Cleaning Data with PySpark (Manipulating DataFrame in real world)

Filtering column content with Python

You've looked at using various operations on DataFrame columns - now you can modify a real dataset. The DataFrame `voter_df` contains information regarding the voters on the Dallas City Council from the past few years. This truncated DataFrame contains the date of the vote being cast and the name and position of the voter. Your manager has asked you to clean this data so it can later be integrated into some desired reports. The primary task is to remove any null entries or odd characters and return a specific set of voters where you can validate their information.

This is often one of the first steps in data cleaning - removing anything that is obviously outside the format. For this dataset, make sure to look at the original data and see what looks out of place for the `VOTER_NAME` column.

The `pyspark.sql.functions` library is already imported under the alias `F` .

```
# Show the distinct VOTER_NAME entries
voter_df.select('VOTER_NAME').distinct().show(40, truncate=False)

# Filter voter_df where the VOTER_NAME is 1-20 characters in length
voter_df = voter_df.filter('length(VOTER_NAME) > 0 and length(VOTER_NAME) < 20')

# Filter out voter_df where the VOTER_NAME contains an underscore
voter_df = voter_df.filter(~ F.col('VOTER_NAME').contains('_'))

# Show the distinct VOTER_NAME entries again
voter_df.select('VOTER_NAME').distinct().show(40, truncate=False)
```

Modifying DataFrame columns

Previously, you filtered out any rows that didn't conform to something generally resembling a name. Now based on your earlier work, your manager has asked you to create two new columns - `first_name` and `last_name` . She asks you to split the `VOTER_NAME` column into words on any space character. You'll treat the last word as the `last_name` , and all other words as the `first_name` . You'll be using some new functions in this exercise including `.split()` ,

`.size()` , and `.getItem()` .

Please note that these operations are always somewhat specific to the use case. Having your data conform to a format often matters more than the specific details of the format. Rarely is a data cleaning task meant just for one person - matching a defined format allows for easier sharing of the data later (ie, Paul doesn't need to worry about names - Mary already cleaned the dataset).

The filtered voter DataFrame from your previous exercise is available as `voter_df` . The `pyspark.sql.functions` library is available under the alias `F` .

```
# Add a new column called splits separated on whitespace
voter_df = voter_df.withColumn('splits', F.split(voter_df.VOTER_NAME, '\s+'))

# Create a new column called first_name based on the first item in splits
voter_df = voter_df.withColumn('first_name', voter_df.splits.getItem(0))

# Get the last entry of the splits list and create a column called last_name
voter_df = voter_df.withColumn('last_name', voter_df.splits.getItem(F.size('splits') - 1))

# Drop the splits column
voter_df = voter_df.drop('splits')

# Show the voter_df DataFrame
voter_df.show()
```

When

The `when()` clause lets you conditionally modify a Data Frame based on its content. You'll want to modify our `voter_df` DataFrame to add a random number to any voting member that is defined as a "Councilmember".

The `voter_df` DataFrame is defined and available to you. The `pyspark.sql.functions` library is available as `F.` You can use `F.rand()` to generate the random value.

```
# Add a column to voter_df for any voter with the title **Councilmember**
voter_df = voter_df.withColumn('random_val',
                               F.when(voter_df.TITLE == 'Councilmember', F.rand()))

# Show some of the DataFrame rows, noting whether the when clause worked
voter_df.show()
```

This requirement is similar to the last, but now you want to add multiple values based on the voter's position. Modify your `voter_df` DataFrame to add a random number to any voting member that is defined as a `Councilmember` . Use 2 for the `Mayor` and 0 for anything other

position.

The `voter_df` Data Frame is defined and available to you. The `pyspark.sql.functions` library is available as `F.` You can use `F.rand()` to generate the random value.

```
# Add a column to voter_df for a voter based on their position
voter_df = voter_df.withColumn('random_val',
                               when(voter_df.TITLE == 'Councilmember', F.rand())
                               .when(voter_df.TITLE == 'Mayor', 2)
                               .otherwise(0))

# Show some of the DataFrame rows
voter_df.show()

# Use the .filter() clause with random_val
voter_df.filter(voter_df.random_val == 0).show()
```

Using User defined functions in Spark

You've seen some of the power behind Spark's built-in string functions when it comes to manipulating DataFrames. However, once you reach a certain point, it becomes difficult to process the data in a without creating a rat's nest of function calls. Here's one place where you can use User Defined Functions to manipulate our DataFrames.

For this exercise, we'll use our `voter_df` DataFrame, but you're going to replace the `first_name` column with the first and middle names.

The `pyspark.sql.functions` library is available under the alias `F` . The classes from `pyspark.sql.types` are already imported.

```
def getFirstAndMiddle(names):
  # Return a space separated string of names
  return ' '.join(names)

# Define the method as a UDF
udfFirstAndMiddle = F.udf(getFirstAndMiddle, StringType())

# Create a new column using your UDF
voter_df = voter_df.withColumn('first_and_middle_name', udfFirstAndMiddle('splits'))

# Drop the unnecessary columns then show the DataFrame
voter_df = voter_df.drop('first_name')
voter_df = voter_df.drop('splits')
voter_df.show()
```

Adding an ID Field

When working with data, you sometimes only want to access certain fields and perform various operations. In this case, find all the **unique** voter names from the DataFrame and add a unique ID number. Remember that Spark IDs are assigned based on the DataFrame **partition** - as such the ID values may be much greater than the actual number of rows in the DataFrame.

With Spark's *lazy* processing, the IDs are not actually generated until an action is performed and can be somewhat random depending on the size of the dataset.

The `spark` session and a Spark DataFrame `df` containing the `DallasCouncilVotes.csv.gz` file are available in your workspace. The `pyspark.sql.functions` library is available under the alias `F`.

```
# Select all the unique council voters
voter_df = df.select(df["VOTER NAME"]).distinct()

# Count the rows in voter_df
print("\nThere are %d rows in the voter_df DataFrame.\n" % voter_df.count())

# Add a ROW_ID
voter_df = voter_df.withColumn('ROW_ID', F.monotonically_increasing_id())

# Show the rows with 10 highest IDs in the set
voter_df.orderBy(voter_df.ROW_ID.desc()).show(10)
```

IDs with different partitions

You've just completed adding an ID field to a DataFrame. Now, take a look at what happens when you do the same thing on DataFrames containing a different number of partitions.

To check the number of partitions, use the method `.rdd.getNumPartitions()` on a DataFrame.

The `spark` session and two DataFrames, `voter_df` and `voter_df_single`, are available in your workspace. The instructions will help you discover the difference between the DataFrames. The `pyspark.sql.functions` library is available under the alias `F`.

```
# Print the number of partitions in each DataFrame
print("\nThere are %d partitions in the voter_df DataFrame.\n" %
voter_df.rdd.getNumPartitions())
print("\nThere are %d partitions in the voter_df_single DataFrame.\n" %
voter_df_single.rdd.getNumPartitions())

# Add a ROW_ID field to each DataFrame
```

```
voter_df = voter_df.withColumn('ROW_ID', F.monotonically_increasing_id())
voter_df_single =  voter_df_single.withColumn('ROW_ID', F.monotonically_increasing_id())

# Show the top 10 IDs in each DataFrame
voter_df.orderBy(voter_df.ROW_ID.desc()).show(10)
voter_df_single.orderBy(voter_df_single.ROW_ID.desc()).show(10)
```

More ID tricks

Once you define a Spark process, you'll likely want to use it many times. Depending on your needs, you may want to start your IDs at a certain value so there isn't overlap with previous runs of the Spark task. This behavior is similar to how IDs would behave in a relational database. You have been given the task to make sure that the IDs output from a monthly Spark task start at the highest value from the previous month.

The `spark` session and two DataFrames, `voter_df_march` and `voter_df_april`, are available in your workspace. The `pyspark.sql.functions` library is available under the alias `F`.

```
# Determine the highest ROW_ID and save it in previous_max_ID
previous_max_ID = voter_df_march.select('ROW_ID').rdd.max()[0]

# Add a ROW_ID column to voter_df_april starting at the desired value
voter_df_april = voter_df_april.withColumn('ROW_ID', previous_max_ID +
F.monotonically_increasing_id())

# Show the ROW_ID from both DataFrames and compare
voter_df_march.select('ROW_ID').show()
voter_df_april.select('ROW_ID').show()
```