Manipulating DataFrames

Cleaning Data with PySpark

Overview

1. DataFrame column operations
2. Conditional DataFrame column operations
3. User defined functions
4. Partitioning and lazy processing

DataFrame column operations

DataFrame refresher

DataFrames:

- Made up of rows & columns
- Immutable
- Use various transformation operations to modify data

```
# Return rows where name starts with "M"
voter_df.filter(voter_df.name.like('M%'))
# Return name and position only
voters = voter_df.select('name', 'position')
```

Common Data Frame transformations

Filter / Where

```
voter_df.filter(voter_df.date > '1/1/2019') # or voter_df.where(...)
```

Select

```
voter_df.select(voter_df.name)
```

withColumn

```
voter_df.withColumn('year', voter_df.date.year)
```

drop

```
voter_df.drop('unused_column')
```

Conditional DataFrame column operations

Conditional clauses

Conditional Clauses are:

- Inline version of if/then/else
- .when()
- .otherwise()

Conditional example

```
.when(<if condition>, <then x>)
```

```
df.select(df.Name, df.Age, F.when(df.Age >= 18, "Adult"))
```

Name	Age	
Alice	14	
Bob	18	Adult
Candice	38	Adult

Another example

Multiple .when()

```
df.select(df.Name, df.Age,
          .when(df.Age >= 18, "Adult")
          .when(df.Age < 18, "Minor"))
```

Name	Age	
Alice	14	Minor
Bob	18	Adult
Candice	38	Adult

Otherwise

`.otherwise()` is like `else`

```
df.select(df.Name, df.Age,
          .when(df.Age >= 18, "Adult")
          .otherwise("Minor"))
```

Name	Age	
Alice	14	Minor
Bob	18	Adult
Candice	38	Adult

User defined functions

Defined

User defined functions or UDFs

- Python method
- Wrapped via the **pyspark.sql.functions.udf** method
- Stored as a variable
- Called like a normal Spark function

Reverse string UDF

Define a Python method

```
def reverseString(mystr):
    return mystr[::-1]
```

Wrap the function and store as a variable

```
udfReverseString = udf(reverseString, StringType())
```

Use with Spark

```
user_df = user_df.withColumn('ReverseName',
                  udfReverseString(user_df.Name))
```

Argument-less example

```
def sortingCap():
    return random.choice(['G', 'H', 'R', 'S'])
udfSortingCap = udf(sortingCap, StringType())
user_df = user_df.withColumn('Class', udfSortingCap())
```

Name	Age	Class
Alice	14	H
Bob	18	S
Candice	63	G

Partitioning and lazy processing

Partitioning

- DataFrames are broken up into partitions
- Partition size can vary
- Each partition is handled independently

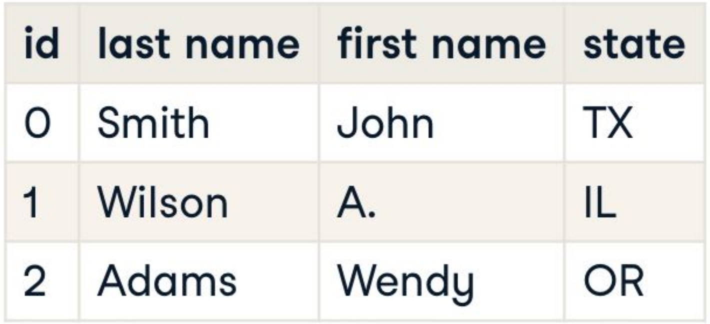
Lazy processing

1. Transformations are lazy
   a. .withColumn(...)
   b. .select(...)
2. Nothing is actually done until an action is performed
   a. .count()
   b. .write()
3. Transformations can be re-ordered for best performance
4. Sometimes causes unexpected behavior

Adding IDs

Normal ID fields:

- Common in relational databases
- Most usually an integer increasing, sequential, and unique
- Not very parallel

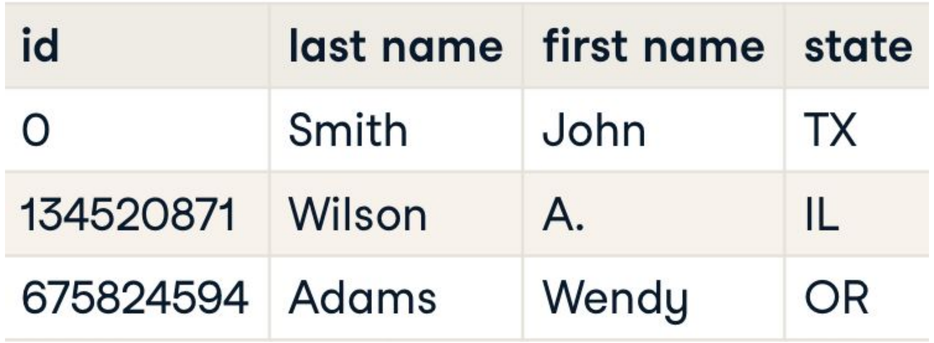


id	last name	first name	state
0	Smith	John	TX
1	Wilson	A.	IL
2	Adams	Wendy	OR

Monotonically increasing IDs

pyspark.sql.functions.monotonically_increasing_id()

- Integer (64-bit), increases in value, unique
- Not necessarily sequential (gaps exist)
- Completely parallel



id	last name	first name	state
0	Smith	John	TX
134520871	Wilson	A.	IL
675824594	Adams	Wendy	OR

Notes

Remember, Spark is lazy!

- Occasionally out of order
- If performing a join, ID may be assigned after the join
- Test your transformations