Data Cleaning with Apache Spark

Spark Schema

```
.printSchema()
```

```
# Import schema

import pyspark.sql.types
from pyspark.sql.types import *

peopleSchema = StructType([
# Define the name field
StructField('name', StringType(), True), # Define the name field
StructField('age', IntegerType(), True), # Add the age field
StructField('city', StringType(), True) ]) # Add the city field
#if false, means cannot be nullable.

people_df = spark.read.format('csv').load(name='rawdata.csv',
schema=peopleSchema)
```

Immutability and Lazy Processing

A component of functional programming
Defined once
Unable to be directly modied
Re-created if reassigned
Able to be shared efficiently

```
# Immutability Example
# Load the CSV file
aa_dfw_df = spark.read.format('csv').options(Header=True).load('AA_DFW_2018.csv

# Add the airport column using the F.lower() method
aa_dfw_df = aa_dfw_df.withColumn('airport', F.lower(aa_dfw_df['Destination Airpo

# Drop the Destination Airport column
aa_dfw_df = aa_dfw_df.drop(aa_dfw_df['Destination Airport'])

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

Understanding Parquet

Difficulties with CSV files

No defined schema (no data type, no column name beyond a header row)

Nested data requires special (content containing a comma requires escaping, using the escape character within content requires even further escaping)

handling Encoding format limited

for spark: slow to parse, cannot be shared during the import process; if no schema is defined, all data must be read before a schema can be inferred, forcing the code to read the file twice.

for spark: files cannot be filtered (no 'predicate pushdown', ordering tasks to do the least amount of work, filtering data prior to processing is one of the primary optimizations of predicate pushdown.)

for spark: any intermediate use requires redefining schema. Spark process are often multi-step and may utilize an intermediate file representation. These representations allow data to be used later without regenerating the data from source.

The Parquet Format

columnar data format

supported in spark and other data processing frameworks

supports predicate pushdown

automatically stores schema information

binary file format

Parquet is a compressed columnar data format developed for use in any Hadoop based system. Include: Spark, Hadoop, Apache Impala... Perfect for intermediary or on-disk representation of processed data.

predicate pushdown This means Spark will only process the data necessary to complete the operations you define versus reading the entire dataset.

Working with Parquet

```
df = spark.read.format('parquet').load('filename.parquet')
df = spark.read.parquet('filename.parquet')

df.write.format('parquet').save('filename.parquet')
df.write.parquet('filename.parquet') #mode='overwrite'
```

Parquet with SQL

parquet - dataframe - table

```
flight_df = spark.read.parquet('flights.parquet')
flight_df.createOrReplaceTempView('flights')
short_flights_df = spark.sql('SELECT * FROM flights WHERE flightduration <100')

# Run a SQL query of the average flight duration
avg_duration = spark.sql('SELECT avg(flight_duration) from flights').collect()[(
print('The average flight time is: %d' % avg_duration)
```

DataFrame column operations

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 DataFrame transformations

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

#Select
voter_df.select(voter_df.name)

#withColumn creates new column
voter_df.withColumn('year', voter_df.date.year)

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

Filtering data

Remove nulls

Remove odd entries

Split data from combined sources

Negate with ~

```
voter_df.filter(voter_df['name'].isNotNull())
voter_df.filter(voter_df.date.year > 1800)
voter_df.where(voter_df['_c0'].contains('VOTE'))
voter_df.where(~ voter_df._c1.isNull())
```

```
# 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)
```

Column string transformations

```
#Contained in pyspark.sql.functions
import pyspark.sql.functions as F

#Applied per column as transformation
voter_df.withColumn('upper', F.upper('name'))

#Can create intermediary columns
voter_df.withColumn('splits', F.split('name', ' '))

#Can cast to other types
voter_df.withColumn('year',voter_df[' c4'].cast(IntegerType()))
```

ArrayType() column functions

Various utility functions / transformations to interact with ArrayType() .

. `size()` - returns length of arrayType() column

. `getItem()` - used to retrieve a specific item at index of list column.

```
# 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('spl

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

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

Conditional DataFrame column operations
Conditional clauses

Conditional Clauses are:

Inline version of if / then / else

`.when()`

`.otherwise()`

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

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

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

# 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))
```

User defined functions

Python method

Wrapped via the `pyspark.sql.functions.udf` method

Stored as a variable

Called like a normal Spark function

```
def getFirstAndMiddle(names):
  # Return a space separated string of names
  return ' '.join(names[:-1])

# 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(voter_

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

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())
```

Argument-less example

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

Partitioning and lazy processing

Partitioning

DataFrames are broken up into partitions

Partition size can vary

Each partition is handled independently

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

Lazy processing

Transformations are lazy
* .withColumn(...)
* .select(...)
* .cache()

Nothing is actually done until an action is performed
* .count()
* .write(...)
*. show()

Transformations can be re-ordered for best performance

Sometimes causes unexpected behavior

Adding IDs

Normal ID fields:

Common in relational databases

Most usually an integer increasing, sequential and unique

Not very parallel

Monotonically increasing IDs

`pyspark.sql.functions.monotonically_increasing_id()`

Integer (64-bit), increases in value, unique

Not necessarily sequential (gaps exist)

Completely parallel

```
# 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)
```

More ID Tricks

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. Make sure that the IDs output from a monthly Spark task start at the highest value from the previous month.

Caching

Caching in Spark:

Stores DataFrames in memory or on disk

Improves speed on later transformations / actions

Reduces resource usage

Disadvantages of caching

Very large data sets may not t in memory

Local disk based caching may not be a performance improvement

Cached objects may not be available

Caching tips When developing Spark tasks:

Cache only if you need it

Try caching DataFrames at various points and determine if your performance improves

Cache in memory and fast SSD / NVMe storage

Cache to slow local disk if needed

Use intermediate files!

Stop caching objects when finished

Eviction Policy Least Recently Used (LRU)

Caching is a lazy operation. It requires an action to trigger it. eg.

spark.sql("select count(*) from text").show()

partitioned_df.count()

```
df.cache()    #df.persist() df.persist(storageLevel=pyspark.StorageLevel.MEMORY_
df.unpersist()

#Determining whether a dataframe is cached
df.is_cached

#storage level useDisk useMemory useOffHeap deserialized replication
df.storageLevel

#Caching a table
df.createOrReplaceTempView('df')
spark.catalog.cacheTable('df')
spark.catalog.isCached(tableName='df')
spark.catalog.dropTempView('table1')
# List the tables
print("Tables:\n", spark.catalog.listTables())

# Uncaching a table
spark.catalog.uncacheTable('df')
spark.catalog.clearCache()
```

Improve import performance

Spark clusters

Spark Clusters are made of two types of processes

Driver process

Worker processes

Import performance

Important parameters:

Number of objects (Files, Network locations, etc)

More objects better than larger ones

Can import via wildcard

```
airport_df = spark.read.csv('airports-*.txt.gz')
```

General size of objects

Spark performs better if objects are of similar size

It's safe to assume the more import objects available, the better the cluster can divvy up the job.

Schemas

A well-dened schema will drastically improve import performance

Avoids reading the data multiple times

Provides validation on import

How to split objects

Use OS utilities / scripts (split, cut, awk)

```
split -l 10000 -d largefile chunk-
```

> 每个文件100000行，字符，名字叫largefile，生成chunk0000开始

Use custom scripts

Write out to Parquet

```
df_csv = spark.read.csv('singlelargefile.csv')
```

```
df_csv.write.parquet('data.parquet')
df = spark.read.parquet('data.parquet')
```

Explaining the Spark execution plan

```
voter_df = df.select(df['VOTER NAME']).distinct()
voter_df.explain()
```

> == Physical Plan ==
> *(2) HashAggregate(keys=[VOTER NAME#15], functions=[]) +- Exchange hashpartitioning(VOTER NAME#15, 200)
> +- *(1) HashAggregate(keys=[VOTER NAME#15], functions=[])
> +- *(1) FileScan csv [VOTER NAME#15] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/DallasCouncilVotes.csv.gz],
> PartitionFilters: [], PushedFilters: [],
> ReadSchema: struct<VOTER NAME:string>

Shuffling

Shuffling refers to moving data around to various workers to complete a task

Hides complexity from the user

Can be slow to complete

Lowers overall throughput

Is often necessary, but try to minimize

How to limit shuffling

Limit use of `.repartition(num_partitions)`
Use `.coalesce(num_partitions)` instead

Use care when calling `.join()` Use `.broadcast()`

May not need to limit it

Broadcasting

Provides a copy of an object to each worker

Prevents undue / excess communication between nodes

Can drastically speed up `.join()` operations

A couple tips:

Broadcast the smaller DataFrame. The larger the DataFrame, the more time required to transfer to the worker nodes.

On small DataFrames, it may be better skip broadcasting and let Spark figure out any optimization on its own.

If you look at the query execution plan, a broadcastHashJoin indicates you've successfully configured broadcasting.

Use the `.broadcast(<DataFrame>)` method

```
from pyspark.sql.functions import broadcast
combined_df = df_1.join(broadcast(df_2))
```

Cluster Configurations

Configuration options

Spark contains many configuration settings

These can be modied to match needs

Reading configuration settings: `spark.conf.get(<configuration name>)`

Writing configuration settings `spark.conf.set(<configuration name>)`

Configure Spark to use 500 partitions
`spark.conf.set('spark.sql.shuffle.partitions', 500)`

```
# Name of the Spark application instance
app_name = spark.conf.get('spark.app.name')

# Driver TCP port
driver_tcp_port = spark.conf.get('spark.driver.port')

# Number of join partitions
num_partitions = spark.conf.get('spark.sql.shuffle.partitions')

# Store the number of partitions in variable
before = departures_df.rdd.getNumPartitions()

# Configure Spark to use 500 partitions
spark.conf.set('spark.sql.shuffle.partitions', 500)

# Recreate the DataFrame using the departures data file
departures_df = spark.read.csv('departures.txt.gz').distinct()
```

Cluster Types

Spark deployment options:

Single node (deploying all components on a single system, can be physical/VM/container)

Standalone (dedicated machines as the driver and workers)

Managed (cluster components are handled by a third party cluster manager)

YARN

Mesos

Kubernetes

Driver

Task assignment

Result consolidation

Shared data access

Tips:

Driver node should have double the memory of the worker

Fast local storage helpful

Worker

Runs actual tasks

Ideally has all code, data, and resources for a given task

Recommendations:

More worker nodes is often better than larger workers

Test to find the balance

Fast local storage extremely useful

Data Pipelines

Input(s)
CSV, JSON, web services, databases

Transformations
`withColumn()` ,
`.filter()` ,
`.drop()`

Output(s)
CSV, Parquet, database Validation

Analysis

Pipeline details

Not formally dened in Spark

Typically all normal Spark code required for task

```
schema = StructType([
 StructField('name', StringType(), False),
 StructField('age', StringType(), False)
 ])
 df = spark.read.format('csv').load('datafile').schema(schema) df = df.withColur
 ...
 df.write.parquet('outdata.parquet')
 df.write.json('outdata.json')
```

```
 # Import the data to a DataFrame
departures_df = spark.read.csv('2015-departures.csv.gz', header=True)

# Remove any duration of 0
departures_df = departures_df.filter(~departures_df[3].contains('0'))

# Add an ID column
departures_df = departures_df.withColumn('id', F.monotonically_increasing_id())

# Write the file out to JSON format
departures_df.write.json('output.json')
```

Data Handling

Parsing

Incorrect data
* Empty rows
* Commented lines
* Headers

Nested structures
* Multiple delimiters

Non-regular data
 * Differing numbers of columns per row

Removing blank lines, headers, and comments

Spark's CSV parser:

Automatically removes blank lines

Can remove comments using an optional argument

```
df1 = spark.read.csv('datafile.csv.gz', comment='#')
```

Handles header fields

Defined via argument

Ignored if a schema is defined

```
df1 = spark.read.csv('datafile.csv.gz', header='True')
```

Count the number of rows beginning with '#'

```
comment_count =
annotations_df.where(col('_c0').startswith('#')).count()
```

```
# Split _c0 on the tab character and store the list in a variable
tmp_fields = F.split(annotations_df['_c0'], '\t')

# Create the colcount column on the DataFrame
annotations_df = annotations_df.withColumn('colcount', F.size(tmp_fields))

# Remove any rows containing fewer than 5 fields
annotations_df_filtered = annotations_df.filter(~ (annotations_df["colcount"] <

# Count the number of rows
final_count = annotations_df_filtered.count()
print("Initial count: %d\nFinal count: %d" % (initial_count, final_count))
```

Automatic column creation

Spark will:

Automatically create columns in a DataFrame based on sep argument `df1 = spark.read.csv('datafile.csv.gz', sep=',')`

Defaults to using `,`

Can still successfully parse if sep is not in string

```
df1 = spark.read.csv('datafile.csv.gz', sep='*')
```

Stores data in column defaulting to `_c0`

Allows you to properly handle nested separators

```
# Split the content of _c0 on the tab character (aka, '\t')
split_cols = F.split(annotations_df['_c0'], '\t')

# Add the columns folder, filename, width, and height
split_df = annotations_df.withColumn('folder', split_cols.getItem(0))
split_df = split_df.withColumn('filename', split_cols.getItem(1))
split_df = split_df.withColumn('width', split_cols.getItem(2))
split_df = split_df.withColumn('height', split_cols.getItem(3))


def retriever(cols, colcount):
  # Return a list of dog data
  return cols[4:colcount]

# Define the method as a UDF
udfRetriever = F.udf(retriever, ArrayType(StringType()))

# Create a new column using your UDF
split_df = split_df.withColumn('dog_list', udfRetriever(split_df.split_cols, sp

# Remove the original column, split_cols, and the colcount
split_df = split_df.drop('_c0').drop('colcount').drop('split_cols')
```

Data Validation

Validation is:

Verifying that a dataset complies with the expected format

Number of rows / columns

Data types

Complex validation rules

Validating via joins

Compares data against known values

Easy to find data in a given set

Comparatively fast

```
parsed_df = spark.read.parquet('parsed_data.parquet')
company_df = spark.read.parquet('companies.parquet')
verified_df = parsed_df.join(company_df, parsed_df.company == company_df.company
```

```
# Rename the column in valid_folders_df
valid_folders_df = valid_folders_df.withColumnRenamed('_c0', 'folder')

# Count the number of rows in split_df
split_count = split_df.count()

# Join the DataFrames
joined_df = split_df.join(F.broadcast(valid_folders_df), "folder")

# Compare the number of rows remaining
joined_count = joined_df.count()
print("Before: %d\nAfter: %d" % (split_count, joined_count))
```

This *automatically* removes any rows with a company not in the valid_df !

Complex rule validation

Using Spark components to validate logic:

Calculations

Verifying against external source

Likely uses a UDF to modify / verify the DataFrame

```
# Determine the row counts for each DataFrame
split_count = split_df.count()
joined_count = joined_df.count()

# Create a DataFrame containing the invalid rows
invalid_df = split_df.join(F.broadcast(joined_df), 'folder', 'left_anti')

# Validate the count of the new DataFrame is as expected
invalid_count = invalid_df.count()
print(" split_df:\t%d\n joined_df:\t%d\n invalid_df: \t%d" % (split_count, joine

# Determine the number of distinct folder columns removed
invalid_folder_count = invalid_df.select('folder').distinct().count()
print("%d distinct invalid folders found" % invalid_folder_count)
```

```
# Select the dog details and show 10 untruncated rows
print(joined_df.select('dog_list').show(truncate=False))

# Define a schema type for the details in the dog list
DogType = StructType([
        StructField("breed", StringType(), False),
    StructField("start_x", IntegerType(), False),
    StructField("start_y", IntegerType(), False),
    StructField("end_x", IntegerType(), False),
    StructField("end_y", IntegerType(), False)
])


# Create a function to return the number and type of dogs as a tuple
def dogParse(doglist):
  dogs = []
  for dog in doglist:
    (breed, start_x, start_y, end_x, end_y) = dog.split(',')
    dogs.append((breed, int(start_x), int(start_y), int(end_x), int(end_y)))
  return dogs

# Create a UDF
udfDogParse = F.udf(dogParse, ArrayType(DogType))

# Use the UDF to list of dogs and drop the old column
joined_df = joined_df.withColumn('dogs', udfDogParse('dog_list')).drop('dog_lis

# Show the number of dogs in the first 10 rows
joined_df.select(F.size('dogs')).show(10)
```

```
# Define a UDF to determine the number of pixels per image
def dogPixelCount(doglist):
  totalpixels = 0
  for dog in doglist:
    totalpixels += (dog[3] - dog[1]) * (dog[4] - dog[2])
  return totalpixels

# Define a UDF for the pixel count
udfDogPixelCount = F.udf(dogPixelCount, IntegerType())
joined_df = joined_df.withColumn('dog_pixels', udfDogPixelCount('dogs'))

# Create a column representing the percentage of pixels
joined_df = joined_df.withColumn('dog_percent', (joined_df.dog_pixels / (joined_

# Show the first 10 annotations with more than 60% dog
joined_df.where('dog_percent > 60').show(10)
```