier", StringType, true),
  StructField("origin", StringType, true),
  StructField("dest", StringType, true),
  StructField("crsdephour", IntegerType, true),
  StructField("crsdeptime", DoubleType, true),
  StructField("depdelay", DoubleType, true),
  StructField("crsarrrtime", DoubleType, true),
  StructField("arrdelay", DoubleType, true),
  StructField("crselapsedtime", DoubleType, true),
  StructField("dist", DoubleType, true)
))

case class Flight(_id: String, dofW: Integer, carrier: String,
origin: String, dest: String, crsdephour: Integer, crsdeptime:
Double, depdelay: Double, crsarrrtime: Double, arrdelay: Double,
crselapsedtime: Double, dist: Double) extends Serializable
```

Define Edges

Edges are the flights between airports. An edge must have src and dst columns and can have multiple relationship columns. In our example, an edge consists of:

id	src	dst	dist	delay	carrier	crsdephour
AA_2017-01-01_SFO_ORD	SFO	ORD	1800	40	AA	17

Below, we load the flights data from a JSON file into a DataFrame. Then we select the columns that we will use for the flight edge DataFrame. The required column names are `id`, `src`, and `dst`, so we rename those columns in the select statement.

```
var file = "maprfs:///data/flights20170102.json"

val df = spark.read.option("inferSchema", "false").
  schema(schema).json(file).as[Flight]

val flights = df.withColumnRenamed("_id", "id")
  .withColumnRenamed("origin", "src")
  .withColumnRenamed("dest", "dst")
  .withColumnRenamed("depdelay", "delay")

flights.show
```

result:

id	src	dst	delay	dist	carrier	crsdephour
ATL_BOS_2017-01-0...	ATL	BOS	30.0	946.0	DL	9
ATL_BOS_2017-01-0...	ATL	BOS	0.0	946.0	DL	11
ATL_BOS_2017-01-0...	ATL	BOS	0.0	946.0	WN	13

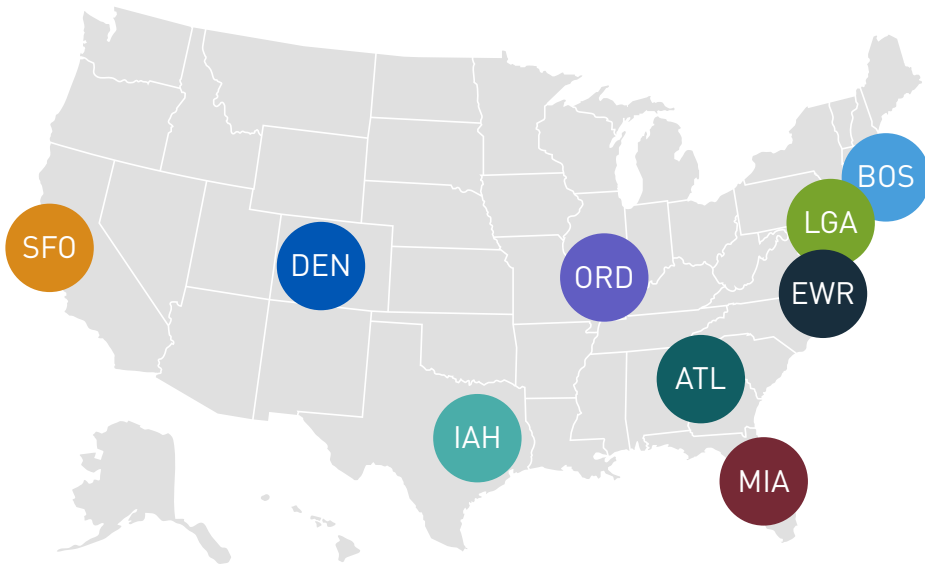
Define Vertices

We define airports as vertices. Vertices can have properties or attributes associated with them. For each airport, we have the following information:

Vertex Table for Airports

id	city	state
SFO	San Francisco	CA

Note that our dataset contains only a subset of the airports in the USA; below are the airports in our dataset shown on a map.



Below, we read the airports information into a DataFrame from a JSON file.

```
// create airports DataFrame

val airports = spark.read.json("maprfs:///data/airports.json")
airports.createOrReplaceTempView("airports")
airports.show
```

result:

```
+-----+-----+-----+-----+
|          City|Country|State| id|
+-----+-----+-----+-----+
|      Houston|      USA|   TX| IAH|
|    New York|      USA|   NY| LGA|
|      Boston|      USA|   MA| BOS|
|      Newark|      USA|   NJ| EWR|
|      Denver|      USA|   CO| DEN|
|      Miami|      USA|   FL| MIA|
|San Francisco|      USA|   CA| SFO|
|      Atlanta|      USA|   GA| ATL|
|      Chicago|      USA|   IL| ORD|
+-----+-----+-----+-----+
```

Create the Property Graph

Below, we create a GraphFrame by supplying a vertex DataFrame and an edge DataFrame.

```
// define the graphframe
val graph = GraphFrame(airports, flights)

// show graph vertices
graph.vertices.show(2)
```

result:

City	Country	State	id
Houston	USA	TX	IAH
New York	USA	NY	LGA

```
// show graph edges
graph.edges.show(2)
```

result:

id	src	dst	delay	dist	carrier	crsdephour
ATL_BOS_2017-01-0...	ATL	BOS	30.0	946.0	DL	9
ATL_BOS_2017-01-0...	ATL	BOS	0.0	946.0	DL	11

Querying the GraphFrame

Now we can query the GraphFrame to answer the following questions:

How many airports are there?

```
// How many airports?
val numairports = graph.vertices.count
```

result:

```
Long = 9
```

How many flights are there?

```
// How many flights?
val numflights = graph.edges.count

result:
// Long = 41348
```

Which flight routes have the longest distance?

```
// show the longest distance routes
graph.edges
  .groupBy("src", "dst")
  .max("dist")
  .sort(desc("max(dist)")).show(4)

result:
+---+---+-----+
|src|dst|max(dist)|
+---+---+-----+
|SFO|BOS|    2704.0|
|BOS|SFO|    2704.0|
|SFO|MIA|    2585.0|
|MIA|SFO|    2585.0|
+---+---+-----+
```

Which flight routes have the highest average delays?

```
graph.edges
  .groupBy("src", "dst")
  .avg("delay")
  .sort(desc("avg(delay)")).show(5)
```

result:

```
+---+---+-----+
|src|dst|      avg(delay) |
+---+---+-----+
|ATL|SFO|      33.505 |
|MIA|SFO| 32.30797101449275 |
|SFO|BOS| 26.77319587628866 |
|DEN|SFO|      26.45375 |
|IAH|SFO| 26.002141327623125 |
+---+---+-----+
```

Which flight hours have the highest average delays?

```
graph.edges
  .groupBy("crsdephour")
  .avg("delay")
  .sort(desc("avg(delay)")).show(5)
```

result:

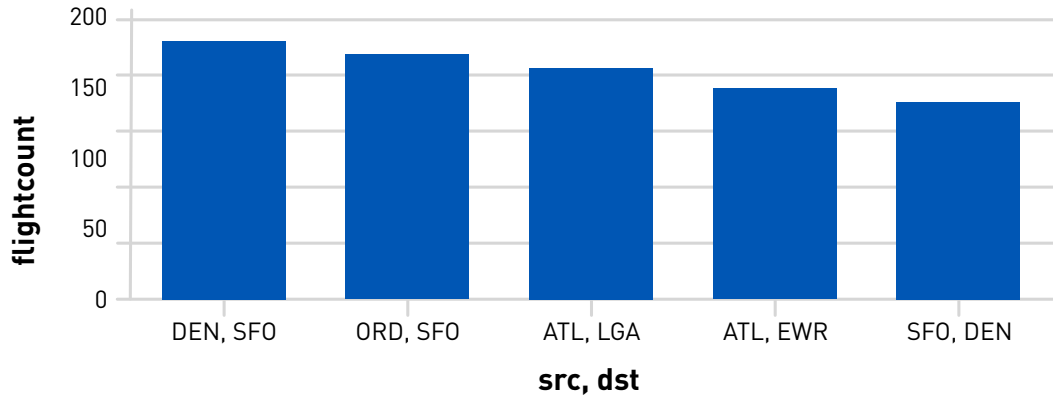
```
+-----+-----+
|crsdephour|      avg(delay) |
+-----+-----+
|      18| 24.24118415324336 |
|      19| 23.348782771535582 |
|      21| 19.617375231053604 |
|      16| 19.30346232179226 |
|      17| 18.77857142857143 |
+-----+-----+
```

Which flight routes have the most delays over 40 minutes?

```
graph.edges
.filter("` delay > 40`")
.groupBy("`src`, `dst`")
.agg(count("`delay`").as("`flightcount`"))
.sort(desc("`flightcount`")).show(5)
```

result:

```
+---+---+-----+
|src|dst|flightcount|
+---+---+-----+
|DEN|SFO|         172|
|ORD|SFO|         168|
|ATL|LGA|         155|
|ATL|EWR|         141|
|SFO|DEN|         134|
+---+---+-----+
```



What are the longest delays for flights that are greater than 1500 miles in distance?

```
// flights > 1500 miles distance ordered by delay

graph.edges.filter("dist > 1500")
.orderBy(desc("delay")).show(3)
```

result:

id	src	dst	delay	dist	carrier	crsdephour
SFO_ORD_2017-02-2...	SFO	ORD	1440.0	1846.0	AA	8
DEN_EWR_2017-02-2...	DEN	EWR	1138.0	1605.0	UA	12
DEN_LGA_2017-02-2...	DEN	LGA	1004.0	1620.0	DL	16

What are the worst hours for delayed flights departing from Atlanta?

```
graph.edges.filter("src = 'ATL' and delay > 1").
groupBy("crsdephour").avg("delay").sort(desc("avg(delay)")).
show(5)
```

result:

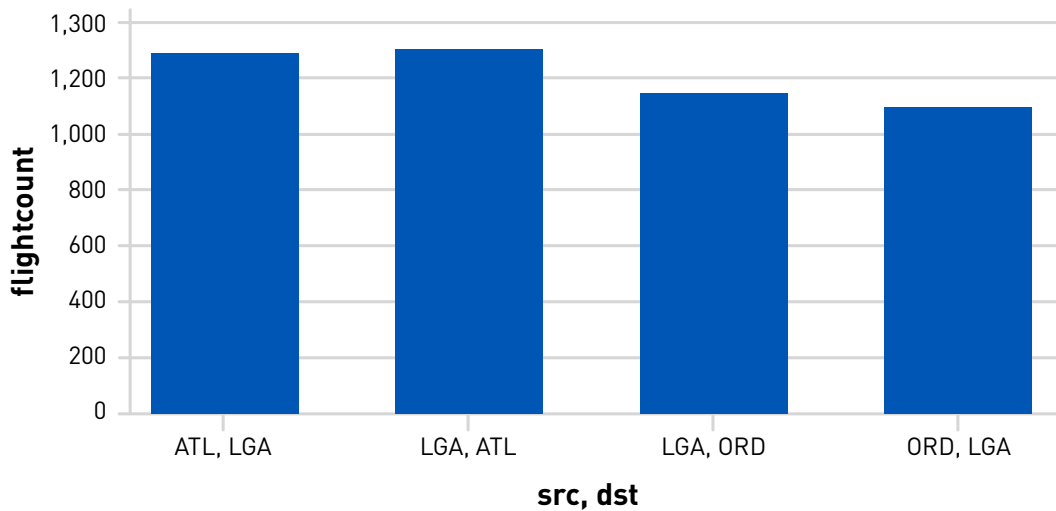
crsdephour	avg(delay)
19	60.15021459227468
20	56.816901408450704
18	55.5187969924812
22	48.61971830985915
17	47.5125

What are the four most frequent flight routes in the data set?

```
graph.edges.groupBy("src", "dst").count().orderBy(desc("count")).  
show(4)
```

result:

```
+---+---+-----+  
|src|dst|count|  
+---+---+-----+  
|ATL|LGA| 1271|  
|LGA|ATL| 1268|  
|LGA|ORD| 1107|  
|ORD|LGA| 1070|  
+---+---+-----+
```



Vertex Degrees

The degree of a vertex is the number of edges that touch the vertex. The degree of a graph vertex v of a graph G is the number of graph edges that touch v .

GraphFrames provides vertex `inDegree`, `outDegree`, and `degree` queries, which determine the number of incoming edges, outgoing edges, and total edges. Using GraphFrames degree queries, we can answer the following questions.

Which airports have the most incoming flights?

```
// get top 3
graph.inDegrees.orderBy(desc("inDegree")).show(3)
```

id	inDegree
ORD	6212
ATL	6012
LGA	4974

Which airports have the most outgoing flights?

```
// which airport has the most outgoing flights?
graph.outDegrees.orderBy(desc("outDegree")).show(3)
```

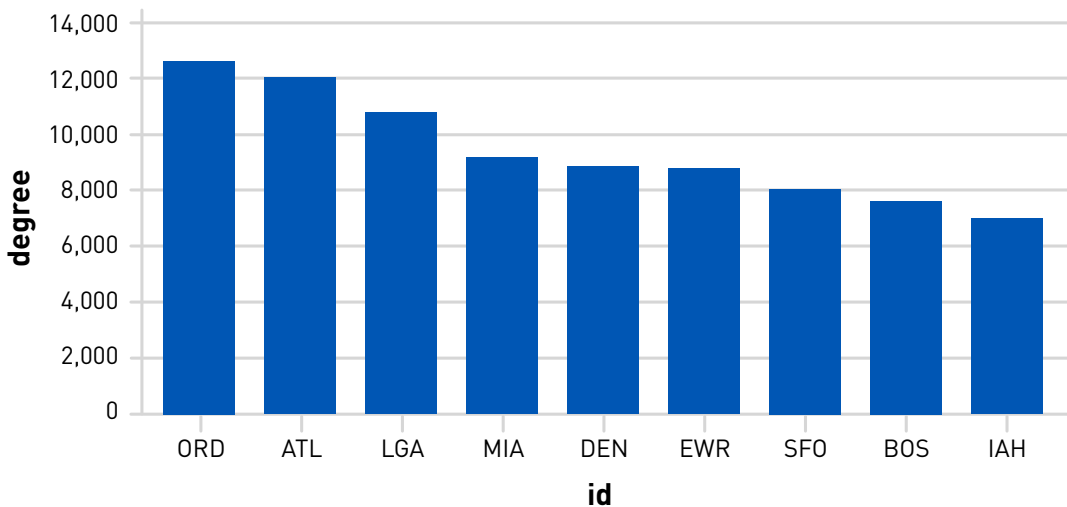
id	outDegree
ORD	6300
ATL	5971
LGA	4992

Which airports have the most incoming and outgoing flights?

```
// Define a reduce operation to compute the highest degree vertex
graph.degrees.orderBy(desc("degree")).show()
```

result:

```
+---+-----+
| id|degree|
+---+-----+
|ORD| 12512|
|ATL| 11983|
|LGA|  9966|
|MIA|  8864|
|DEN|  8486|
|EWR|  8428|
|SFO|  7623|
|BOS|  7423|
|IAH|  7411|
+---+-----+
```



PageRank

Another GraphFrames query is PageRank, which is based on the Google PageRank algorithm. PageRank measures the importance of each vertex in a graph, by determining which vertices have the most edges with other vertices. In our example, we can use PageRank to determine which airports are the most important, by measuring which airports have the most connections to other airports. We have to specify the probability tolerance, which is the measure of convergence.

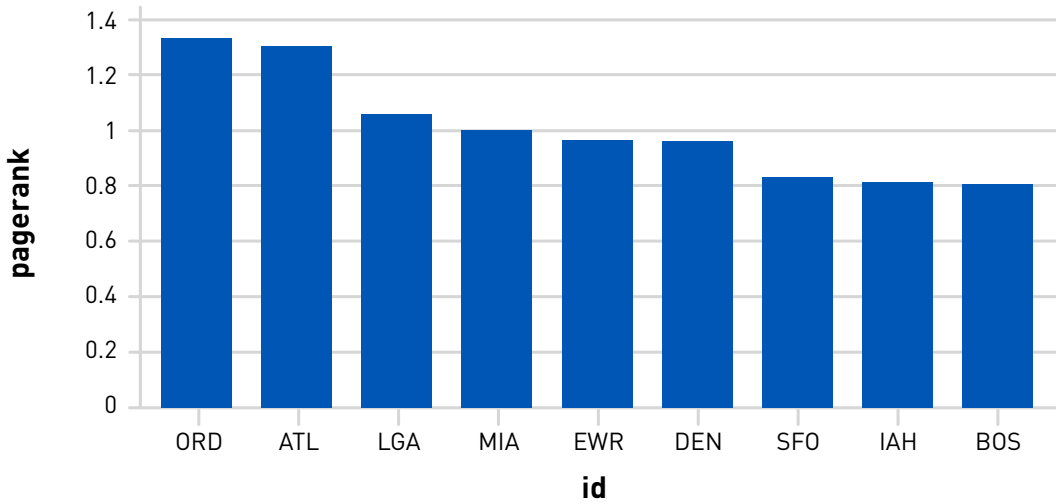
What are the most important airports, according to PageRank?

```
// use pageRank
val ranks = graph.pageRank.resetProbability(0.15).maxIter(10).
run()
```

```
ranks.vertices.orderBy($"pagerank".desc).show()
```

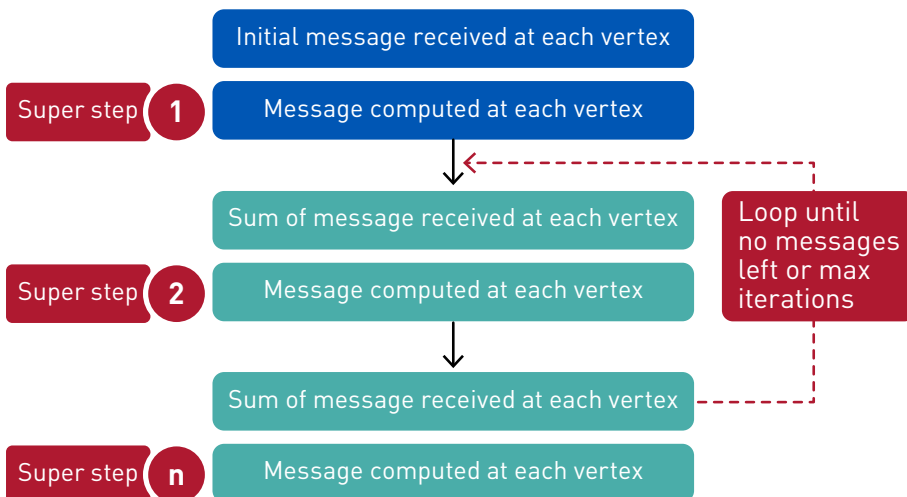
result:

City	Country	State	id	pagerank
Chicago	USA	IL	ORD	1.3093391746898806
Atlanta	USA	GA	ATL	1.2643315314643224
New York	USA	NY	LGA	1.0647854334640885
Miami	USA	FL	MIA	0.9682949340363207
Newark	USA	NJ	EWR	0.9322291015836434
Denver	USA	CO	DEN	0.9318184799701961
San Francisco	USA	CA	SFO	0.8558792499530605
Houston	USA	TX	IAH	0.8397625821927315
Boston	USA	MA	BOS	0.8335595126457567



Message Passing via AggregateMessages

Many important graph algorithms are iterative algorithms, since properties of vertices depend on properties of their neighbors, which depend on properties of *their* neighbors. Pregel is an iterative graph processing model, developed at Google, which uses a sequence of iterations of messages passing between vertices in a graph. GraphFrames provides `aggregateMessages`, which implements an aggregation message-passing API, based on the Pregel model. GraphFrames `aggregateMessages` sends messages between vertices and aggregates message values from the neighboring edges and vertices of each vertex.



The code below shows how to use `aggregateMessages` to compute the average flight delay by the originating airport. The flight delay for each flight is sent to the `src` vertex, then the average is calculated for the vertices.

```
import org.graphframes.lib.AggregateMessages

val AM = AggregateMessages
val msgToSrc = AM.edge("delay")
val agg = { graph.aggregateMessages
  .sendToSrc(msgToSrc)
  .agg(avg(AM.msg).as("avgdelay"))
  .orderBy(desc("avgdelay"))
  .limit(5) }
agg.show()
```

result:

```
+---+-----+
| id|          avgdelay|
+---+-----+
|SFO|20.306176084099867|
|EWR|16.317373785257170|
|DEN|16.167720777699696|
|IAH|15.925946093111898|
|ORD|14.880476190476191|
+---+-----+
```

Summary

GraphFrames provides a scalable and easy way to query and process large graph datasets, which can be used to solve many types of analysis problems. In this chapter, we gave an overview of the GraphFrames graph processing APIs. We encourage you to try out GraphFrames in more depth on some of your own projects.

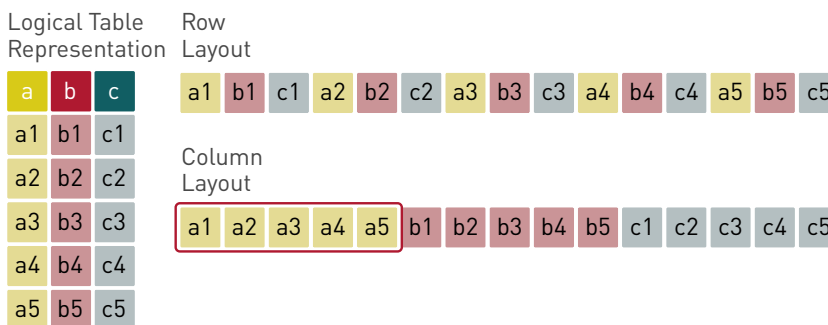
Tips and Best Practices to Take Advantage of Spark 2.x

With Apache Spark 2.0 and later versions, big improvements were implemented to enable Spark to execute faster, making lot of earlier tips and best practices obsolete. This chapter will first give a quick overview of what changes were made and then some tips to take advantage of these changes.

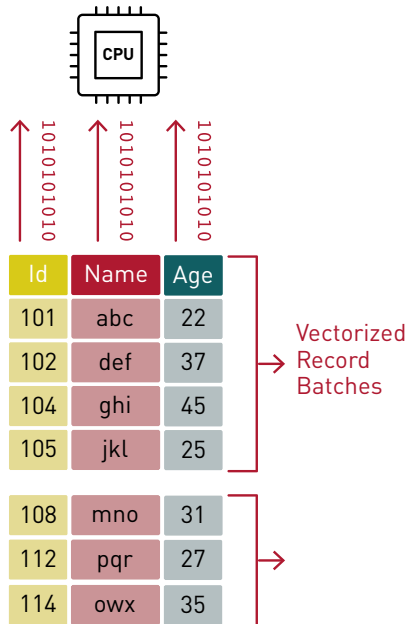
Project Tungsten

Tungsten is the code name for the Spark project that makes changes to Apache Spark's execution engine, focusing on improvements to the efficiency of memory and CPU usage. Tungsten builds upon ideas from modern compilers and massively parallel processing (MPP) technologies, such as [Apache Drill](#), [Presto](#), and [Apache Arrow](#). Spark 2.x improvements include:

- To reduce JVM object memory size, creation, and garbage collection processing, Spark explicitly manages memory and converts most operations to operate directly against binary data.
- [Columnar layout for memory data](#) avoids unnecessary I/O and accelerates analytical processing performance on modern CPUs and GPUs.



- Vectorization allows the CPU to operate on vectors, which are arrays of column values from multiple records. This takes advantage of modern CPU designs, by keeping all pipelines full to achieve efficiency.



- To improve the speed of data processing through more effective use of L1/ L2/L3 CPU caches, Spark algorithms and data structures exploit memory hierarchy with cache-aware computation.
- Spark SQL's Catalyst Optimizer underpins all the major new APIs in Spark 2.0 and later versions, from [DataFrames](#) and [Datasets](#) to Structured Streaming. The Catalyst optimizer handles: analysis, logical optimization, physical planning, and code generation to compile parts of queries to Java bytecode. Catalyst now supports both rule-based and cost-based optimization.

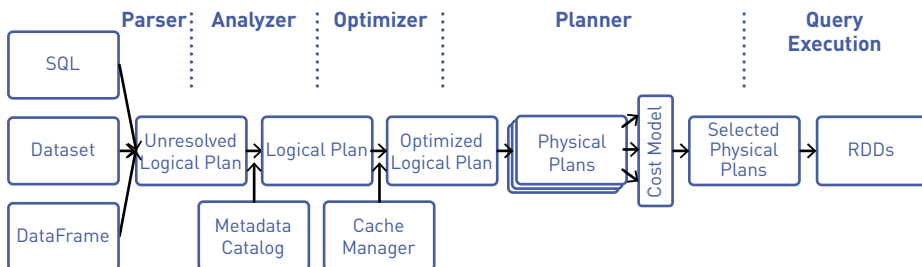


Image reference: Databricks

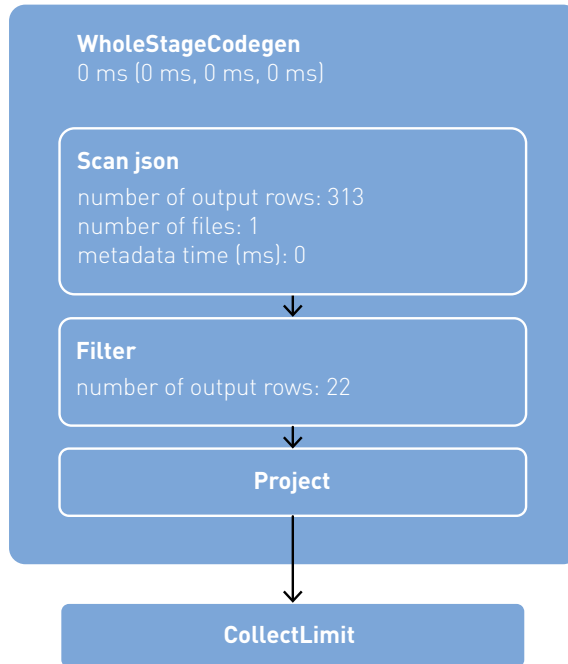
- Spark SQL “Whole-Stage Java Code Generation” optimizes CPU usage by generating a single optimized function in bytecode for the set of operators in a SQL query (when possible), instead of generating iterator code for each operator.

DETAILS FOR QUERY 0

Submitted Time: 2018/07/31 1:22:12

Duration: 0.9 s

Succeeded Jobs: 1



Tips for Taking Advantage of Spark 2.x Improvements

Use Dataset, DataFrames, Spark SQL

In order to take advantage of Spark 2.x, you should be using Datasets, DataFrames, and Spark SQL, instead of RDDs. Datasets, DataFrames and Spark SQL provide the following advantages:

- Compact columnar memory format
- Direct memory access
- Reduced garbage collection processing overhead
- Catalyst query optimization
- Whole-stage code generation

When possible, use Spark SQL functions – for example, `to_date()`, `hour()` – instead of custom UDFs in order to benefit from the advantages above.

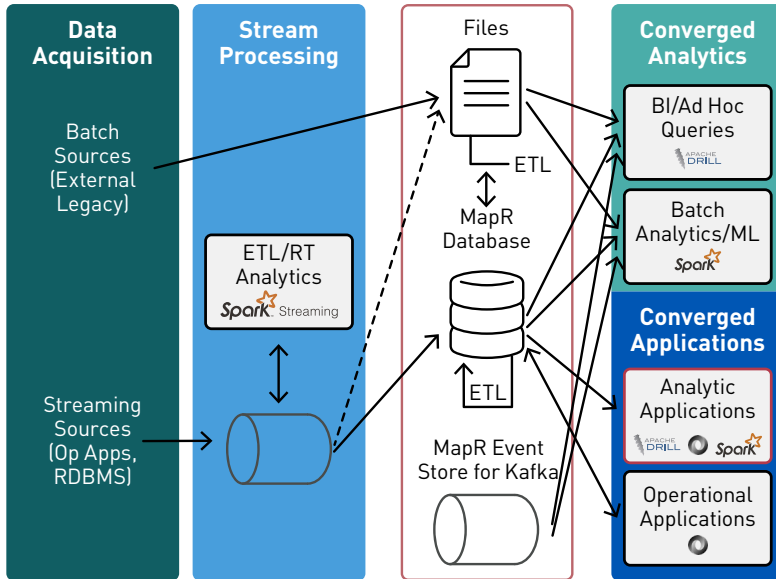
Datasets provide the advantage of compile time type safety over DataFrames. However, Dataset functional transformations (like `map`) will not take advantage of query optimization, whole-stage code generation, and reduced GC. To learn more about Datasets, DataFrames, and Spark SQL, refer to [chapters 2](#) and [3](#).

Use the Best Data Store for Your Use Case

Spark supports several data formats, including CSV, JSON, ORC, and Parquet, and several data sources or connectors, including distributed file stores such as MapR XD, Hadoop's HDFS, and Amazon's S3, popular NoSQL databases such as MapR Database and Apache HBase, and distributed messaging stores such as Apache Kafka and MapR Event Store for Kafka.

But just because Spark supports a given data storage or format doesn't mean you'll get the same performance with all of them. Typically, data pipelines will involve multiple data sources and sinks and multiple formats to support different use cases and different read/write latency requirements. Here are some guidelines:

- File data stores are good for write once (append only), read many use cases. CSV and JSON data formats give excellent write path performance but are slower for reading; these formats are good candidates for collecting raw data for example logs, which require high throughput writes. Parquet is slower for writing but gives the best performance for reading; this format is good for BI and analytics, which require low latency reads.
- Apache HBase and MapR Database are good for random read/write use cases. MapR Database supports consistent, predictable, high throughput, fast reads and writes with efficient updates, automatic partitioning, and sorting. MapR Database is multi-model: wide-column, key-value with the HBase API or JSON (document) with the OJAI API. MapR Database is good for real-time analytics on changing data use cases.
- Apache Kafka and MapR Event Store for Kafka are good for scalable reading and writing of real-time streaming data. MapR Event Store is good for data pipelines with stream-first architecture patterns and kappa or lambda architectures.



CSV and JSON Tips and Best Practices

Text File Formats

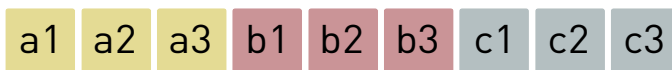
CSV
JSON

a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3

When persisting and compressing CSV and JSON files, make sure they are splittable, give high speeds, and yield reasonable compression. ZIP compression is not splittable, whereas Snappy is splittable; Snappy also gives reasonable compression with high speed. When reading CSV and JSON files, you will get better performance by specifying the schema, instead of using inference; specifying the schema reduces errors for data types and is recommended for production code. See chapter two for examples of specifying the schema on read.

Parquet Tips and Best Practices

Parquet Columnar Format

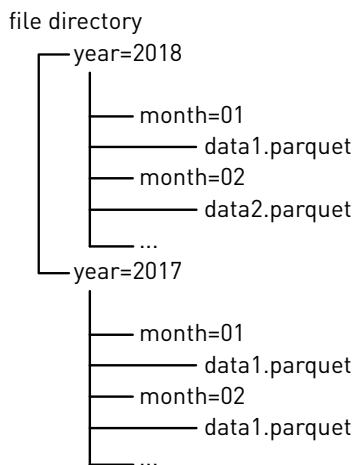


Apache Parquet gives the fastest read performance with Spark. Parquet arranges data in columns, putting related values in close proximity to each other to optimize query performance, minimize I/O, and facilitate compression. Parquet detects and encodes the same or similar data, using a technique that conserves resources. Parquet also stores column metadata and statistics, which can be pushed down to filter columns (discussed below). Spark 2.x has a vectorized Parquet reader that does decompression and decoding in column batches, providing ~ 10x faster read performance.

Parquet files are immutable; modifications require a rewrite of the dataset. For streaming data, you can stream to a fast read/write data store, such as MapR Database, then extract data to Parquet files for specific analytic use cases, or stream new datasets to a new partition (see partitioning below).

Parquet Partitioning

Spark table partitioning optimizes reads by storing files in a hierarchy of directories based on partitioning columns. For example, a directory structure could be organized by location, such as state/city, or by date, such as year/month, shown below:



DataFrames can be saved as persistent tables into a Hive metastore, using the [saveAsTable](#) command. If you do not have Hive setup, Spark will create a default local Hive metastore (using Derby). Persistent tables have several optimization benefits: partition and statistic metadata, and they can be bucketed (discussed later).

As an example with the flight dataset (used in chapters 2, 3, 5, and 9), a lot of queries about departure delays are organized around the originating airport (the `src` column), so this could make a good partitioning column. Here is a JSON row from this Dataset:

```
{
  "id": "ATL_LGA_2017-01-01_AA_1678",
  "dofW": 7,
  "carrier": "AA",
  "src": "ATL",
  "dst": "LGA",
  "crsdephour": 17,
  "crsdeptime": 1700,
  "depdelay": 0.0,
  "crsarrrtime": 1912,
  "arrdelay": 0.0,
  "crselapsedtime": 132.0,
  "dist": 762.0
}
```

Here is the code to persist a flights DataFrame as a table consisting of Parquet files partitioned by the `src` column:

```
df.write.format("parquet")
  .partitionBy("src")
  .option("path", "/user/mapr/data/flights")
  .saveAsTable("flights")
```

Below is the resulting directory structure as shown by a Hadoop list files command:

```
hadoop fs -ls /user/mapr/data/flights

/user/mapr/data/flights/src=ATL
/user/mapr/data/flights/src=BOS
/user/mapr/data/flights/src=CLT
/user/mapr/data/flights/src=DEN
/user/mapr/data/flights/src=DFW
/user/mapr/data/flights/src=EWR
/user/mapr/data/flights/src=IAH
/user/mapr/data/flights/src=LAX
/user/mapr/data/flights/src=LGA
/user/mapr/data/flights/src=MIA
/user/mapr/data/flights/src=ORD
/user/mapr/data/flights/src=SEA
/user/mapr/data/flights/src=SFO
```

Below, we see that the src=DEN subdirectory contains two Parquet files:

```
hadoop fs -ls /user/mapr/data/flights/src=DEN

/user/mapr/data/flights/src=DEN/part-00000-deb4a3d4-d8c3-4983-8756-
ad7e0b29e780.c000.snappy.parquet
/user/mapr/data/flights/src=DEN/part-00001-deb4a3d4-d8c3-4983-8756-
ad7e0b29e780.c000.snappy.parquet
```

Partition Pruning and Predicate Pushdown

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. When partition filters are present, the catalyst optimizer pushes down the partition filters. The scan reads only the directories that match the partition filters, thus reducing disk I/O. For example, the following query reads only the files in the src=DEN partition directory in order to query the average departure delay for flights originating from Denver.

```
df.filter("src = 'DEN' and depdelay > 1")
  .groupBy("src", "dst").avg("depdelay")
  .sort(desc("avg(depdelay)")).show()
```

result:

```
+---+---+-----+
|src|dst|      avg(depdelay) |
+---+---+-----+
|DEN|EWR|54.352020860495436|
|DEN|MIA| 48.95263157894737|
|DEN|SFO|47.189473684210526|
|DEN|ORD| 46.47721518987342|
|DEN|DFW|44.473118279569896|
|DEN|CLT|37.097744360902254|
|DEN|LAX|36.398936170212764|
|DEN|LGA| 34.594444444444444|
|DEN|BOS|33.633187772925766|
|DEN|IAH| 32.10775862068966|
|DEN|SEA|30.532345013477087|
|DEN|ATL| 29.29113924050633|
+---+---+-----+
```

Or in SQL:

```
%sql
select src, dst, avg(depdelay)
from flights where src='DEN' and depdelay > 1
group by src, dst
ORDER BY src
```

You can see the physical plan for a DataFrame query in the Spark web UI SQL tab (discussed in [chapter 3](#)) or by calling the explain method shown below. Here in red, we see partition filter push down, which means that the src=DEN filter is pushed down into the Parquet file scan. This minimizes the files and data scanned and reduces the amount of data passed back to the Spark engine for the aggregation average on the departure delay.


```

df.filter("src = 'DEN' and depdelay > 1")
.groupBy("src", "dst").avg("depdelay")
.sort(desc("avg(depdelay)")).explain

== Physical Plan ==
TakeOrderedAndProject(limit=1001, orderBy=[avg(depdelay)#304 DESC
NULLS LAST], output=[src#157,dst#149,avg(depdelay)#314])

+- *(2) HashAggregate(keys=[src#157, dst#149],
    functions=[avg(depdelay#152)],
    output=[src#157, dst#149, avg(depdelay)#304])

    +- Exchange hashpartitioning(src#157, dst#149, 200)

        +- *(1) HashAggregate(keys=[src#157, dst#149],
            functions=[partial_avg(depdelay#152)],
            output=[src#157, dst#149,
                sum#321, count#322L])

            +- *(1) Project [dst#149, depdelay#152, src#157]

                +- *(1) Filter (isnotnull(depdelay#152) && (depdelay#152 >
                    1.0))

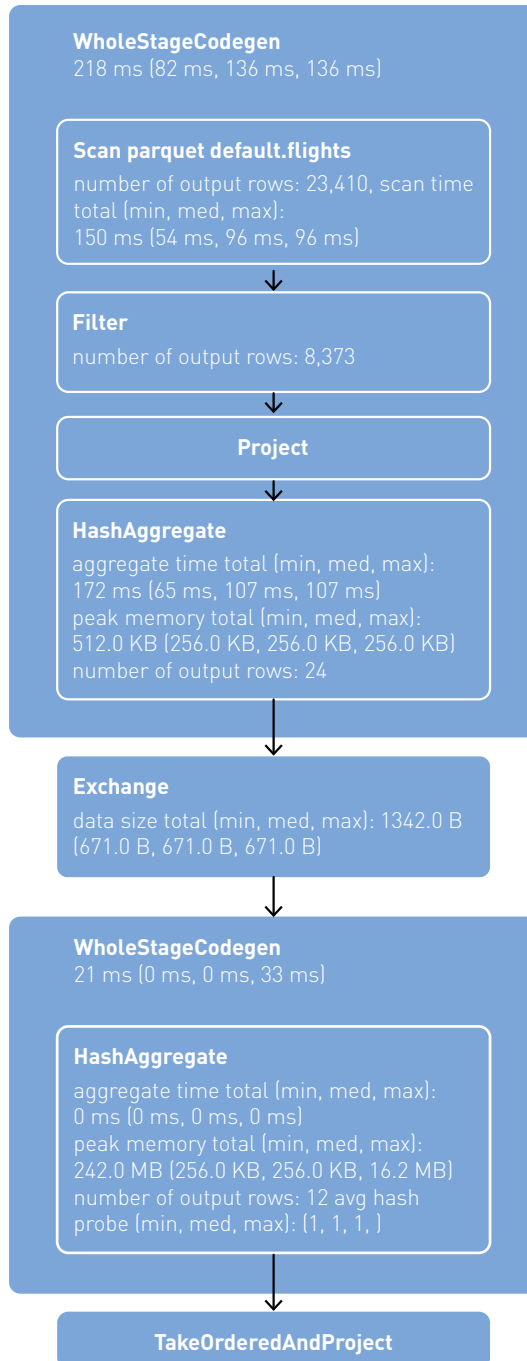
                    +- *(1) FileScan parquet default.
flights[dst#149,depdelay#152,src#157] Batched: true, Format:
Parquet, Location: PrunedInMemoryFileIndex[maprfs:/user/mapr/
data/flights/src=DEN], PartitionCount: 1, PartitionFilters:
[isnotnull(src#157), (src#157 = DEN)], PushedFilters:
[IsNotNull(depdelay), GreaterThan(depdelay,1.0)], ReadSchema:
struct<dst:string,depdelay:double>

```

The physical plan is read from the bottom up, whereas the DAG is read from the top down. Note: the Exchange means a shuffle occurred between stages.

Duration: 1 s

Succeeded Jobs: 7



Partitioning Tips

The partition columns should be used frequently in queries for filtering and should have a small range of values with enough corresponding data to distribute the files in the directories. You want to avoid too many small files, which make scans less efficient with excessive parallelism. You also want to avoid having too few large files, which can hurt parallelism.

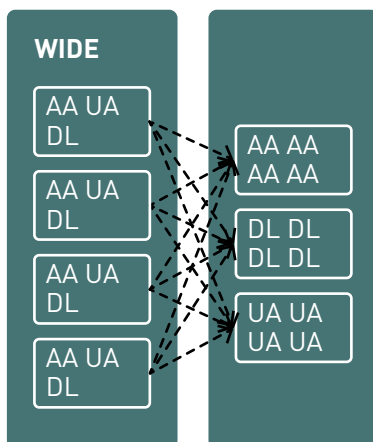
Coalesce and Repartition

Before or when writing a DataFrame, you can use `dataframe.coalesce(N)` to reduce the number of partitions in a DataFrame, without shuffling, or `df.repartition(N)` to reorder and either increase or decrease the number of partitions with shuffling data across the network to achieve even load balancing.

```
df.write.format("parquet")
.repartition(13)
.partitionBy("src")
.option("path", "/user/mapr/data/flights")
.saveAsTable("flights")
```

Bucketing

Bucketing is another data organization technique that groups data with the same bucket value across a fixed number of “buckets.” This can improve performance in wide transformations and joins by avoiding “shuffles.” Recall from [chapter 3](#), with wide transformation shuffles, data is sent across the network to other nodes and written to disk, causing network and disk I/O, and making the shuffle a costly operation. Below is a shuffle caused by a `df.groupBy("carrier").count`; if this dataset were bucketed by “carrier,” then the shuffle could be avoided.



Bucketing is similar to partitioning, but partitioning creates a directory for each partition, whereas bucketing distributes data across a fixed number of buckets by a hash on the bucket value. Tables can be bucketed on more than one value and bucketing can be used with or without partitioning.

As an example with the flight dataset, here is the code to persist a flights DataFrame as a table, consisting of Parquet files partitioned by the src column and bucketed by the dst and carrier columns (sorting by the id will sort by the src, dst, flightdate, and carrier, since that is what the id is made up of):

```
df.write.format("parquet")
  .sortBy("id")
  .partitionBy("src")
  .bucketBy(4, "dst", "carrier")
  .option("path", "/user/mapr/data/flightsbkdc")
  .saveAsTable("flightsbkdc")
```

The resulting directory structure is the same as before, with the files in the src directories bucketed by dst and carrier. The code below computes statistics on the table, which can then be used by the Catalyst optimizer. Next, the partitioned and bucketed table is read into a new DataFrame df2.

```
spark.sql("ANALYZE TABLE flightsbkdc COMPUTE STATISTICS")
val df2 = spark.table("flightsbkdc")
```

Next, let's look at the optimizations for the following query:

```
df2.filter("src = 'DEN' and depdelay > 1")
  .groupBy("src", "dst", "carrier")
  .avg("depdelay")
  .sort(desc("avg(depdelay)")).show()
```

result:

```
+---+---+-----+-----+
|src|dst|carrier|    avg(depdelay) |
+---+---+-----+-----+
|DEN|EWR|    UA| 60.95841209829867|
|DEN|LAX|    DL| 59.849624060150376|
|DEN|SFO|    UA| 59.058282208588956|
. . .
```

Here again, we see partition filter and filter pushdown, but we also see that there is no “Exchange” like there was before bucketing, which means there was no shuffle to aggregate by src, dst, and carrier.

```

== Physical Plan ==
TakeOrderedAndProject(limit=1001, orderBy=[avg(depdelay)#491 DESC
NULLS LAST], output=[src#460,dst#452,carrier#451,avg(depdelay)#504])

+- *(1) HashAggregate(keys=[src#460, dst#452, carrier#451],
functions=[avg(depdelay#455)], output=[src#460, dst#452,
carrier#451, avg(depdelay)#491])
  +- *(1) HashAggregate(keys=[src#460, dst#452, carrier#451],
functions=[partial_avg(depdelay#455)], output=[src#460, dst#452,
carrier#451, sum#512, count#513L])

    +- *(1) Project [carrier#451, dst#452, depdelay#455, src#460]

      +- *(1) Filter (isnotnull(depdelay#455) && (depdelay#455 > 1.0))

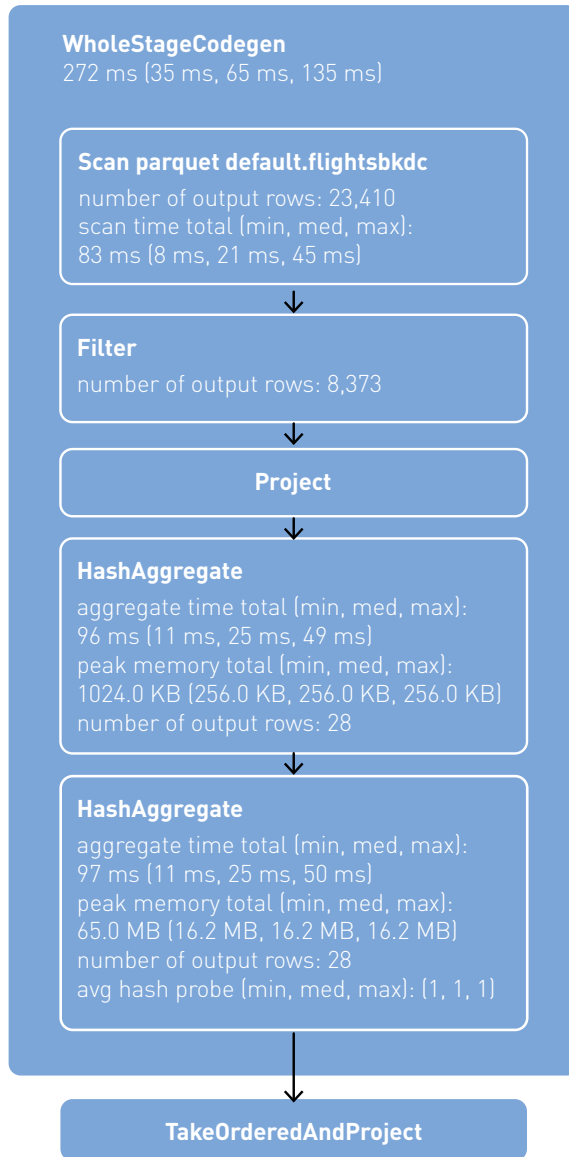
        +- *(1) FileScan parquet default.flightsbkdc
          [carrier#451,dst#452,depdelay#455,src#460]
          Batched: true, Format: Parquet, Location:
PrunedInMemoryFileIndex
          [maprfs:/user/mapr/data/flightsbkdc/src=DEN],
          PartitionCount: 1, PartitionFilters: [isnotnull(src#460),
(src#460 = DEN)],
          PushedFilters: [IsNull(depdelay),
GreaterThan(depdelay,1.0)],
          ReadSchema:
struct<carrier:string,dst:string,depdelay:double>

```

In the DAG below, we see that there is no exchange shuffle, and we see “Whole-Stage Java Code Generation,” which optimizes CPU usage by generating a single optimized function in bytecode.

Duration: 0.2 s

Succeeded Jobs: 11



Bucketing Tips

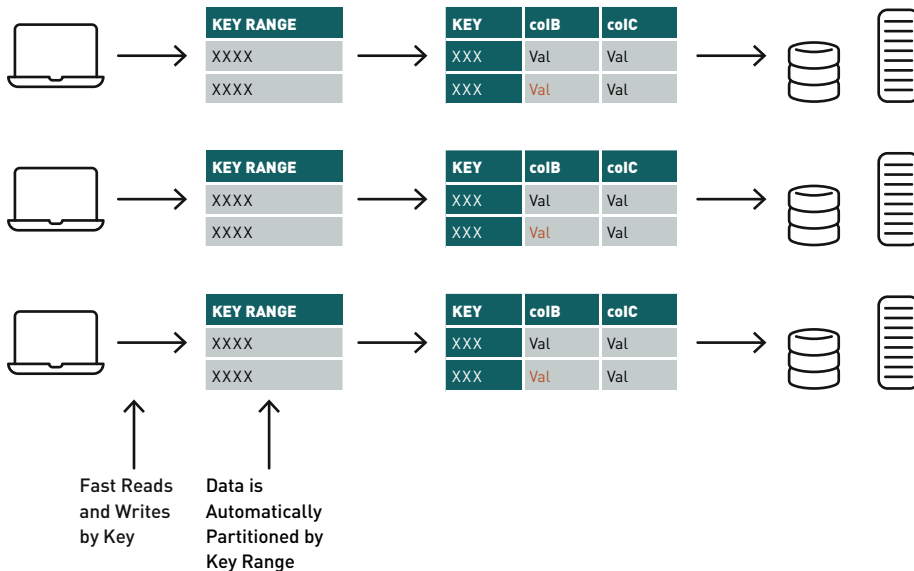
Partitioning should only be used with columns that have a limited number of values; bucketing works well when the number of unique values is large. Columns which are used often in queries and provide high selectivity are good choices for bucketing. Spark tables that are bucketed store metadata about how they are bucketed and sorted, which optimizes:

- Queries on bucketed values (Spark 2.4 supports bucket pruning)
- Aggregations on bucketed values (wide transformations)
- Joins on bucketed values

MapR Database, Data Modeling, Partitioning, and Filter Pushdown

Partitioning and Row Key Design

With [MapR Database](#), a table is automatically partitioned into tablets across a cluster by key range, providing for scalable and fast reads and writes by row key.



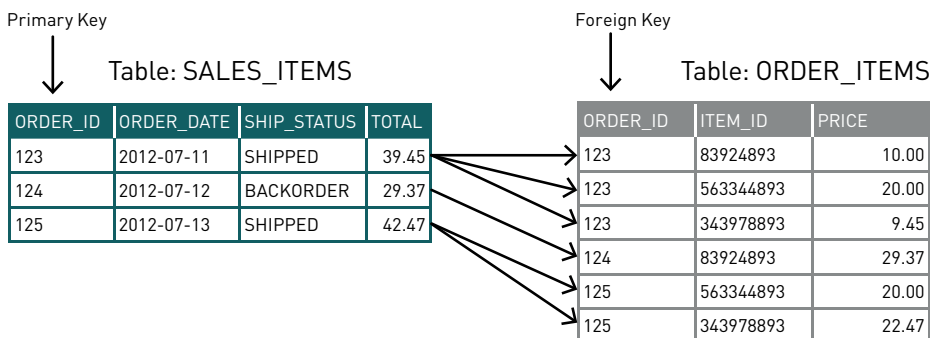
In this use case, the row key (the id) starts with the origin (destination airport codes), followed by the flightdate and carrier, so the table is automatically partitioned and sorted by the src, dst, date, and carrier.

```
{
  "id": "ATL_LGA_2017-01-01_AA_1678",
  "dofw": 7,
  "carrier": "AA",
  "src": "ATL",
  "dst": "LGA",
  "crsdehour": 17,
  "crsdeptime": 1700,
  "depdelay": 0.0,
  "crsarrrtime": 1912,
  "arrdelay": 0.0,
  "crselapsedtime": 132.0,
  "dist": 762.0
}
```

Table is automatically
partitioned and sorted
by id row key

MapR Database Data Modeling: Avoiding JOINS with Nested Entities

If your tables exist in a one-to-many relationship, it's possible to model it as a single document; this can avoid expensive JOINS. In the one-to-many relationship example below, we have an order table, which has a one-to-many relationship with an order items table.



Here is a nested entity example of this one-to-many relationship in a document database. In this example, the order and related line items are stored together and can be read together with a find on the row key (`_id`). This makes the reads a lot faster than joining tables together.

```
{
  "id": "123",
  "date": "10/10/2017",
  "ship_status": "backordered"
  "orderitems": [
    {
      "itemid": "4348",
      "price": 10.00
    },
    {
      "itemid": "5648",
      "price": 15.00
    }
  ]
}
```

See [Data Modeling Guidelines for NoSQL JSON Document Databases](#) and [Guidelines for HBase Schema Design](#) for more information on designing your MapR Database schema. (Nested Entities are also possible with JSON and Parquet files.)

Projection and Filter Pushdown into MapR Database

Below, we see the physical plan for a DataFrame query, with projection and filter pushdown highlighted in red. This means that the scanning of the `src`, `dst`, and `depdelay` columns and the filter on the `depdelay` column are pushed down into MapR Database, meaning that the scanning and filtering will take place in MapR Database before returning the data to Spark. Projection pushdown minimizes data transfer between MapR Database and the Spark engine by omitting unnecessary fields from table scans. It is especially beneficial when a table contains many columns. Filter pushdown improves performance by reducing the amount of data passed between MapR Database and the Spark engine when filtering data.

```
df.filter("src = 'ATL' and depdelay > 1")
.groupBy("src", "dst")
.avg("depdelay").sort(desc("avg(depdelay)")).explain

== Physical Plan ==
*(3) Sort [avg(depdelay)#273 DESC NULLS LAST], true, 0
+- Exchange rangepartitioning(avg(depdelay)#273 DESC NULLS LAST,
200)
    +- *(2) HashAggregate(keys=[src#5, dst#6],
        functions=[avg(depdelay#9)])
        +- Exchange hashpartitioning(src#5, dst#6, 200)
            +- *(1) HashAggregate(keys=[src#5, dst#6],
                functions=[partial_avg(depdelay#9)])
                +- *(1) Filter (((isnotnull(src#5) &&
                    isnotnull(depdelay#9)) &&
                    (src#5 = ATL)) && (depdelay#9 > 1.0))
                    +- *(1) Scan MapRDBRelation(/user/mapr/flighttable
[src#5,dst#6,depdelay#9] PushedFilters: [IsNotNull(src),
IsNotNull(depdelay), EqualTo(src,ATL), GreaterThan(depdelay,1.0)]
```

Spark Web UI and SQL Tips

Read or review [chapter 3](#) in order to understand how to use the Spark Web UI to explore your task jobs, storage, and SQL query plan. Here is a summary of tips and what to look for:

SQL Tab

You can see details about the query plan produced by Catalyst on the web UI SQL tab. In the query plan details, you can check and 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; 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.

Stages Tab

You can use the stage detail metrics to identify problems with an executor or task distribution. Things to look for:

- Tasks that are taking longer and/or killed tasks. If your task process time is not balanced, then resources could be wasted.
- Shuffle read size that is not balanced.
- If your partitions/tasks are not balanced, then consider repartition as described under partitioning.

Storage Tab

Caching Datasets can make execution faster if the data will be reused. You can use the storage tab to see if important Datasets are fitting into memory.

Executors Tab

You can use the executors tab to confirm that your application has the amount of resources needed.

- Shuffle Read Write Columns: shows size of data transferred between stages
- Storage Memory Column: shows the current used/available memory
- Task Time Column: shows task time/garbage collection time

References and More Information

[Project Tungsten: Bringing Apache Spark Closer to Bare Metal](#)

[Apache Spark as a Compiler](#)

[Apache Drill Architecture](#)

[MapR Spark Troubleshooting Hub](#)

[Apache Spark SQL Performance Tuning](#)

[Spark Summit Session Optimizing Apache Spark SQL Joins](#)

[Diving into Spark and Parquet Workloads, by Example](#)

[Spark Summit Hive Bucketing in Apache Spark](#)

[Lessons from the Field, Episode II: Applying Best Practices to Your Apache Spark Applications](#)

[Why You Should Care about Data Layout in the Filesystem](#)

[Spark + Parquet In Depth](#)

[Apache Spark: Config Cheatsheet](#)

[Spark Configuration](#)

[Projection and Filter Pushdown with Apache Spark DataFrames and Datasets](#)

[Apache Spark: Debugging and Logging Best Practices](#)

Appendix

We have covered a lot of ground in this book. By no means, however, does it cover everything to be experienced with Spark. Spark is constantly growing and adding functionality to make Spark programs easier to program, use less memory, execute faster, run more predictably, and work with new machine learning libraries or frameworks.

Code

You can download the code, data, and instructions to run the examples in the book from here: <https://github.com/mapr-demos/mapr-spark2-ebook>

Running the Code

All of the components of the examples discussed in this book can run on the same cluster with the MapR Data Platform.



- [MapR Sandbox](#) is a single node MapR cluster, available as a VMware or VirtualBox VM that lets you get started quickly with MapR and Spark.
- [MapR Container for Developers](#) is a Docker container that enables you to create a single node MapR cluster. The container is lightweight and designed to run on your laptop.
- [MapR Data Science Refinery](#) is an easy-to-deploy and scalable data science toolkit with native access to all platform assets and superior out-of-the-box security.
- Find out more at [Get Started with MapR](#).

Additional Resources

- [MapR Developer Portal](#)
- [MapR Spark Documentation](#)
- [Spark SQL, DataFrames, and Datasets Guide](#)
- [Structured Streaming Programming Guide](#)
- [Spark GraphX Guide](#)
- [Spark Machine Learning Library \(MLlib\) Guide](#)
- [Streaming Architecture](#) ebook
- [Machine Learning Logistics](#) ebook
- [Event-Driven Microservices Patterns](#)
- [Free On-Demand Training: Apache Spark](#)
- [MapR and Spark](#)
- [Spark: The Definitive Guide](#) - O'Reilly Media
- Spark documentation including deployment and configuration:
<https://spark.apache.org/docs/latest/>

About the Authors

Carol McDonald

Ian Downard

About the Reviewers

Rachel Silver

Jim Scott

training

MAPR®

MAPR INSTRUCTOR-LED & ON-DEMAND TRAINING LEADS TO GREAT THINGS

DATA ANALYSTS

DEVELOPERS

ADMINISTRATORS



Start today at mapr.com/training