Applying Schemas to JSON Data

Apache Spark™ and Databricks® provide a number of ways to project structure onto semi-structured data allowing for quick and easy access.

In this lesson you:

- Infer the schema from JSON files
- Create and use a user-defined schema with primitive data types
- Use non-primitive data types such as `ArrayType` and `MapType` in a schema

Schemas

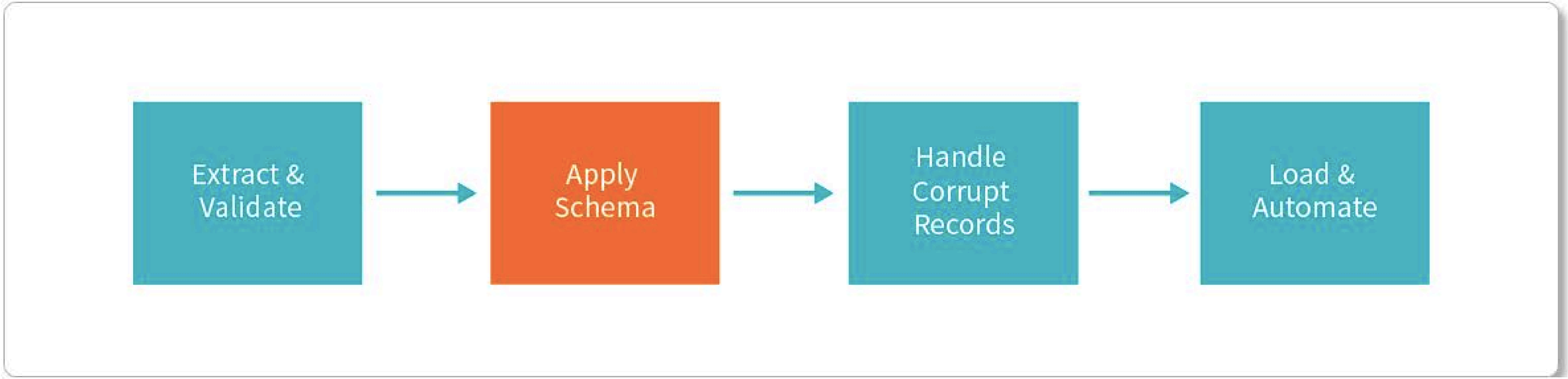
Schemas are at the heart of data structures in Spark. **A schema describes the structure of your data by naming columns and declaring the type of data in that column.** Rigorously enforcing schemas leads to significant performance optimizations and reliability of code.

Why is open source Spark so fast, and why is Databricks Runtime even faster? While there are many reasons for these performance improvements, two key reasons are:

- First and foremost, Spark runs first in memory rather than reading and writing to disk.
- Second, using DataFrames allows Spark to optimize the execution of your queries because it knows what your data looks like.

Two pillars of computer science education are data structures, the organization and storage of data and algorithms, and the computational procedures on that data. A rigorous understanding of computer science involves both of these domains. When you apply the most relevant data structures, the algorithms that carry out the computation become significantly more eloquent.

In the road map for ETL, this is the **Apply Schema** step:



Cmd 7

Schemas with Semi-Structured JSON Data

Tabular data, such as that found in CSV files or relational databases, has a formal structure where each observation, or row, of the data has a value (even if it's a NULL value) for each feature, or column, in the data set.

Semi-structured data does not need to conform to a formal data model. Instead, a given feature may appear zero, once, or many times for a given observation.

Semi-structured data storage works well with hierarchical data and with schemas that may evolve over time. One of the most common forms of semi-structured data is JSON data, which consists of attribute-value pairs.

Cmd 9

Print the first few lines of a JSON file holding ZIP Code data.

Cmd 10

```
%fs head /mnt/training/zips.json
```


```
[Truncated to first 65536 bytes]
{ "_id" : "01001", "city" : "AGAWAM", "loc" : [ -72.622739, 42.070206 ], "pop" : 15338, "state" : "MA" }
{ "_id" : "01002", "city" : "CUSHMAN", "loc" : [ -72.51564999999999, 42.377017 ], "pop" : 36963, "state" : "MA" }
{ "_id" : "01005", "city" : "BARRE", "loc" : [ -72.10835400000001, 42.409698 ], "pop" : 4546, "state" : "MA" }
{ "_id" : "01007", "city" : "BELCHERTOWN", "loc" : [ -72.41095300000001, 42.275103 ], "pop" : 10579, "state" : "MA" }
{ "_id" : "01008", "city" : "BLANDFORD", "loc" : [ -72.936114, 42.182949 ], "pop" : 1240, "state" : "MA" }
{ "_id" : "01010", "city" : "BRIMFIELD", "loc" : [ -72.188455, 42.116543 ], "pop" : 3706, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [ -72.988761, 42.279421 ], "pop" : 1688, "state" : "MA" }
{ "_id" : "01012", "city" : "CHESTERFIELD", "loc" : [ -72.833309, 42.38167 ], "pop" : 177, "state" : "MA" }
{ "_id" : "01013", "city" : "CHICOPEE", "loc" : [ -72.607962, 42.162046 ], "pop" : 23396, "state" : "MA" }
{ "_id" : "01020", "city" : "CHICOPEE", "loc" : [ -72.576142, 42.176443 ], "pop" : 31495, "state" : "MA" }
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [ -72.558657, 42.196672 ], "pop" : 1764, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [ -72.905767, 42.435296 ], "pop" : 1484, "state" : "MA" }
{ "_id" : "01027", "city" : "MOUNT TOM", "loc" : [ -72.67992099999999, 42.264319 ], "pop" : 16864, "state" : "MA" }
{ "_id" : "01028", "city" : "EAST LONGMEADOW", "loc" : [ -72.505565, 42.067203 ], "pop" : 13367, "state" : "MA" }
{ "_id" : "01030", "city" : "FEEDING HILLS", "loc" : [ -72.675077, 42.07182 ], "pop" : 11985, "state" : "MA" }
{ "_id" : "01031", "city" : "GILBERTVILLE", "loc" : [ -72.19858499999999, 42.332194 ], "pop" : 2385, "state" : "MA" }
{ "_id" : "01032", "city" : "GOSHEN", "loc" : [ -72.844092, 42.466234 ], "pop" : 122, "state" : "MA" }
{ "_id" : "01033", "city" : "GRANBY", "loc" : [ -72.52000099999999, 42.255704 ], "pop" : 5526, "state" : "MA" }
{ "_id" : "01034", "city" : "TOLLAND", "loc" : [ -72.908793, 42.070234 ], "pop" : 1652, "state" : "MA" }
{ "_id" : "01035", "city" : "HADLEY", "loc" : [ -72.571499, 42.36062 ], "pop" : 4231, "state" : "MA" }
{ "_id" : "01036", "city" : "HAMPDEN", "loc" : [ -72.43182299999999, 42.064756 ], "pop" : 4709, "state" : "MA" }
{ "_id" : "01038", "city" : "HATFIELD", "loc" : [ -72.61673500000001, 42.38439 ], "pop" : 3184, "state" : "MA" }
{ "_id" : "01039", "city" : "HAYDENVILLE", "loc" : [ -72.70317799999999, 42.381799 ], "pop" : 1387, "state" : "MA" }
```

Schema Inference

Import data as a DataFrame and view its schema with the `printSchema()` DataFrame method.

Cmd 12

```
zipsDF = spark.read.json("/mnt/training/zips.json")
zipsDF.printSchema()
```

▸ (1) Spark Jobs

▸ zipsDF: pyspark.sql.dataframe.DataFrame = [_id: string, city: string ... 3 more fields]

```
root
 |-- _id: string (nullable = true)
 |-- city: string (nullable = true)
 |-- loc: array (nullable = true)
 |    |-- element: double (containsNull = true)
 |-- pop: long (nullable = true)
 |-- state: string (nullable = true)
```

Command took 2.36 seconds -- by huseyinyilmaz01@gmail.com at 4/27/2020, 2:28:11 AM on My Cluster

Store the schema as an object by calling `.schema` on a DataFrame. Schemas consist of a `StructType`, which is a collection of `StructField`s. Each `StructField` gives a name and a type for a given field in the data.

Cmd 14

```
zipsSchema = zipsDF.schema
print(type(zipsSchema))

[field for field in zipsSchema]
```

```
<class 'pyspark.sql.types.StructType'>
Out[14]: [StructField(_id,StringType,true),
 StructField(city,StringType,true),
 StructField(loc,ArrayType(DoubleType,true),true),
 StructField(pop,LongType,true),
 StructField(state,StringType,true)]
```

Command took 0.03 seconds -- by huseyinyilmaz01@gmail.com at 4/27/2020, 2:28:48 AM on My Cluster

User-Defined Schemas

Spark infers schemas from the data, as detailed in the example above. Challenges with inferred schemas include:

- Schema inference means Spark scans all of your data, creating an extra job, which can affect performance
- Consider providing alternative data types (for example, change a `Long` to a `Integer`)
- Consider throwing out certain fields in the data, to read only the data of interest

To define schemas, build a `StructType` composed of `StructField` s.

Import the necessary types from the `types` module. Build a `StructType`, which takes a list of `StructField`s. Each `StructField` takes three arguments: the name of the field, the type of data in it, and a `Boolean` for whether this field can be `Null`.

Cmd 18

```
from pyspark.sql.types import StructType, StructField, IntegerType, StringType

zipsSchema2 = StructType([
  StructField("city", StringType(), True),
  StructField("pop", IntegerType(), True)
])
```

Apply the schema using the `.schema` method. This `read` returns only the columns specified in the schema and changes the column `pop` from `LongType` (which was inferred above) to `IntegerType`.

A `LongType` is an 8-byte integer ranging up to 9,223,372,036,854,775,807 while `IntegerType` is a 4-byte integer ranging up to 2,147,483,647. Since no American city has over two billion people, `IntegerType` is sufficient.

Cmd 20

```
zipsDF2 = (spark.read
  .schema(zipsSchema2)
  .json("/mnt/training/zips.json")
)

display(zipsDF2)
```

(1) Spark Jobs

zipsDF2: pyspark.sql.dataframe.DataFrame = [city: string, pop: integer]

city	pop
AGAWAM	15338
CUSHMAN	36963
BARRE	4546
BELCHERTOWN	10579
BLANDFORD	1240
BRIMFIELD	3706
CHESTER	1688
CHESTERFIELD	177

Primitive and Non-primitive Types

The Spark `types` package provides the building blocks for constructing schemas.

A primitive type contains the data itself. The most common primitive types include:

Numeric	General	Time
`FloatType`	`StringType`	`TimestampType`
`IntegerType`	`BooleanType`	`DateType`
`DoubleType`	`NullType`	
`LongType`		
`ShortType`		

Non-primitive types are sometimes called reference variables or composite types. Technically, non-primitive types contain references to memory locations and not the data itself. Non-primitive types are the composite of a number of primitive types such as an Array of the primitive type `Integer`.

The two most common composite types are `ArrayType` and `MapType`. These types allow for a given field to contain an arbitrary number of elements in either an Array/List or Map/Dictionary form.

See the Spark documentation for a complete picture of types in Spark.

The ZIP Code dataset contains an array with the latitude and longitude of the cities. Use an `ArrayType`, which takes the primitive type of its elements as an argument.

Cmd 24

```
from pyspark.sql.types import StructType, StructField, IntegerType, StringType, ArrayType, FloatType

zipsSchema3 = StructType([
  StructField("city", StringType(), True),
  StructField("loc",
    ArrayType(FloatType(), True), True),
  StructField("pop", IntegerType(), True)
])
```

Command took 0.04 seconds -- by huseyinyilmaz01@gmail.com at 4/27/2020, 3:33:08 AM on My Cluster

Cmd 25

Apply the schema using the `.schema()` method and observe the results. Expand the array values in the column `loc` to explore further.

Cmd 26

```
zipsDF3 = (spark.read
  .schema(zipsSchema3)
  .json("/mnt/training/zips.json")
)
display(zipsDF3)
```

(1) Spark Jobs

zipsDF3: pyspark.sql.dataframe.DataFrame = [city: string, loc: array ... 1 more fields]

city	loc	pop
AGAWAM	[-72.62274,42.070206]	15338
CUSHMAN	[-72.51565,42.377018]	36963
BARRE	[-72.10835,42.4097]	4546

Exercise 1: Exploring JSON Data

Smartphone data from UCI Machine Learning Repository is available under `/mnt/training/UbiqLog4UCI`. This is log data from the open source project Ubiqlog.

Import this data and define your own schema.

Cmd 28

Step 1: Import the Data

Import data from `/mnt/training/14_F/log*`. (This is the log files from a given user.)

Cmd 29

Look at the head of one file from the data set. Use `/mnt/training/UbiqLog4UCI/14_F/log_1-6-2014.txt`.

Cmd 30

```
# ANSWER
print(dbutils.fs.head('/mnt/training/UbiqLog4UCI/14_F/log_1-6-2014.txt', 200)) # this evaluates to the thing as %fs head /mnt/training/UbiqLog4UCI/14_F/log_1-6-2014.txt
```

```
[Truncated to first 200 bytes]
{"Application":{"ProcessName":"com.jb.gosms","Start":"1-5-2014 22:36:16","End":"1-5-2014 22:41:17"}}
{"Application":{"ProcessName":"com.jb.gosms","Start":"1-5-2014 22:41:17","End":"1-5-2014 22:46:18"}
```

Command took 0.16 seconds -- by huseyinyilmaz01@gmail.com at 4/27/2020, 3:42:32 AM on My Cluster

Cmd 31

Read the data and save it to `smartphoneDF` . Read the logs using a `*` in your path like `/mnt/training/UbiqLog4UCI/14_F/log*` .

Cmd 32

```
# ANSWER
smartphoneDF = spark.read.json("/mnt/training/UbiqLog4UCI/14_F/log*")
```

▸ (4) Spark Jobs

▸ smartphoneDF: pyspark.sql.dataframe.DataFrame = [Application: struct, Bluetooth: struct ... 5 more fields]

Command took 4.90 seconds -- by huseyinyilmaz01@gmail.com at 4/27/2020, 3:40:58 AM on My Cluster

Print the schema to get a sense for the data.

Cmd 35

```
# ANSWER
smartphoneDF.printSchema()
```

```
root
 |-- Application: struct (nullable = true)
 |    |-- End: string (nullable = true)
 |    |-- ProcessName: string (nullable = true)
 |    |-- Start: string (nullable = true)
 |-- Bluetooth: struct (nullable = true)
 |    |-- address: string (nullable = true)
 |    |-- bond status: string (nullable = true)
 |    |-- name: string (nullable = true)
 |    |-- time: string (nullable = true)
 |-- Call: struct (nullable = true)
 |    |-- Duration: string (nullable = true)
 |    |-- Number: string (nullable = true)
 |    |-- Time: string (nullable = true)
 |    |-- Type: string (nullable = true)
 |-- Location: struct (nullable = true)
 |    |-- Accuracy: string (nullable = true)
 |    |-- Altitude: string (nullable = true)
 |    |-- Latitude: string (nullable = true)
 |    |-- Longtitude: string (nullable = true)
 |    |-- Provider: string (nullable = true)
```

Command took 0.02 seconds -- by huseyinyilmaz01@gmail.com at 4/27/2020, 3:43:10 AM on My Cluster

Cmd 36

The schema shows:

- Six categories of tracked data
- Nested data structures
- A field showing corrupt records

Cmd 37

Exercise 2: Creating a User Defined Schema

Cmd 38

Step 1: Set Up Your workflow

Often the hardest part of a coding challenge is setting up a workflow to get continuous feedback on what you develop.

Start with the import statements you need, including functions from two main packages:

Package	Function
`pyspark.sql.types`	`StructType`, `StructField`, `StringType`
`pyspark.sql.functions`	`col`

Cmd 39

```
# ANSWER

from pyspark.sql.types import StructType, StructField, StringType
from pyspark.sql.functions import col
```

Command took 0.56 seconds -- by huseyinyilmaz01@gmail.com at 4/28/2020, 2:05:03 AM on My Cluster

Cmd 40

The **SMS** field needs to be parsed. Create a placeholder schema called `schema` that's a `StructType` with one `StructField` named **SMS** of type `StringType`. This imports the entire attribute (even though it contains nested entities) as a String.

This is a way to get a sense for what's in the data and make a progressively more complex schema.

Cmd 41

```
# ANSWER

from pyspark.sql.types import StructType, StructField, StringType
from pyspark.sql.functions import col

schema = StructType([
  StructField("SMS", StringType(), True)
])
```

Command took 0.04 seconds -- by huseyinyilmaz01@gmail.com at 4/28/2020, 2:06:44 AM on My Cluster

Apply the schema to the data and save the result as `SMSDF`. This closes the loop on which to iterate and develop an increasingly complex schema. The path to the data is `/mnt/training/UbiqLog4UCI/14_F/log*`.

Include only records where the column `SMS` is not `Null`.

Cmd 44

```
# ANSWER
from pyspark.sql.types import StructType, StructField, StringType
from pyspark.sql.functions import col

schema = StructType([
  StructField("SMS", StringType(), True)
])

SMSDF = (spark.read
  .schema(schema)
  .json("/mnt/training/UbiqLog4UCI/14_F/log*")
  .filter(col("SMS").isNotNull())
)

display(SMSDF)
```

▸ (4) Spark Jobs

▸ SMSDF: pyspark.sql.dataframe.DataFrame = [SMS: string]

SMS
{"Address":"+98214428####","type":"1","date":"1-10-2014 11:30:05","body":"ANONYMIZED","Type":"1","metadata":{"name":""}}
{"Address":"+985000406500####","type":"1","date":"1-10-2014 11:32:01","body":"ANONYMIZED","Type":"1","metadata":{"name":""}}

Step 2: Create the Full Schema for SMS

Define the Schema for the following fields in the `StructType` `SMS` and name it `schema2`. Apply it to a new DataFrame `SMSDF2`:

- `Address`
- `date`
- `metadata`
  - `name`

Note there's `Type` and `type`, which appears to be redundant data.

```python
from pyspark.sql.types import StructType, StructField, StringType
from pyspark.sql.functions import col

schema2 = StructType([
  StructField("SMS", StructType([
    StructField("Address",StringType(),True),
    StructField("date",StringType(),True),
    StructField("metadata", StructType([
      StructField("name",StringType(), True)
    ]), True),
  ]), True)
])

# Here is the full schema as well
# fullSchema = StructType([
#   StructField("SMS", StructType([
#     StructField("Address",StringType(),True),
#     StructField("Type",StringType(),True),
#     StructField("body",StringType(),True),
#     StructField("date",StringType(),True),
#     StructField("metadata", StructType([
#       StructField("name",StringType(), True)
#     ]), True),
#     StructField("type",StringType(),True)
#   ]), True)
# ])

SMSDF2 = (spark.read
  .schema(schema2)
  .json("/mnt/training/UbiqLog4UCI/14_F/log*")
  .filter(col("SMS").isNotNull()))

display(SMSDF2)
```

Step 3: Compare Solution Performance

Compare the dafault schema inference to applying a user defined schema using the `%timeit` function. Which completed faster? Which triggered more jobs? Why?

Cmd 50

```
%timeit SMSDF = spark.read.schema(schema2).json("/mnt/training/UbiqLog4UCI/14_F/log*").count()
```

▸ (16) Spark Jobs

```
3.86 s ± 466 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Command took 31.75 seconds -- by huseyinyilmaz01@gmail.com at 4/28/2020, 2:15:23 AM on My Cluster

Cmd 51

```
%timeit SMSDF = spark.read.json("/mnt/training/UbiqLog4UCI/14_F/log*").count()
```

▸ (40) Spark Jobs

```
5.87 s ± 558 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Command took 48.43 seconds -- by huseyinyilmaz01@gmail.com at 4/28/2020, 2:15:59 AM on My Cluster

Cmd 52

Providing a schema increases performance two to three times, depending on the size of the cluster used. Since Spark doesn't infer the schema, it doesn't have to read through all of the data. This is also why there are fewer jobs when a schema is provided: Spark doesn't need one job for each partition of the data to infer the schema.

Review

Question: What are two ways to attain a schema from data?
Answer: Allow Spark to infer a schema from your data or provide a user defined schema. Schema inference is the recommended first step; however, you can customize this schema to your use case with a user defined schema.

Question: Why should you define your own schema?
Answer: Benefits of user defined schemas include:
- Avoiding the extra scan of your data needed to infer the schema
- Providing alternative data types
- Parsing only the fields you need

Question: Why is JSON a common format in big data pipelines?
Answer: Semi-structured data works well with hierarchical data and where schemas need to evolve over time. It also easily contains composite data types such as arrays and maps.

Question: By default, how are corrupt records dealt with using `spark.read.json()` ?
Answer: They appear in a column called `_corrupt_record` . These are the records that Spark can't read (e.g. when characters are missing from a JSON string).