SEAS 8515 - Lecture 3

Apache Spark's Structured APIs

School of Engineering
& Applied Science
THE GEORGE WASHINGTON UNIVERSITY

Dr. Adewale Akinfaderin

Spark’s Application Concepts

These concepts help in understanding the transformation and execution of code across Spark executors in a Spark application.

- **Application**: A user program built on Spark, utilizing its APIs. It includes both a driver program and executors on a cluster.
- **SparkSession**: An object that serves as the entry point to Spark's underlying functionality, enabling programming with Spark APIs. It's automatically instantiated in an interactive Spark shell, whereas in a Spark application, you create the SparkSession object yourself.
- **Job**: A parallel computation comprising multiple tasks, triggered by a Spark action like save() or collect().
- **Stage**: Jobs are divided into smaller sets of tasks called stages, which have dependencies on each other.
- **Task**: The smallest unit of execution or work sent to a Spark executor.

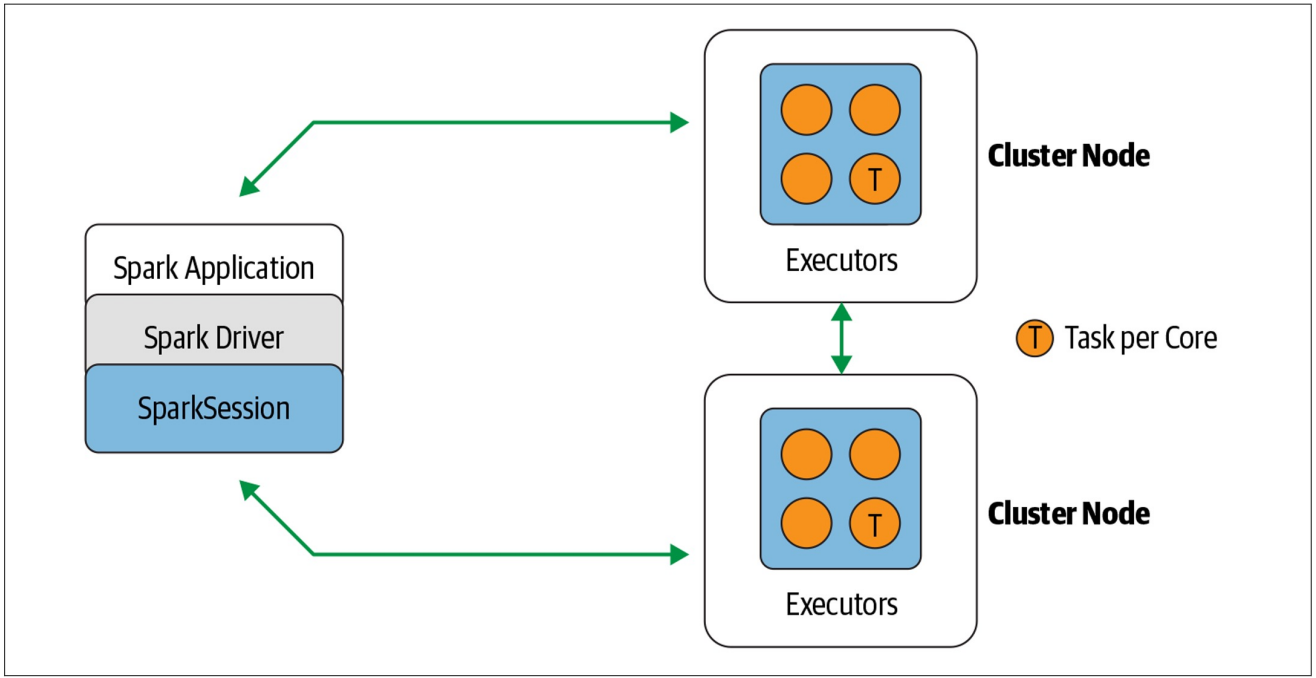
Spark Application and SparkSession

Spark Application and Driver: Every Spark application revolves around a driver program that creates a SparkSession. While in a Spark shell the driver and SparkSession are pre-setup, in other scenarios, they enable running operations either locally or on a distributed cluster.

SparkSession in Distributed Architecture: A SparkSession allows programming with Spark APIs. It plays a key role in facilitating communication between Spark components in a distributed system, enabling operations across a cluster.

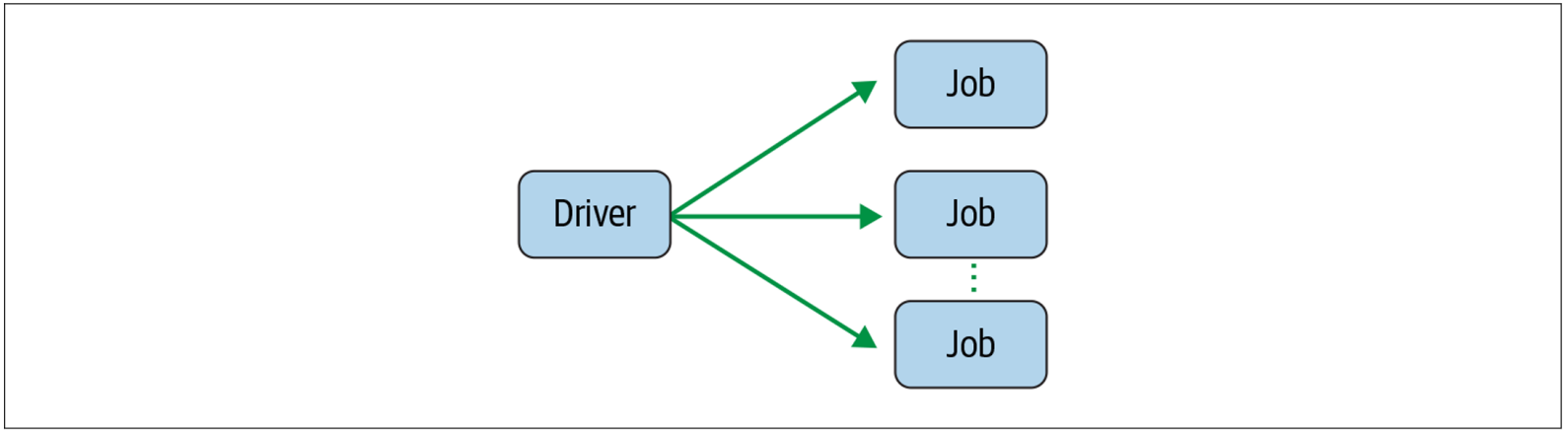
Spark Application and SparkSession

Spark components communicate through the Spark driver in Spark's distributed architecture



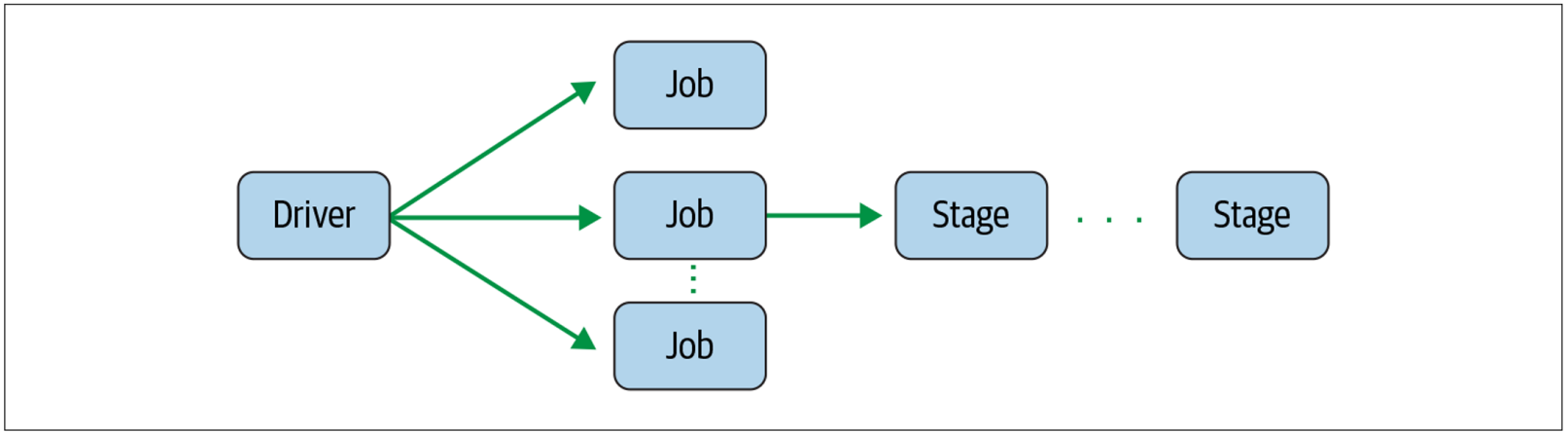
Spark Jobs

- **Spark Application to Jobs:** In interactive Spark shell sessions, the driver turns Spark applications into multiple Spark jobs.
- **Job to DAG Transformation:** Each Spark job is transformed into a Directed Acyclic Graph (DAG), serving as the execution plan with nodes representing one or more Spark stages.



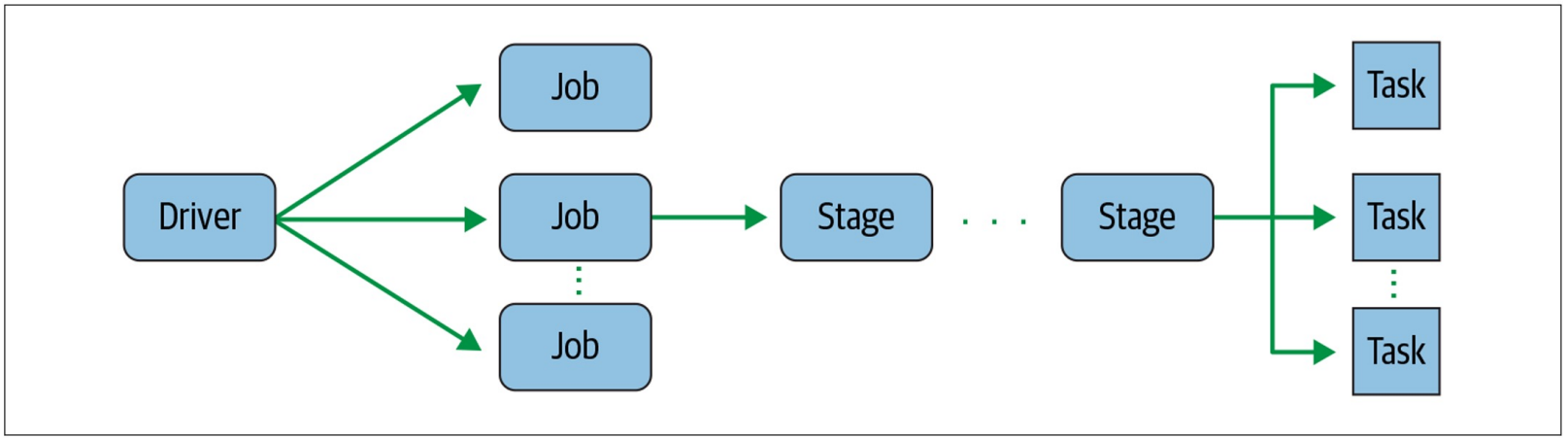
Spark Stages

- **DAG Nodes and Stages:** Within the DAG nodes, stages are determined based on the possibility of serial or parallel execution of operations. Not all operations can occur in a single stage, leading to division into multiple stages.
- **Stage Delineation and Data Transfer:** Stages are often separated based on computation boundaries of operators, dictating the data transfer between Spark executors.



Spark Tasks

- **Tasks in Stages:** Each stage in Spark includes tasks, assigned to separate cores on Spark executors, with each task processing a single data partition.
- **Parallel Execution:** Executors with multiple cores can perform numerous tasks simultaneously, enabling high-level parallelism in processing data partitions.



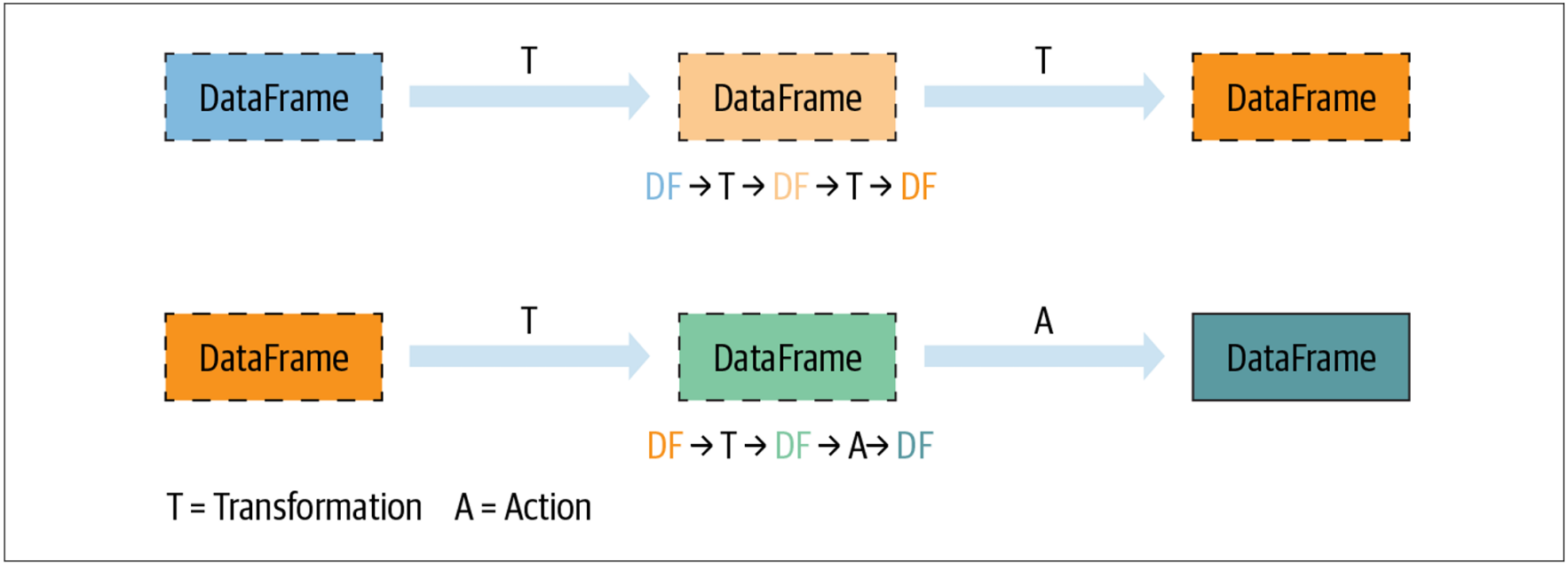
Transformations, Actions, and Lazy Evaluation

- **Transformations in Spark:** Transformations in Spark, like *select()* or *filter()*, create a new DataFrame from an existing one without modifying the original, showcasing data immutability. These operations result in a new DataFrame with the transformed data.
- **Lazy Evaluation of Transformations:** Spark employs lazy evaluation for transformations, meaning they are not immediately computed but recorded as a lineage. This approach allows Spark to optimize execution by rearranging or coalescing transformations into stages, with actual computation occurring only when an action is invoked or data is accessed.

Transformations, Actions, and Lazy Evaluation

- **Action Triggers Evaluation:** In Spark, an action initiates the lazy evaluation of recorded transformations, each leading to a new DataFrame.
- **Lazy Transformations and Eager Actions:** Transformations are recorded and evaluated lazily, while actions prompt immediate execution, striking a balance between efficiency and immediacy.
- **Query Optimization through Lineage:** Spark optimizes queries by examining chained transformations, using the recorded lineage of these transformations for efficient processing.
- **Fault Tolerance via Lineage and Immutability:** Spark ensures fault tolerance and resilience by utilizing the immutable nature of DataFrames and the recorded lineage of transformations, allowing it to recreate states post-failure.

Lazy Transformations and Eager Actions



Transformations and actions as Spark operations

Transformations	Actions
orderBy()	show()
groupBy()	take()
filter()	count()
select()	collect()
join()	save()

- **Query Plan Composition:** Spark's query plan is composed of both actions and transformations, detailed in the upcoming chapter. No part of this plan executes until an action is called.
- **Triggering Execution with Action:** In the given example, the execution of transformations like read() and filter() only occurs when the action count() is invoked, demonstrating Spark's lazy evaluation mechanism

Transformations and actions as Spark operations

```
# In Python
>>> strings = spark.read.text("../README.md")
>>> filtered = strings.filter(strings.value.contains("Spark"))
>>> filtered.count()
20

// In Scala
scala> import org.apache.spark.sql.functions._
scala> val strings = spark.read.text("../README.md")
scala> val filtered = strings.filter(col("value").contains("Spark"))
scala> filtered.count()
res5: Long = 20
```

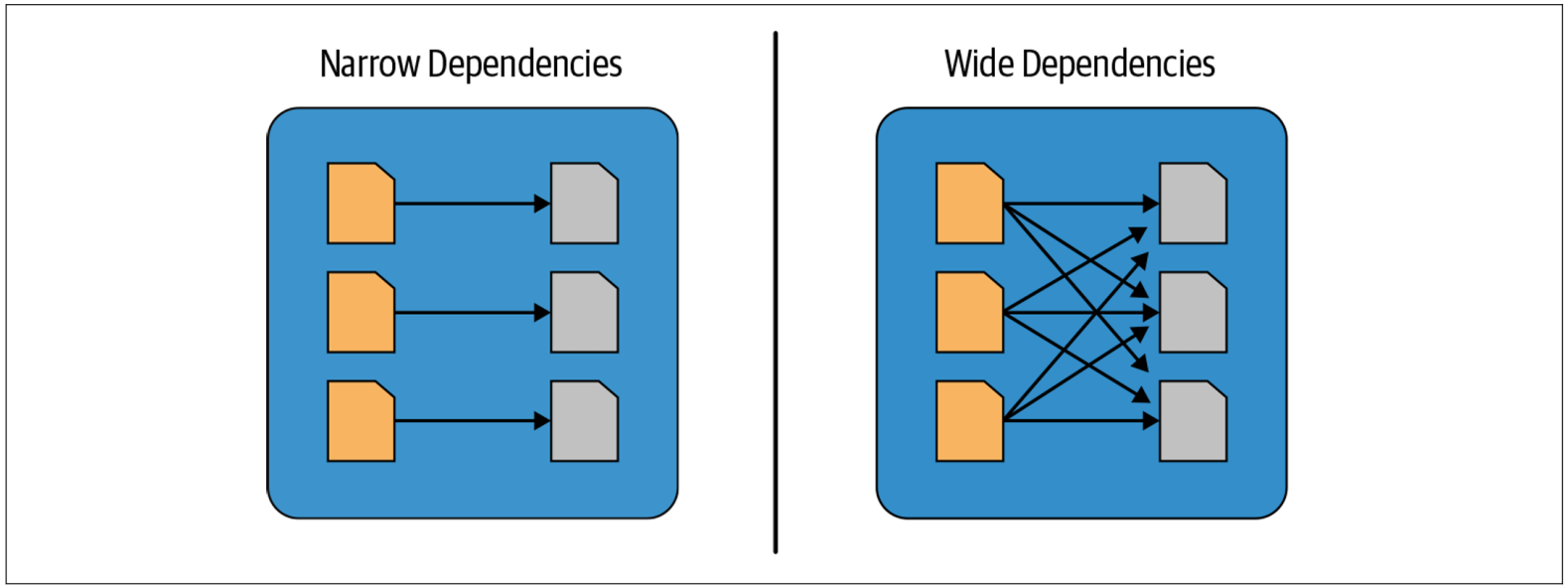
Narrow and Wide Transformations

- **Lazy Evaluation of Transformations:** Transformations in Spark are evaluated lazily, allowing Spark to analyze the entire computational query for potential optimization.
- **Optimization Strategies:** Spark optimizes by either joining or pipelining operations into a single stage, or by breaking them into multiple stages, particularly when operations necessitate shuffling or exchanging data across clusters.
- **Narrow vs. Wide Dependencies:** Transformations are categorized based on dependencies: narrow dependencies allow for output partition computation from a single input partition.

Narrow and Wide Transformations

- **Narrow Transformations Example:** Operations like filter() and contains() in the provided code snippet are narrow transformations, as they can process data within a single partition without requiring data exchange between partitions.
- **Wide Transformations with groupBy() or orderBy():** These operations command Spark to execute wide transformations, involving the reading, combining, and disk writing of data from multiple partitions.
- **Data Shuffle in Wide Transformations:** In wide transformations, like a count using groupBy(), Spark necessitates a shuffle of data across all executor partitions in the cluster, with orderBy() requiring outputs from various partitions to complete the final aggregation.

Narrow and Wide Transformations



Spark: What's Underneath an RDD?

- **Basic Abstraction:** The Resilient Distributed Dataset (RDD) is the fundamental abstraction in Spark.
- **Three Key Characteristics of RDDs:**
  - Dependencies: Indicate how an RDD is constructed from inputs, crucial for recreating RDDs.
  - Partitions: Facilitate splitting work across executors, with locality information used for optimizing data transmission.
  - Compute Function: A function that generates an Iterator[T], representing data in the RDD.
- **Dependencies and Resiliency:** Dependencies allow Spark to recreate an RDD if needed, providing resilience and fault tolerance.
- **Partitions and Parallel Computation:** Partitions enable Spark to parallelize computation across executors, enhancing efficiency.

Spark: What's Underneath an RDD?

- **Compute Function Opacity:** Spark sees the compute function as a lambda expression without understanding its specific operation (e.g., join, filter), making it opaque.
- **Iterator[T] Data Type Opacity:** In Python RDDs, the Iterator[T] is a generic object, opaque to Spark, limiting its understanding of the data's nature.
- **Limitation in Computation Optimization:** Due to Spark's inability to inspect the computation or expression within the function, it can't optimize these expressions effectively.
- **Data Type and Serialization:** The data type in T is opaque to Spark, leading to generic serialization without data compression, impacting the efficiency of query plan arrangement.

Key Merits and Benefits

- **Structural Benefits:** Adopting a structured approach in Spark offers improved performance and space efficiency across various components. Detailed exploration of these benefits will be covered with DataFrame and Dataset APIs.
- **Advantages Beyond Performance:** Besides performance enhancements, structure in Spark brings expressivity, simplicity, composability, and uniformity. These advantages will be demonstrated through examples, particularly focusing on expressivity and composability in common data analysis patterns.

Key Merits and Benefits

```
# In Python
# Create an RDD of tuples (name, age)
dataRDD = sc.parallelize([("Brooke", 20), ("Denny", 31), ("Jules", 30),
  ("TD", 35), ("Brooke", 25)])
# Use map and reduceByKey transformations with their lambda
# expressions to aggregate and then compute average

agesRDD = (dataRDD
  .map(lambda x: (x[0], (x[1], 1)))
  .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
  .map(lambda x: (x[0], x[1][0]/x[1][1])))
```

Key Merits and Benefits

By contrast, what if we were to express the same query with high-level DSL operators and the DataFrame API, thereby instructing Spark what to do? Have a look:

```
# In Python
from pyspark.sql import SparkSession
from pyspark.sql.functions import avg
# Create a DataFrame using SparkSession
spark = (SparkSession
  .builder
  .appName("AuthorsAges")
  .getOrCreate())
# Create a DataFrame
data_df = spark.createDataFrame([("Brooke", 20), ("Denny", 31), ("Jules", 30),
  ("TD", 35), ("Brooke", 25)], ["name", "age"])
# Group the same names together, aggregate their ages, and compute an average
avg_df = data_df.groupBy("name").agg(avg("age"))
# Show the results of the final execution
avg_df.show()
```

```
+------+--------+
|  name|avg(age)|
+------+--------+
|Brooke|    22.5|
| Jules|    30.0|
|    TD|    35.0|
| Denny|    31.0|
+------+--------+
```

Key Merits and Benefits - Takeaways

- **High-Level DSLs for Improved Expression:** Employing high-level DSLs and APIs in Spark makes the code more expressive and simpler, allowing for efficient query composition and optimization.
- **Flexibility and Choice in APIs**: Spark offers flexibility in API usage, allowing a switch between high-level structured patterns and the unstructured low-level RDD API, catering to diverse development needs.
- **Consistency Across Spark Modules:** The Spark SQL engine ensures uniform APIs across different Spark components, facilitating consistent data handling and analysis in various modules like Structured Streaming and MLlib.

The DataFrame API

- **Inspired by pandas:** Spark DataFrames, inspired by pandas, function as distributed in-memory tables with named columns and predefined schemas, each column having a specific data type like integer, string, array, etc.
- **Table-like Visualization:** To the user, Spark DataFrames appear as tables, making them easy to understand and interact with.
- **Ease of Operation:** The structured table format of DataFrames simplifies common data operations on rows and columns.
- **Immutability and Lineage:** Spark DataFrames are immutable, with a lineage of transformations maintained by Spark, allowing new DataFrames to be created while preserving older versions.
- **Schema Definition:** In a DataFrame, each column and its associated data type can be explicitly defined in the schema, adding clarity and structure to the data.

The DataFrame API - Format

Id (Int)	First (String)	Last (String)	Url (String)	Published (Date)	Hits (Int)	Campaigns (List[Strings])
1	Jules	Damji	https://tinyurl.1	1/4/2016	4535	[twitter, LinkedIn]
2	Brooke	Wenig	https://tinyurl.2	5/5/2018	8908	[twitter, LinkedIn]
3	Denny	Lee	https://tinyurl.3	6/7/2019	7659	[web, twitter, FB, LinkedIn]
4	Tathagata	Das	https://tinyurl.4	5/12/2018	10568	[twitter, FB]

Spark’s Basic Data Types

- **Support for Basic Data Types:** Spark accommodates basic internal data types, compatible with its supported programming languages, allowing users to specify these types in applications or schemas.
- **Data Type Declaration in Scala:** In Scala, for example, users can define or declare column names in Spark with specific data types such as String, Byte, Long, or Map, assigning appropriate variable names to these types.

```
$SPARK_HOME/bin/spark-shell
scala> import org.apache.spark.sql.types._
import org.apache.spark.sql.types._
scala> val nameTypes = StringType
nameTypes: org.apache.spark.sql.types.StringType.type = StringType
scala> val firstName = nameTypes
firstName: org.apache.spark.sql.types.StringType.type = StringType
scala> val lastName = nameTypes
lastName: org.apache.spark.sql.types.StringType.type = StringType
```

Spark's Basic Data Types - Python

Data type	Value assigned in Python	API to instantiate
ByteType	int	DataTypes.ByteType
ShortType	int	DataTypes.ShortType
IntegerType	int	DataTypes.IntegerType
LongType	int	DataTypes.LongType
FloatType	float	DataTypes.FloatType
DoubleType	float	DataTypes.DoubleType
StringType	str	DataTypes.StringType
BooleanType	bool	DataTypes.BooleanType
DecimalType	decimal.Decimal	DecimalType

Spark’s Structured and Complex Data Types

- Handling Complex Data: In advanced data analytics, the data often goes beyond simple types, being complex, structured, or nested, necessitating Spark's capability to manage these complex data types.
- Variety of Complex Types: Spark supports a wide range of complex data types, including maps, arrays, structs, dates, timestamps, and fields, to accommodate diverse data structures in analysis.

Data type	Value assigned in Python	API to instantiate
`BinaryType`	`bytearray`	`BinaryType()`
`TimestampType`	`datetime.datetime`	`TimestampType()`
`DateType`	`datetime.date`	`DateType()`
`ArrayType`	List, tuple, or array	`ArrayType(dataType, [nullable])`
`MapType`	`dict`	`MapType(keyType, valueType, [nullable])`
`StructType`	List or tuple	`StructType([fields])`
`StructField`	A value type corresponding to the type of this field	`StructField(name, dataType, [nullable])`

Spark’s Structured and Complex Data Types

- Handling Complex Data: In advanced data analytics, the data often goes beyond simple types, being complex, structured, or nested, necessitating Spark's capability to manage these complex data types.
- Variety of Complex Types: Spark supports a wide range of complex data types, including maps, arrays, structs, dates, timestamps, and fields, to accommodate diverse data structures in analysis.

Data type	Value assigned in Python	API to instantiate
`BinaryType`	`bytearray`	`BinaryType()`
`TimestampType`	`datetime.datetime`	`TimestampType()`
`DateType`	`datetime.date`	`DateType()`
`ArrayType`	List, tuple, or array	`ArrayType(dataType, [nullable])`
`MapType`	`dict`	`MapType(keyType, valueType, [nullable])`
`StructType`	List or tuple	`StructType([fields])`
`StructField`	A value type corresponding to the type of this field	`StructField(name, dataType, [nullable])`

Schemas and Creating DataFrames

- Schema Definition in Spark: A schema in Spark specifies the column names and their corresponding data types for a DataFrame, often used when reading structured data from external sources.
- Advantages of Predefined Schema:
  - Eliminates the need for Spark to infer data types, reducing processing overhead.
  - Avoids the necessity for Spark to execute a separate job to determine the schema from large data files, saving time and resources.
  - Facilitates early error detection when data doesn’t conform to the predefined schema.
- Best Practice Recommendation: It's recommended to always define a schema upfront, particularly when dealing with large files from external data sources, to enhance efficiency and accuracy.

Two ways to define a schema

Spark allows you to define a schema in two ways. One is to define it programmatically, and the other is to employ a Data Definition Language (DDL) string, which is much simpler and easier to read.

To define a schema programmatically for a DataFrame with three named columns, author, title, and pages, you can use the Spark DataFrame API. For example:

```scala
// In Scala
import org.apache.spark.sql.types._
val schema = StructType(Array(StructField("author", StringType, false),
  StructField("title", StringType, false),
  StructField("pages", IntegerType, false)))
```

```python
# In Python
from pyspark.sql.types import *
schema = StructType([StructField("author", StringType(), False),
  StructField("title", StringType(), False),
  StructField("pages", IntegerType(), False)])
```

Two ways to define a schema

Defining the same schema using DDL is much simpler. You can choose whichever way you like to define a schema (you can use both!)

```
// In Scala
val schema = "author STRING, title STRING, pages INT"

# In Python
schema = "author STRING, title STRING, pages INT"
```

Columns and Expressions

- **Named Columns in DataFrames:** Similar to pandas, R DataFrames, or RDBMS tables, named columns in Spark DataFrames describe a type of field, allowing listing and operation of columns by their names using relational or computational expressions.
- **Column Objects and Methods:** In Spark's supported languages, columns are treated as objects with public methods, represented by the Column type, enabling various operations on them.
- **Expressions on Columns:** Logical or mathematical expressions can be applied to DataFrame columns, using methods like expr(), which is available in Spark’s Python and Scala libraries. expr() interprets arguments as expressions, allowing Spark to compute their results.

Rows

Spark Row Object: In Spark, a row is a Row object with ordered columns of varying data types, accessible by zero-based indexing in any of Spark's supported languages.

```
// In Scala
import org.apache.spark.sql.Row
// Create a Row
val blogRow = Row(6, "Reynold", "Xin", "https://tinyurl.6", 255568, "3/2/2015",
  Array("twitter", "LinkedIn"))
// Access using index for individual items
blogRow(1)
res62: Any = Reynold
```

```
# In Python
from pyspark.sql import Row
blog_row = Row(6, "Reynold", "Xin", "https://tinyurl.6", 255568, "3/2/2015",
  ["twitter", "LinkedIn"])
# access using index for individual items
blog_row[1]
'Reynold'
```

Rows

Row objects can be used to create DataFrames if you need them for quick interactivity and exploration:

```
# In Python
rows = [Row("Matei Zaharia", "CA"), Row("Reynold Xin", "CA")]
authors_df = spark.createDataFrame(rows, ["Authors", "State"])
authors_df.show()
```

```
// In Scala
val rows = Seq(("Matei Zaharia", "CA"), ("Reynold Xin", "CA"))
val authorsDF = rows.toDF("Author", "State")
authorsDF.show()
```

```
+--------------+-----+
|        Author|State|
+--------------+-----+
|Matei Zaharia|   CA|
|  Reynold Xin|   CA|
+--------------+-----+
```

Common DataFrame Operations

- **Loading Data into DataFrames:** To perform operations on DataFrames, data must first be loaded into Spark from various structured data sources using the DataFrameReader interface.
- **Support for Multiple Formats:** DataFrameReader in Spark supports reading data in numerous formats, including JSON, CSV, Parquet, Text, Avro, and ORC.
- **Writing Data from DataFrames:** For saving or exporting DataFrames back to a data source in a specific format, Spark utilizes the DataFrameWriter.

Using DataFrameReader and DataFrameWriter

- Ease of Data I/O: Spark's high-level abstractions, combined with community contributions, simplify reading and writing data, supporting a variety of sources like NoSQL stores, RDBMSs, and streaming engines.
- Example of Reading Data: To illustrate, reading a large CSV file, such as one with San Francisco Fire Department call data, is streamlined using Spark's DataFrameReader class and its methods, enhanced by predefined schema.
- Efficiency in Handling Large Data: For substantial datasets, like a CSV file with over 4.38 million records and 28 columns, defining a schema upfront is more efficient than having Spark infer the schema.

```python
# In Python, define a schema
from pyspark.sql.types import *

# Programmatic way to define a schema
fire_schema = StructType([StructField('CallNumber', IntegerType(), True),
                StructField('UnitID', StringType(), True),
                StructField('IncidentNumber', IntegerType(), True),
                StructField('CallType', StringType(), True),
                StructField('CallDate', StringType(), True),
                StructField('WatchDate', StringType(), True),
                StructField('CallFinalDisposition', StringType(), True),
                StructField('AvailableDtTm', StringType(), True),
                StructField('Address', StringType(), True),
                StructField('City', StringType(), True),
                StructField('Zipcode', IntegerType(), True),
                StructField('Battalion', StringType(), True),
                StructField('StationArea', StringType(), True),
                StructField('Box', StringType(), True),
                StructField('OriginalPriority', StringType(), True),
                StructField('Priority', StringType(), True),
                StructField('FinalPriority', IntegerType(), True),
                StructField('ALSUnit', BooleanType(), True),
                StructField('CallTypeGroup', StringType(), True),
                StructField('NumAlarms', IntegerType(), True),
                StructField('UnitType', StringType(), True),
                StructField('UnitSequenceInCallDispatch', IntegerType(), True),
                StructField('FirePreventionDistrict', StringType(), True),
                StructField('SupervisorDistrict', StringType(), True),
                StructField('Neighborhood', StringType(), True),
                StructField('Location', StringType(), True),
                StructField('RowID', StringType(), True),
                StructField('Delay', FloatType(), True)])

# Use the DataFrameReader interface to read a CSV file
sf_fire_file = "/databricks-datasets/learning-spark-v2/sf-fire/sf-fire-calls.csv"
fire_df = spark.read.csv(sf_fire_file, header=True, schema=fire_schema)
```

Using DataFrameReader and DataFrameWriter

- **Reading CSV Files:** The spark.read.csv() function is used to read CSV files, returning a DataFrame with rows and columns defined by the specified schema.
- **Writing Data with DataFrameWriter:** To save a DataFrame to an external data source in various formats, Spark employs the DataFrameWriter interface, supporting multiple data sources.
- **Parquet Format and Schema Preservation:** Parquet, a default columnar format in Spark with snappy compression, preserves the DataFrame's schema within its metadata, eliminating the need for manual schema provision in subsequent reads.

Using DataFrameReader and DataFrameWriter

- **Reading CSV Files:** The spark.read.csv() function is used to read CSV files, returning a DataFrame with rows and columns defined by the specified schema.
- **Writing Data with DataFrameWriter:** To save a DataFrame to an external data source in various formats, Spark employs the DataFrameWriter interface, supporting multiple data sources.
- **Parquet Format and Schema Preservation:** Parquet, a default columnar format in Spark with snappy compression, preserves the DataFrame's schema within its metadata, eliminating the need for manual schema provision in subsequent reads.

Saving a DataFrame as a Parquet file or SQL table

- **Data Exploration and Transformation:** A typical operation in data handling involves exploring and transforming data, followed by persisting the processed DataFrame in a format like Parquet or as a SQL table.
- **Saving as Parquet File:** Persisting a DataFrame is straightforward, similar to reading it. For example, in both Scala and Python, a DataFrame can be saved as a Parquet file using the write.format("parquet").save(path) method.
- **Saving as a SQL Table:** Alternatively, DataFrames can be saved as SQL tables, which involves registering the metadata with the Hive metastore, a process that will be detailed in subsequent chapters.

Saving a DataFrame as a Parquet file or SQL table

```scala
// In Scala to save as a Parquet file
val parquetPath = ...
fireDF.write.format("parquet").save(parquetPath)
```

```python
# In Python to save as a Parquet file
parquet_path = ...
fire_df.write.format("parquet").save(parquet_path)
```

```scala
// In Scala to save as a table
val parquetTable = ... // name of the table
fireDF.write.format("parquet").saveAsTable(parquetTable)
```

```python
# In Python
parquet_table = ... # name of the table
fire_df.write.format("parquet").saveAsTable(parquet_table)
```

Projections and filters.

- **Projections and Filters in Relational Data:** In relational data processing, a projection is used to return rows that match a specific condition, achieved through filters.
- **Spark's Methods for Projections and Filters**: In Spark, projections are performed using the select() method, and filters are applied using either the filter() or where() method.
- **Applying Techniques to Data Sets:** These methods can be utilized to examine specific aspects of datasets, such as analyzing particular data points in the SF Fire Department dataset.

Projections and filters.

```python
# In Python
few_fire_df = (fire_df
  .select("IncidentNumber", "AvailableDtTm", "CallType")
  .where(col("CallType") != "Medical Incident"))
few_fire_df.show(5, truncate=False)
```

```scala
// In Scala
val fewFireDF = fireDF
  .select("IncidentNumber", "AvailableDtTm", "CallType")
  .where($"CallType" =!= "Medical Incident")
fewFireDF.show(5, false)
```

```
+--------------+----------------------+--------------+
|IncidentNumber|AvailableDtTm         |CallType      |
+--------------+----------------------+--------------+
|2003235       |01/11/2002 01:47:00 AM|Structure Fire|
|2003235       |01/11/2002 01:51:54 AM|Structure Fire|
|2003235       |01/11/2002 01:47:00 AM|Structure Fire|
|2003235       |01/11/2002 01:47:00 AM|Structure Fire|
|2003235       |01/11/2002 01:51:17 AM|Structure Fire|
+--------------+----------------------+--------------+
only showing top 5 rows
```

Renaming, adding, and dropping columns

- **Renaming Columns for Various Reasons:** Columns in a DataFrame might be renamed for style, convention, readability, or brevity.
- **Handling Problematic Column Names:** In cases like the SF Fire Department dataset, original column names with spaces (e.g., "Incident Number") can be problematic, especially for formats like Parquet that prohibit spaces.
- **Methods for Renaming Columns:** Columns can be renamed by specifying new names in the schema with StructField, or selectively using the withColumnRenamed() method, like changing "Delay" to "ResponseDelayedInMins" to analyze specific data such as response times exceeding five minutes.

Renaming, adding, and dropping columns

```python
# In Python
new_fire_df = fire_df.withColumnRenamed("Delay", "ResponseDelayedinMins")
(new_fire_df
  .select("ResponseDelayedinMins")
  .where(col("ResponseDelayedinMins") > 5)
  .show(5, False))
```

```scala
// In Scala
val newFireDF = fireDF.withColumnRenamed("Delay", "ResponseDelayedinMins")
newFireDF
  .select("ResponseDelayedinMins")
  .where($"ResponseDelayedinMins" > 5)
  .show(5, false)
```

This gives us a new renamed column:

```
+---------------------+
|ResponseDelayedinMins|
+---------------------+
|5.233333             |
|6.9333334            |
|6.116667             |
|7.85                 |
|77.333336            |
+---------------------+
only showing top 5 rows
```

Renaming, adding, and dropping columns

- **Column Content and Type Modification:** Altering the contents or data types of a column is a common task in data exploration, often due to raw or dirty data, or incompatible types for relational operations.
- **Handling Incompatible Data Types:** For instance, in the SF Fire Department dataset, columns like CallDate, WatchDate, and AlarmDtTm are string types instead of Unix timestamps or SQL dates, which Spark handles more efficiently for transformations or time-based analyses.
- **Conversion to Usable Formats:** Converting these data types into more usable formats is straightforward using high-level API methods. The spark.sql.functions library offers functions like to_timestamp() and to_date(), making it easy to transform string dates into timestamp or date types in Spark.

Renaming, adding, and dropping columns

```
# In Python
fire_ts_df = (new_fire_df
  .withColumn("IncidentDate", to_timestamp(col("CallDate"), "MM/dd/yyyy"))
  .drop("CallDate")
  .withColumn("OnWatchDate", to_timestamp(col("WatchDate"), "MM/dd/yyyy"))
  .drop("WatchDate")
  .withColumn("AvailableDtTS", to_timestamp(col("AvailableDtTm"),
  "MM/dd/yyyy hh:mm:ss a"))
  .drop("AvailableDtTm"))

# Select the converted columns
(fire_ts_df
  .select("IncidentDate", "OnWatchDate", "AvailableDtTS")
  .show(5, False))
```

Renaming, adding, and dropping columns

- Data Type Conversion: The queries convert the data type of an existing column from a string to a Spark-supported timestamp format.
- Specifying New Format: They use a specified format string, such as "MM/dd/yyyy" or "MM/dd/yyyy hh:mm:ss a", for the conversion process.
- Column Replacement: After the conversion, the old column is removed using the drop() method, and the new one is appended using the withColumn() method.
- Creating Modified DataFrame: Finally, the resulting DataFrame with the modified columns is assigned to fire_ts_df.

```
+-------------------+-------------------+-------------------+
|IncidentDate       |OnWatchDate        |AvailableDtTS      |
+-------------------+-------------------+-------------------+
|2002-01-11 00:00:00|2002-01-10 00:00:00|2002-01-11 01:58:43|
|2002-01-11 00:00:00|2002-01-10 00:00:00|2002-01-11 02:10:17|
|2002-01-11 00:00:00|2002-01-10 00:00:00|2002-01-11 01:47:00|
|2002-01-11 00:00:00|2002-01-10 00:00:00|2002-01-11 01:51:54|
|2002-01-11 00:00:00|2002-01-10 00:00:00|2002-01-11 01:47:00|
+-------------------+-------------------+-------------------+
```

Aggregations

- **Aggregation Queries in Spark:** Common data analysis queries often involve understanding patterns, like the most frequent types of fire calls or the zip codes with the highest number of calls.
- **Using Transformations and Actions:** To answer these questions, Spark provides DataFrame transformations and actions like groupBy(), orderBy(), and count().
- **Column-Based Aggregation:** These methods enable aggregation based on specific column names and facilitate the computation of aggregate counts across them.

Aggregations

```
# In Python
(fire_ts_df
  .select("CallType")
  .where(col("CallType").isNotNull())
  .groupBy("CallType")
  .count()
  .orderBy("count", ascending=False)
  .show(n=10, truncate=False))
```

```
+---------------------------------+-------+
|CallType                         |count  |
+---------------------------------+-------+
|Medical Incident                 |2843475|
|Structure Fire                   |578998 |
|Alarms                           |483518 |
|Traffic Collision                |175507 |
|Citizen Assist / Service Call    |65360  |
|Other                            |56961  |
|Outside Fire                     |51603  |
|Vehicle Fire                     |20939  |
|Water Rescue                     |20037  |
|Gas Leak (Natural and LP Gases)  |17284  |
+---------------------------------+-------+
```

Other common DataFrame operations.

- **Common DataFrame Operations:** In addition to previously discussed methods, the DataFrame API in Spark also includes descriptive statistical methods like min(), max(), sum(), and avg().
- **Application in Data Analysis:** These methods can be applied to datasets, such as the SF Fire Department dataset, to compute various statistical measures.

Other common DataFrame operations.

```python
# In Python
import pyspark.sql.functions as F
(fire_ts_df
  .select(F.sum("NumAlarms"), F.avg("ResponseDelayedinMins"),
      F.min("ResponseDelayedinMins"), F.max("ResponseDelayedinMins"))
    .show())
```

```scala
// In Scala
import org.apache.spark.sql.{functions => F}
fireTsDF
  .select(F.sum("NumAlarms"), F.avg("ResponseDelayedinMins"),
  F.min("ResponseDelayedinMins"), F.max("ResponseDelayedinMins"))
  .show()
```

```
+--------------+-------------------------+-------------------------+---------+
|sum(NumAlarms)|avg(ResponseDelayedinMins)|min(ResponseDelayedinMins)|max(...) |
+--------------+-------------------------+-------------------------+---------+
|       4403441|         3.902170335891614|              0.016666668|1879.6167|
+--------------+-------------------------+-------------------------+---------+
```

End-to-End DataFrame Example

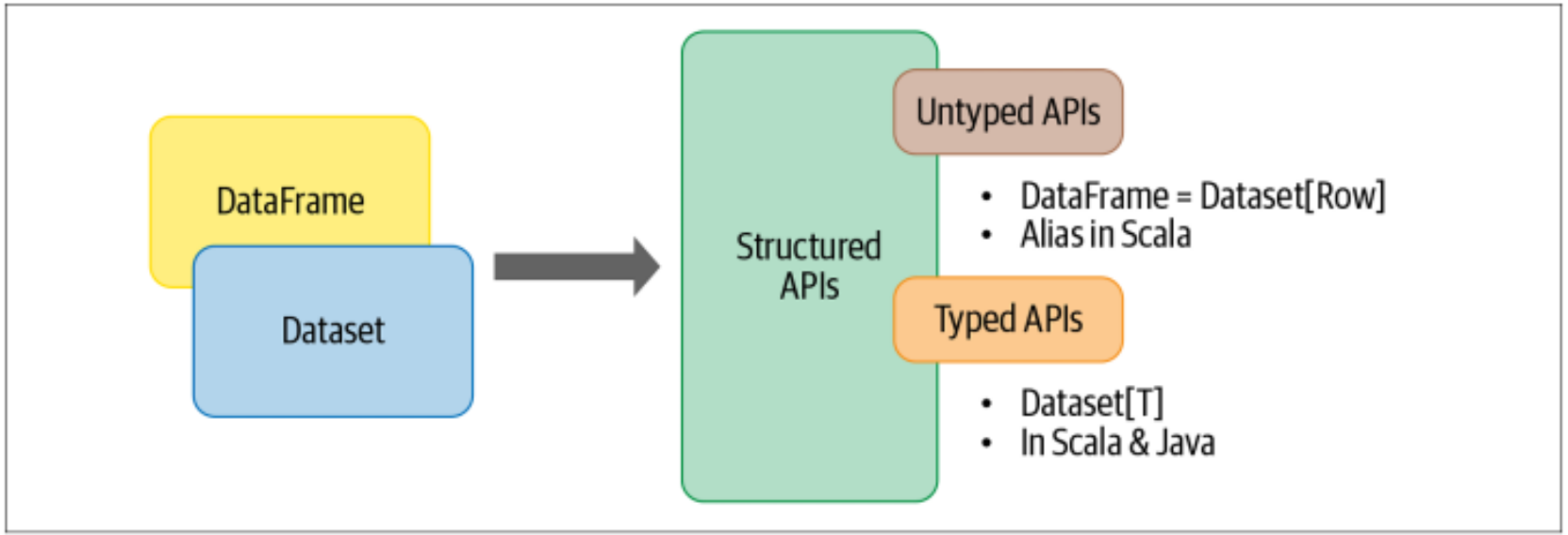
The San Francisco Fire Department public dataset opens up a vast range of possibilities for data analysis and ETL, beyond the examples provided here. For more in-depth exploration, Python and Scala notebooks are available on the book's GitHub repository, offering end-to-end DataFrame examples using this dataset with the DataFrame API and DSL relational operators.

- What were all the different types of fire calls in 2018?
- What months within the year 2018 saw the highest number of fire calls?
- Which neighborhood in San Francisco generated the most fire calls in 2018?
- Which neighborhoods had the worst response times to fire calls in 2018?
- Which week in the year in 2018 had the most fire calls?
- Is there a correlation between neighborhood, zip code, and number of fire calls?
- How can we use Parquet files or SQL tables to store this data and read it back?

The Dataset API

- **Unified DataFrame and Dataset APIs:** In Spark 2.0, the DataFrame and Dataset APIs were unified as Structured APIs, simplifying learning by offering a single set of interfaces.
- **Typed and Untyped APIs in Datasets:** Datasets feature two characteristics: typed and untyped APIs, with the untyped version essentially being DataFrames.
- **DataFrame Concept in Scala:** In Scala, a DataFrame is conceptualized as a collection of generic objects, Dataset[Row], where Row is a generic untyped JVM object capable of holding various field types.
- **Dataset as Strongly Typed Collections:** In contrast, a Dataset represents a collection of strongly typed JVM objects in Scala or classes in Java, providing both functional and relational operation capabilities, with an untyped view known as a DataFrame.

The Dataset API



Typed Objects, Untyped Objects, and Generic Rows

- **Language-Specific Dataset Usage in Spark:** Datasets are relevant only in Java and Scala, while DataFrames are used in Python and R. This distinction stems from the fact that Python and R are dynamically typed, whereas Scala and Java enforce compile-time type safety.
- **DataFrame as Untyped Dataset in Scala:** In Scala, a DataFrame is essentially an alias for an untyped Dataset[Row], with Row being a generic object type holding various data types.
- **Type Handling Across Languages:** While Scala and Java can have both typed and untyped datasets (Dataset[T] and DataFrame), Python and R operate with generic, untyped DataFrame structures.
- **Internal Row Manipulation:** Spark internally handles Row objects, converting them to equivalent types as needed across Scala, Java, and Python, with fields in a Row mapped to specific data types like IntegerType.

Typed Objects, Untyped Objects, and Generic Rows

Language	Typed and untyped main abstraction	Typed or untyped
Scala	`Dataset[T]` and DataFrame (alias for `Dataset[Row]`)	Both typed and untyped
Java	`Dataset<T>`	Typed
Python	DataFrame	Generic Row untyped
R	DataFrame	Generic Row untyped

Creating Datasets

- **Schema Knowledge for Dataset Creation:** Just like creating DataFrames, when constructing a Dataset, it's essential to know the schema, meaning the specific data types involved.
- **Inference Considerations with JSON and CSV:** While it's possible to infer the schema with JSON and CSV data, doing so with large datasets can be resource-intensive and costly.
- **Schema Specification in Scala:** In Scala, the most straightforward method to define the schema for a Dataset is by utilizing a case class.

Creating Datasets

- **Similarity in Operations for DataFrames and Datasets:** Just like DataFrames, Datasets in Spark can undergo various transformations and actions.
- **Use of Functions in Dataset Methods:** Functions can be used as arguments in Dataset methods, such as the filter() method which can take a lambda function as an argument.
- **Accessing Data Fields in Datasets:** When working with Datasets, individual data fields of JVM objects can be accessed using dot notation, akin to accessing fields in Scala classes or JavaBeans.
- **Filtering Differences between DataFrames and Datasets:** While DataFrames use SQL-like Domain-Specific Language (DSL) operations for filtering, Datasets utilize language-native expressions, such as Scala or Java code.

Creating Datasets

```
// In Scala
val filterTempDS = ds.filter({d => {d.temp > 30 && d.humidity > 70})

filterTempDS: org.apache.spark.sql.Dataset[DeviceIoTData] = [battery_level...]

filterTempDS.show(5, false)

+-------------|---------|----|----|-------------|---------|---+
|battery_level|c02_level|cca2|cca3|cn           |device_id|...|
+-------------|---------|----|----|-------------|---------|---+
|0            |1466     |US  |USA |United States|17       |...|
|9            |986      |FR  |FRA |France       |48       |...|
|8            |1436     |US  |USA |United States|54       |...|
|4            |1090     |US  |USA |United States|63       |...|
|4            |1072     |PH  |PHL |Philippines  |81       |...|
+-------------|---------|----|----|-------------|---------|---+
only showing top 5 rows
```

End-to-End Dataset Example

This end-to-end Dataset example mirrors the DataFrame example in conducting exploratory data analysis and ETL on a small, simulated IoT dataset, emphasizing query clarity and readability. Detailed operations are demonstrated in a notebook available in the GitHub repository, showcasing the use of the Dataset API.

- Detect failing devices with battery levels below a threshold.
- Identify offending countries with high levels of CO2 emissions.
- Compute the min and max values for temperature, battery level, CO2, and humidity.
- Sort and group by average temperature, CO2, humidity, and country.

SEAS 8515

Next Class:

Spark SQL and DataFrames: Introduction to Built-in Data Sources

School of Engineering & Applied Science
THE GEORGE WASHINGTON UNIVERSITY

Dr. Adewale Akinfaderin