Day4

Data Cleaning

- Reformatting or replacing
-

Spark Schemas

filter garbage data during import

datatypes -

`inferSchema = True` is costly operation -> loads the data in-memory -> losses lazy evaluation
Best Practice: Design your own schemas

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

# Define custom schema
schema = StructType([
StructField('Student_Name', StringType(), True),
StructField('Student_Age', IntegerType(), True),
StructField('Student_Subject', StringType(), True),
StructField('Student_Class', IntegerType(), True),
StructField('Student_Fees', IntegerType(), True)
])
# Nullable = False/True

# Load data with custom schema
df_custom = spark.read.format("csv").schema(schema).option("header",
True).load("/path/to/student_data.csv")


df = spark.read.format('csv').load('python/test_support/sql/people.json',
schema=schema)
```

Variable Review

- Mutable
- Flexibility
- Potential for issues with concurrency

- Likely adds complexity

Immutability

new columns are added to new spark dataFrame

withColumn

column arithmetic

```
df.withColumn('age2', df.age + 2).collect()
```

lower functions

```
import pyspark.sql.functions as F

aa_dfw_df = aa_dfw_df.withColumn('airport', F.lower(aa_dfw_df['Destination
Airport']))
```

drop

drop column

```
df.drop('age').collect()
```

Lazy Processing

Optimizer
query plan

transformations
actions
allows efficient planning

can load file

parquet file format

https://ieeexplore.ieee.org/document/5767933
IEEE Xplore Full-Text PDF:
predicate pushdown -> columnar data file format
can be compressed -> smaller than text file format

columnar data format
supported in spark and other data processing framework
supports

```
df = spark.read.parquet("file_path")
```

```
spark.write.parquet("file_path")
```

Spark and CSV files

slow to parse
no predicate pushdown
any intermediate use requires redefining schema

load data

Filtering data

isNotNull()
isNull()

contains
`>`

```
import pyspark.sql.functions as F
```

column as transformation

```
aa_dfw_df = aa_dfw_df.withColumn('airport', F.lower(aa_dfw_df['Destination
Airport']))
```

create intermediary columns

```
# split
voter_df = voter_df.withColumn("splits", F.split(voter_df.VOTER_NAME, '\s+'))

# getItem
voter_df = voter_df.withColumn("last_name",
voter_df.splits.getItem(F.size('splits') - 1))
```

cast to other types

Conditional example

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

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

```
voter_df = voter_df.withColumn('random_val',
                                F.when(voter_df.TITLE == 'Councilmember', 1)
                                .when(voter_df.TITLE == 'Mayor', 2)
                                .when(voter_df.TITLE == 'Mayor Pro Tem', 3)
                                .when(voter_df.TITLE == 'Deputy Mayor Pro Tem',
4)
                               .otherwise(5))
```

User Defined Function

```
from pyspark.sql import SparkSession from pyspark.sql.functions import udf,
col from pyspark.sql.types import StringType

def simple_greeting(name):
    return f"Hello, {name}!"

# Registering without decorator
simple_greeting_udf = udf(simple_greeting, StringType())

df.withColumn("greeting", simple_greeting_udf(col("name"))).show()
```

```
print("Missing Values %: ", voter_df.filter(voter_df.splits.isNull()).count()/
voter_df.count() * 100)

voter_df = voter_df.dropna()
#voter_df = voter_df.fillna() # 5% or more missing values in df

def getFirstAndMiddle(names):
  # Return a space separated string of names
  return ' '.join(names[0:-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_df.splits))
```