Data 603 – Big Data Platforms

Lecture 5
Structured APIs (Part 1)

Lecture Outline

- Reading
- Docker Intro
- Going beyond RDD
- Apache Spark Structured API
- Apache Spark DataFrames
- Working with columns
- Working with rows
- Dataset API
- Spark SQL Intro
- Homework

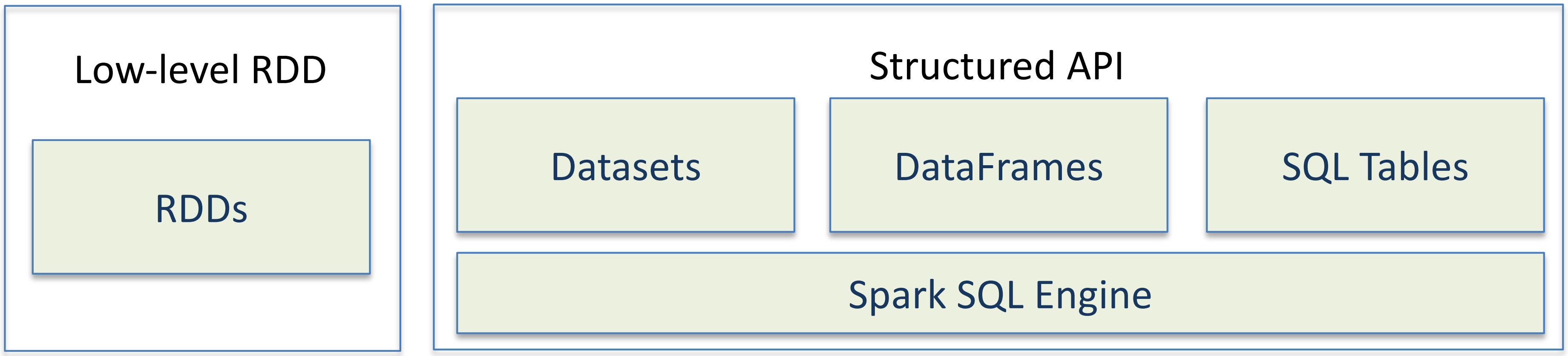
Reading

- Learning Spark 2nd ed.
  - RDD Review: Chapter 3 (1st half)
  - Today's Material: Chapter 3 (2nd half) and 4
  - Next week: Chapter 5

https://pages.databricks.com/rs/094-YMS-629/images/LearningSpark2.0.pdf

Going beyond RDD

Spark Core Abstractions



RDD Characteristics Review

- Dependencies
  - Instructions on how to construct an RDD with required inputs
  - RDDs can be recreated with dependencies and application of operations on them (**R**esiliency)
- Partitions (with locality information)
  - Collection of rows
  - Ability to split the work to parallelize computation on partitions across executors
  - One partition = parallelism of one
- Compute function: Partition => Iterator[T]
  - Associated with each RDD is a compute function that produces an Iterator[T] for the data that will be stored in RDD

Issues with RDDs

- The compute function (join, filter, select, aggregation) is opaque to Spark – Spark only sees it as a lambda expression
- Iterator[T] data type is opaque to Python RDDs – generic object in Python
- Spark is not able to optimize the expression
- Spark has no knowledge of the specific data type in T
- Telling Spark how to do things (vs. what you want) can make the code more complicated to write and read.

Issues with RDDs

Example from the textbook:

“Aggregating all the ages for each name, group by name, and then average the ages”

```
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])))
```

Apache Spark Structured API

Structuring Spark

- Add clarity and simplicity by using common patterns found in data analysis to express computations
  - High-level operations: filtering, selecting, counting, aggregating, averaging and grouping
- Common set of operators in DSL available as APIs
  - Tell Spark what you wish to compute with the data NOT how to do it
  - Provides an opportunity for Spark optimize the query plan for execution
- Spark's high-level APIs are uniform across its components and languages
  - Similar APIs for Python and Scala => Built on Spark SQL engine
- DSL operators that are modeled after relational operators that are similar to SQL

Structuring Spark - Benefits

- Better performance and space efficiency via DataFrame and Dataset APIs
  - Spark is able to parse queries and understand the intention, it can optimize and arrange operations for efficient execution.
  - Spark has better understanding of what needs to be done:
    - e.g. group by names, aggregate ages and computing the average.
- Expressivity, simplicity, composability and uniformity
  - Ability to compose entire computations using high-level operators as simple queries.
  - Possible due to the Spark SQL engine upon which the high-level Structured APIs are built

NOTE:

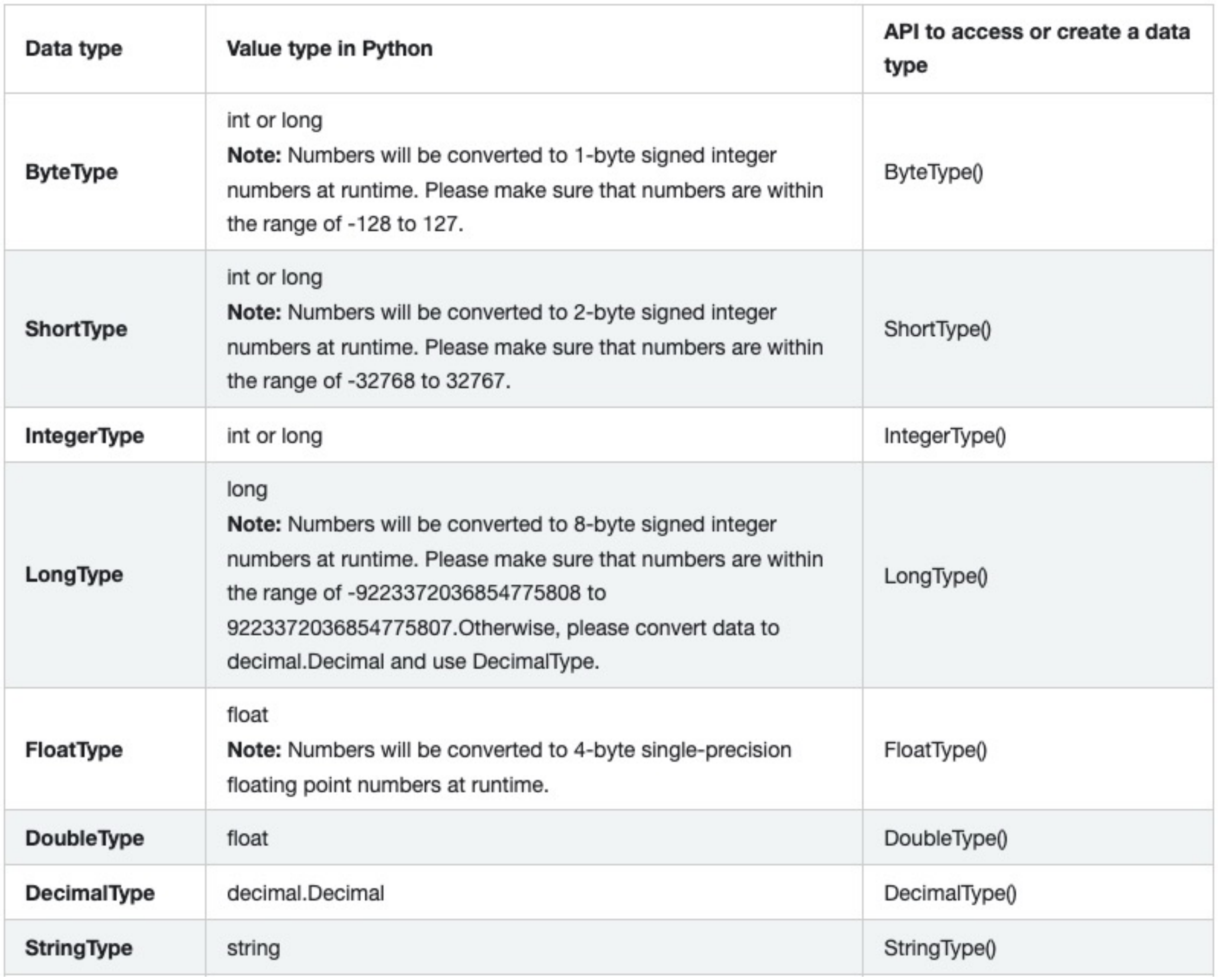
- You are always transforming and operating on DataFrames as structured data
- Possible to switch back to unstructured low-level RDD API

Apache Spark DataFrames

DataFrame API

- Inspired by Pandas DataFrames => structure, format, operations
- Analogous to distributed in-memory tables with named columns and schemas
  - Each column has a specific data type: integer, string, array, map, real, date, timestamp, etc.
  - Schema: A list that defines the columns and types within those columns
  - A named column in a DataFrame and its associated Spark data type can be declared in the schema
- DataFrames are immutable
  - When columns are added or changed, a new DataFrame is created
- Spark keeps lineage of all transformations

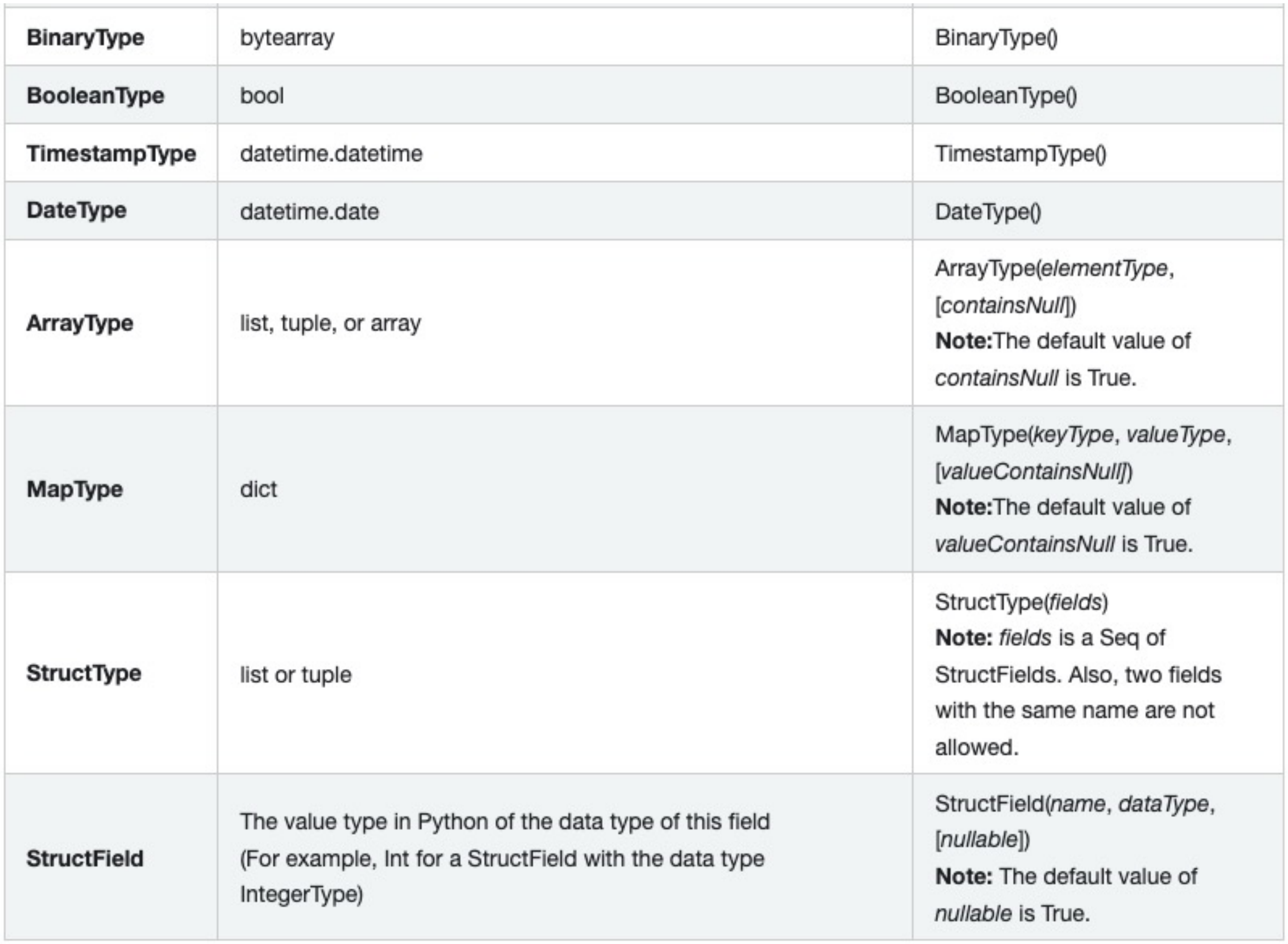
Spark Data Types 1/2



Data type	Value type in Python	API to access or create a data type
ByteType	int or long **Note:** Numbers will be converted to 1-byte signed integer numbers at runtime. Please make sure that numbers are within the range of -128 to 127.	ByteType()
ShortType	int or long **Note:** Numbers will be converted to 2-byte signed integer numbers at runtime. Please make sure that numbers are within the range of -32768 to 32767.	ShortType()
IntegerType	int or long	IntegerType()
LongType	long **Note:** Numbers will be converted to 8-byte signed integer numbers at runtime. Please make sure that numbers are within the range of -9223372036854775808 to 9223372036854775807.Otherwise, please convert data to decimal.Decimal and use DecimalType.	LongType()
FloatType	float **Note:** Numbers will be converted to 4-byte single-precision floating point numbers at runtime.	FloatType()
DoubleType	float	DoubleType()
DecimalType	decimal.Decimal	DecimalType()
StringType	string	StringType()

https://spark.apache.org/docs/latest/sql-ref-datatypes.html

Spark Data Types 2/2



BinaryType	bytearray	BinaryType()
BooleanType	bool	BooleanType()
TimestampType	datetime.datetime	TimestampType()
DateType	datetime.date	DateType()
ArrayType	list, tuple, or array	ArrayType(*elementType*, [*containsNull*]) **Note:**The default value of *containsNull* is True.
MapType	dict	MapType(*keyType*, *valueType*, [*valueContainsNull*]) **Note:**The default value of *valueContainsNull* is True.
StructType	list or tuple	StructType(*fields*) **Note:** *fields* is a Seq of StructFields. Also, two fields with the same name are not allowed.
StructField	The value type in Python of the data type of this field (For example, Int for a StructField with the data type IntegerType)	StructField(*name*, *dataType*, [*nullable*]) **Note:** The default value of *nullable* is True.

https://spark.apache.org/docs/latest/sql-ref-datatypes.html

Complex Types

- `ArrayType(elementType, containsNull)`
  - Values comprising a sequence of elements with the type of elementType.
- `MapType(keyType, valueType, valueContainsNull)`
  - Values comprising a set of key-value pairs.
  - The data type of keys is described by keyType and the data type of values is described by valueType.
  - For a MapType value, keys are not allowed to have null values.
- `StructType(fields)`
  - Represents values with the structure described by a sequence of StructFields (fields).
- `StructField(name, dataType, nullable)`
  - Represents a field in a StructType. The name of a field is indicated by name. The data type of a field is indicated by dataType.

https://spark.apache.org/docs/latest/sql-ref-datatypes.html

DataFrame Schemas

Q: What is a data schema?

DataFrame Schemas

Q: What makes structured data “structured”?

DataFrame Schemas

- Schemas come into play when reading structured data from an external source
- Defining schema up front vs. schema-on-read
  - Relieving Spark from the onus of inferring data types
    - Q: Why is this important for ingesting large data sets?
  - Spark does not need to create a separate job for the purpose of reading a large portion of the file to ascertain the schema
  - Provides data type validation on columns
  - When reading a large amount of data, schema should be defined up front.
- Schema can be defined in two ways
  - Programmatically
  - Using Data Definition Language (DDL) string

Working with columns

DataFrame Columns

- Named columns are conceptually similar to columns in RDBMS tables.
- Named column describe a type of field.
- Columns are objects with public methods represented by the Column type.
  - Spark documentation refers to both *col()* and *Column*
  - *col()* is a standard built-in function that returns a *Column*

DataFrame Columns

- Logical or mathematical expressions can be used on columns
  - expr("columnName * 5") or

    (expr("columnName - 5") > col(anothercolumnName))
  - expr() is part of the **pyspark.sql.functions** (Python) and **org.apache.spark.sql.functions** (Scala) packages.
- Each Column object in a DataFrame is a part of a Row object.
- All Rows form a DataFrame

pyspark.sql.Column Cheat Sheet

API	Description	Example
alias(**alias*, ***kwargs*)	Returns this column aliased with a new name or names (in the case of expressions that return more than one column, such as explode).	df.select(df.age.alias(“age2”)).collect()
asc()	Returns a sort expression based on ascending order of the column.	df.select(df.name).orderBy(df.name.asc()).collect()
astype(dataType)	astype() is an alias for cast().	*See cast()*
between(lowerBound, upperBound)	A boolean expression that is evaluated to true if the value of this expression is between the given columns.	df.select(df.name, df.age.between(2, 4)).show()
bitwiseAND(other)	Compute bitwise AND of this expression with another expression.	df.select(df.a.bitwiseAND(df.b)).collect()
bitwiseOR(other)	Compute bitwise OR of this expression with another expression.	df.select(df.a.bitwiseOR(df.b)).collect()
bitwiseXOR(other)	Compute bitwise XOR of this expression with another expression.	df.select(df.a.bitwiseXOR(df.b)).collect()
cast(dataType)[source]	Convert the column into type dataType.	df.select(df.age.cast("string").alias('ages')).collect()
contains(other)	Contains the other element. Returns a boolean Column based on a string match.	df.filter(df.name.contains('o')).collect()

https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.Column.html

pyspark.sql.Column Cheat Sheet

API	Description	Example
desc()	Returns a sort expression based on the descending order of the column.	df.select(df.name).orderBy(df.name.desc()).collect()
endswith(other)	Equality test that is safe for null values.	df.filter(df.name.endswith('ice')).collect()
eqNullSafe(other)	astype() is an alias for cast().	df2.select(... df2['value'].eqNullSafe(None), df2['value'].eqNullSafe(float('NaN')), df2['value'].eqNullSafe(42.0)).show()
isNotNull()	True if the current expression is NOT null.	df.filter(df.height.isNotNull()).collect()
isNull()	True if the current expression is null.	df.filter(df.height.isNull()).collect()
isin(*cols)	A boolean expression that is evaluated to true if the value of this expression is contained by the evaluated values of the arguments.	df[df.name.isin("Bob", "Mike")].collect()
like(other)	SQL like expression. Returns a boolean Column based on a SQL LIKE match.	ddf.filter(df.name.like('Al%')).collect()
startswith(other)	String starts with. Returns a boolean Column based on a string match.	df.filter(df.name.startswith('Al')).collect()
substr(startPos, length)	Return a Column which is a substring of the column..	df.select(df.name.substr(1, 3).alias("col")).collect()
when(condition, value)	Evaluates a list of conditions and returns one of multiple possible result expressions. If Column.otherwise() is not invoked, None is returned for unmatched conditions.	df.select(df.name, F.when(df.age > 4, 1).when(df.age < 3, -1).otherwise(0)).show()

https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.Column.html

Working with rows

DataFrame Rows

- Each DataFrame column is part of a row
  - A row is an order collection of columns
- A *Row* in Spark is a **generic object type** and all rows together constitute a DataFrame
- Row objects can be used to create DataFrames

DataFrame Rows

- `Row` is an object in Spark and an ordered collection of fields
  – A `Row` object can be instantiated, and each of its fields can be accessed using an index (0-based).
- A row in DataFrame. The fields in it can be accessed:
  – like attributes (row.key)
  – like dictionary values (row[key])
- `key in row` will search through row keys.

DataFrame Rows

- Row objects can be used to create DataFrames

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

authors_df.show()
```

- In practice, contents of files are read in as DataFrames (along with schema information to make the reading more efficient).

DataFrame Rows

- Row can be used to create another Row-like class

```
Person = Row("name", "age")
>> <Row('name', 'age')>
Person("Alice", 11)
>> Row(name='Alice', age=11)
```

https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.Row

DataFrame Operations

- Loading of DataFrames from a data source:
  - *DataFrameReader*, an interface for reading data into a DataFrame
  - A large list of data sources including JSON, CSV, Parquet, Text, Avro, ORC, etc.
- Writing DataFrames to files or other targets:
  - *DataFrameWriter*, an interface for writing data back to a data source
  - Parquet using snappy compression is the default format
  - The schema is included in the Parquet file itself – no need to manually specify a schema during read
- Both support multiple data sources

DataFrame Operations

- Reading

```
# Programmatic way to define a schema

fire_schema = StructType([StructField('CallNumber', IntegerType(), 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)
```

Q: What is the *spark* object above?

DataFrame Operations

- Writing to a file

```
parquet_path = ...

fire_df.write.format("parquet").save(parquet_path)
```

- Saving as a table

```
parquet_table = ... # name of the table

fire_df.write.format("parquet").saveAsTable(parquet_table)
```

  – Saving a DataFrame as a table registers metadata with the Hive metadata store.

DataFrame Operations

- Projection – returning rows matching certain conditions using filtering
  - *select()* method is used
  - Filters can be expressed using *filter() or where()* method

```
few_fire_df = (fire_df
.select("IncidentNumber", "AvailableDtTm", "CallType")
.where(col("CallType") != "Medical Incident"))

few_fire_df.show(5, truncate=False)
```

DataFrame Operations

- Column manipulation
  - Renaming, adding and dropping columns using desired column names in the schema with *StructField.* e.g. when there are spaces in the column name.
  - A column can be renamed using *withColumnRenamed()* method.

```
new_fire_df = fire_df.withColumnRenamed("Delay", "ResponseDelayedinMins")

(new_fire_df
.select("ResponseDelayedinMins")
.where(col("ResponseDelayedinMins") > 5)
.show(5, False))
```

```
fire_ts_df = (new_fire_df
.withColumn("IncidentDate", to_timestamp(col("CallDate"), "MM/dd/yyyy"))
.drop("CallDate"))
```

For date time formats refer here: https://spark.apache.org/docs/latest/sql-ref-datetime-pattern.html

DataFrame Operations

- Date Functions
  - Date string format
  - spark.sql.functions
  - month(), year(), day()
- Aggregations
  - groupBy(), orderBy(), count()

```
(fire_ts_df
.select(year('IncidentDate'))
.distinct()
.orderBy(year('IncidentDate'))
.show())
```

DataFrame Operations

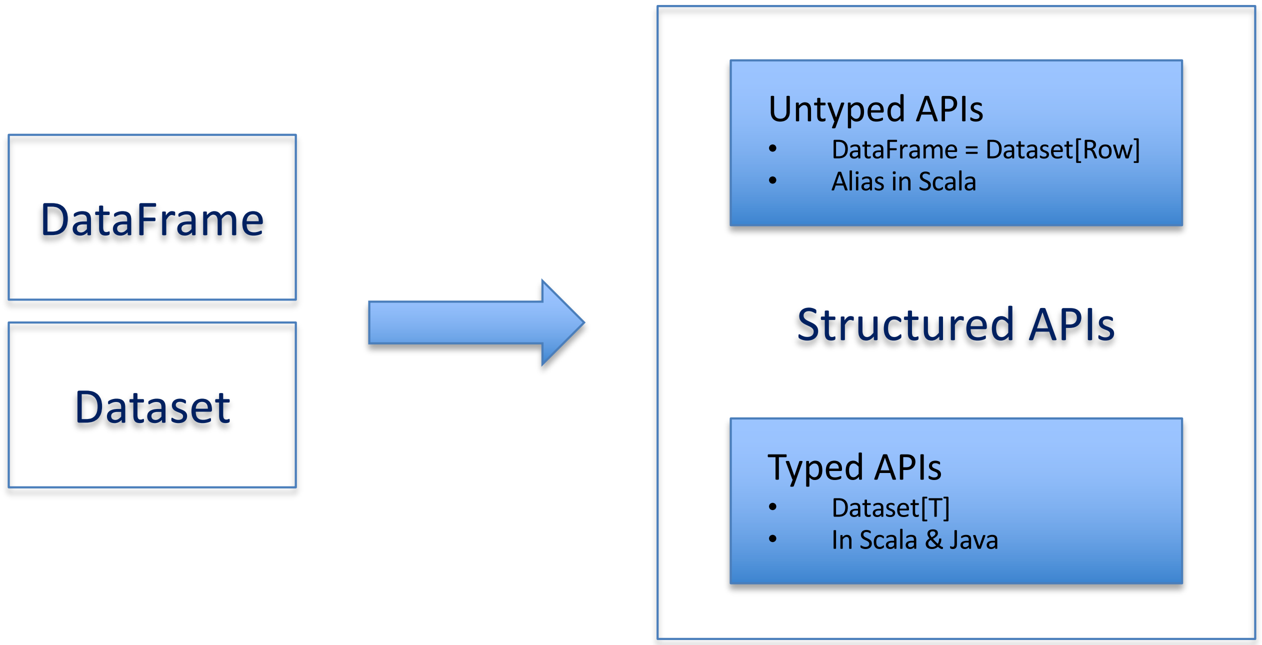
- Collect() method
  - DataFrame API supports *collect()* method
  - * Note: collect() is an action vs. select() is a transformation.
  - For large data size, this can cause out-of-memory (OOM) exceptions.
- Statistical methods
  - min(), max(), sum(), avg(), state(), describe(), correlation(), covariance(), sampleBy(), apporxQuantile(), frequentItems()

Dataset API

Dataset API

- DataFrame and Dataset APIs are unified as Structured APIs with similar interfaces in Spark 2.0
- Dataset can be either type or untyped
- In Scala, DataFrame is an alias for collection of *Dataset[Row]* objects, where *Row* is a generic untyped JVM object that can hold different types of fields

Structured APIs in Apache Spark



Note: Dataset is not available in Python

Typed vs Untyped Objects

- Since types are bound to variables and objects at compile time, in Spark, only Java and Scala supports Datasets.
- Python and R supports DataFrames
  - Not compile time type-safe
  - Types are dynamically inferred and assigned during execution
- In Scala, DataFrame is an alias for untyped Dataset[Row].
- *Row* is a generic object type in Spark
  - Holds a collection of mixed types that can be accessed using an index.
  - Internally, Spark manipulates Row objects, converting them to the equivalent types. E.g. Int => IntegerType for Scala, or IntegerType for Python

Dataset Operations

- Operations filter(), map(), groupBy(), select(), take(), etc. are similar to the ones on DataFrames.
- DataFrames – *filter()* conditions are SQL-like DSL operations which are language agnostic vs. Datasets – language-native expressions in Scala and Java code
- Datasets are similar to RDDs
  - Provide similar interface to methods and compile time safety, but with much easier to read and an object-oriented programming (OOP) interface.
- Spark SQL engine handles the creation, conversion, serialization and deserialization of the Datasets JVM objects along.
- Via Dataset encoders, Spark SQL engine also takes care of off-Java heap memory management.

DataFrames vs. Datasets

- If you want to tell Spark what to do, not how to do it, use DataFrames or Datasets.
- If you want rich semantics, high-level abstractions, and DSL operators, use DataFrames or Datasets.
- If you want strict compile-time type safety and don't mind creating multiple case classes for a specific Dataset[T], use Datasets.
- If your processing demands high-level expressions, filters, maps, aggregations, computing averages or sums, SQL queries, columnar access, or use of relational operators on semi-structured data, use DataFrames or Datasets.
- If your processing dictates relational transformations similar to SQL-like queries, use DataFrames.
- If you want to take advantage of and benefit from Tungsten's efficient serialization with Encoders, use Datasets.
- If you want unification, code optimization, and simplification of APIs across Spark components, use DataFrames.
- If you are an R user, use DataFrames.
- If you are a Python user, use DataFrames and drop down to RDDs if you need more control.
- If you want space and speed efficiency, use DataFrames.
- If you want errors caught during compilation rather than at runtime, choose the appropriate API

From page 74 of "Learning Spark"

Spark SQL Intro

Spark SQL

- Allows developers to work with standard SQL (ANSI SQL:2003) to make queries on structured data with a schema
  - High-level structured functionalities have been built on top of it (DataFrame and Dataset).
- Unifies Spark components and simplifies working with structured data sets by providing abstractions on DataFrames/Datasets.
- Connects to Apache Hive metastore and tables

Spark SQL

- Reads and writes structured data with a specific schema from structured file formats (JSON, CSV, Text, Avro, Parquet, ORC, etc.) and converts data into temporary tables.
- Connecting with external tools, including BI tools (Power BI, Tableau, SAS), via JDBC and ODBC
- Optimization via Catalyst Optimizer and Project Tungsten.
  - These support high-level DataFrame, Dataset APIs and SQL queries.

Catalyst Optimizer

- Takes a computation query and converts it into an execution plan.
- Four Transformational phases:
  - Phase 1: Analysis
    - Abstract Syntax Tree (AST) is generated for the SQL/DataFrame query
    - Any columns or table names will be resolved by consulting an internal Catalog
  - Phase 2: Logical Optimization
    - Construct a set of multiple plans and then, using its cost-based optimizer (CBO), assign costs to each plan which are laid out as operator trees
    - Logical plan becomes the input for the physical plan
  - Phase 3: Physical Planning
    - Selection of the optimal physical plan based on the logical plan
  - Phase 4: Code Generation
    - Project Tungsten facilitates whole-stage code generation to generate efficient Java bytecode.

Catalyst Optimizer

- In Python, use *explain(True)* method on DataFrames to see the different stages the code goes through.
- In Scala, call *df.queryExecution.logical*, or *df.queryExecution.optimizedPlan*

Spark SQL and DataFrames

- *SparkSesssion* – entry point for programming Spark with the Structured APIs.
  - Provides *sql()* method for SQL queries
    - All SQL queries return a DataFrame
    - ANSI:2003-compliance SQL
- Any DataFrame can be registered as a table or view (a temporary table) and query it using SQL.
  - No performance difference between using SQL or DataFrame API
  - Both compile to the same underlying plan specified in DataFrame code.

Important PySpark modules

- pyspark.sql.SparkSession Main entry point for DataFrame and SQL functionality.
- pyspark.sql.DataFrame A distributed collection of data grouped into named columns.
- pyspark.sql.Column A column expression in a DataFrame.
- pyspark.sql.Row A row of data in a DataFrame.
- pyspark.sql.GroupedData Aggregation methods, returned by DataFrame.groupBy().
- pyspark.sql.DataFrameNaFunctions Methods for handling missing data (null values).
- pyspark.sql.DataFrameStatFunctions Methods for statistics functionality.
- pyspark.sql.functions List of built-in functions available for DataFrame.
- pyspark.sql.types List of data types available.
- pyspark.sql.Window For working with window functions.

https://spark.apache.org/docs/latest/api/python/pyspark.sql.html

Questions

Homework

Baltimore City Crime Data

https://data.baltimorecity.gov/datasets/part1-crime-data/explore?showTable=true

Directions – Using Spark DataFrames:

1. Specify the schema for the crime data set.
2. Read the file using the schema definition
3. Cache the DataFrame
4. Show the count of the rows
5. Print the schema
6. Display first 5 rows
7. Answer the following questions **using DataFrame operations**

Questions

1. What are distinct crime codes?
2. Count the number of crimes by the crime codes and order by the resulting counts in descending order
3. Which neighborhood had most crimes?
4. Which month of the year had most crimes?
5. What weapons were used?
6. Which weapon was used the most?