**F1 Project**

Data can be accessed from:

<http://ergast.com/mrd/>

Data Overview

|  |  |
| --- | --- |
| **File** | **Type** |
| Circuits | CSV |
| Races | CSV |
| Constructors | Single Line JSON |
| Drivers | Single Line Nested JSON |
| Results | Single Line JSON |
| PitStops | Multi Line JSON |
| LapTimes | Split CSV Files |
| Qualifying | Split Multi Line JSON Files |

**Requirements for Ingestion**

* Ingest All 8 files into data lake
* Ingested data must have the schema applied
* Ingested data must have audit columns
* Ingested data must be stored in columnar format (i.e. Parquet)
* Must be able to analyse the ingested data via SQL
* Ingestion logic must be able to handle incremental load

A screenshot of a computer

Description automatically generatedThis is the ingestion path:

**Circuits.csv Ingestion**

Spark documentation:

<https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/io.html>

We want to load a CSV file and return the result as a DataFrame.

A screen shot of a computer code

Description automatically generatedHere is an example:

Spark.read.csv() is what we need.

**Create Mount Points**

A screen shot of a computer

Description automatically generatedA screen shot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generatedNotice that the header is ‘c0, c1, c2’. This means that the header has been interpreted incorrectly. To fix this we can refer to documentation:

The documentation specifies that there is an option that we can specify with regards to the header.

To fix our code:

A screenshot of a computer

Description automatically generatedWe specify that header=’true’, which means the first line in our table will become the column names.

**PrintSchema()**

A close-up of a computer code

Description automatically generated<https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.printSchema.html?highlight=printschema>

This can show us information about our schema.

A screenshot of a computer program

Description automatically generatedTo implement the method:

Notice that all of the column types have been identified as strings. We need to rectify this, because we know that we have some integers.

**Circuits\_df.describe().show()**

.describe().show() performs a summary of descriptive statistics on a DataFrame and displays the results.

* Count: the number of non-null entries
* Mean: the average value of the column (for numeric columns)
* Stddev: the standard deviation of the column (for numeric columns)
* Min: the minimum value in the column.
* Max: the maximum value in the column.

A screenshot of a computer

Description automatically generated**inferSchema()**

InferSchema is an option you can set when reading data to automatically detect the data types of each column in your dataset. This is particularly useful when reading data from formats that do not explicitly define the schema, such as CSV or JSON files.

Some of the benefits of ‘inferSchema’ include;

* **Automatic Type Detection**: You don’t need to manually specify the schema, which can save time and reduce errors, especially for large datasets with many columns.
* **Improved Data Handling**: by correctly identifying data types, Spark can apply the appropriate operations and optimisations on the data.
* A screenshot of a computer

  Description automatically generated**Ease of Use**: Makes it easier to work with semi-structured data or when the schema is not known beforehand.

Now we can see that the column types have been correctly updated.

A white text with black text

Description automatically generated

Explicitly defining the schema:

**StructType**

StructType is used to define the overall structure (schema) of the Data Frame.

StructField is used to define each column in the schema, it takes three arguments:

* Name: the name of the column
* datatype: The data type of the column
* nullable: a Boolean flag indicating whether the column can contain null values

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedThis allows us to define the schema explicitly. This can improve performance. Schema inference with ‘inferSchema’ can incur a performance penalties.

A screenshot of a computer program

Description automatically generatedA screenshot of a computer

Description automatically generated

Now it looks like our dataframe has been edited correctly.

**Select/Drop columns**

There are some columns we do not need in our table; for example ‘url’.

**DataFrame.select**

<https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.select.html>

Here is an example:



Notice we are able to select the schema that we want to use.

There are multiple ways we can achieve this:

1: circuits\_selected\_df = circuits\_df.select("circuitId", "circuitRef", "name", "location", "country", "lat", "lng", "alt")

A screenshot of a computer

Description automatically generated

2: circuits\_selected\_df = circuits\_df.select(circuits\_df.circuitId, circuits\_df.circuitRef, circuits\_df.name, circuits\_df.location, circuits\_df.country, circuits\_df.lat, circuits\_df.lng, circuits\_df.alt)

A screenshot of a computer

Description automatically generated

3: circuits\_selected\_df = circuits\_df.select(circuits\_df["circuitId"], circuits\_df["circuitRef"], circuits\_df["name"], circuits\_df["location"], circuits\_df["country"], circuits\_df["lat"], circuits\_df["lng"], circuits\_df["alt"])

A screenshot of a computer

Description automatically generated

4: circuits\_selected\_df = circuits\_df.select(col("circuitId"), col("circuitRef"), col("name"), col("location"), col("country"), col("lat"), col("lng"), col("alt"))

A screenshot of a computer

Description automatically generated

The final method requires ‘col’ to be imported.

It is important to be aware that there are a variety of ways to select/rename columns.

Note that col can give more flexibility:A screenshot of a computer

Description automatically generated

We can use ‘alias’ to rename columns.

**Rename Columns**

DataFrame.withColumnRenamed

[https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.withColumnRenamed.html?highlight=withcolumnrenamedA computer screen shot of a computer code

Description automatically generated](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.withColumnRenamed.html?highlight=withcolumnrenamed)

This allows us to rename columns.

A screenshot of a computer

Description automatically generatedNote that we only need to call on columns we want to rename:

**Adding Columns**

<https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.sql.DataFrame.withColumn.html>

A screenshot of a computer

Description automatically generateddataframe.withColumn() can be used to add columns.

Now we have added a timestamp!

**Dataframe Writer**

**A screenshot of a computer

Description automatically generated**

We have all of the data as we need it. Our next step is to write it to our DataLake. We need to write it as a parquet file.

<https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.sql.DataFrameWriter.parquet.html>

Here is an example:



A screenshot of a computer code

Description automatically generated

Note that we can’t keep running this bit of code. If the filepath already exists, it won’t work! To get round this we can use ‘mode’ and overwrite;

file\_path = "/mnt/formula1dl/processed/circuits"

circuits\_final\_df.write.mode("overwrite").parquet(file\_path)

Now we can keep running the code and it will work.

**Data Ingestion – Races**

First establish credentials to allow mount to blob:

# Access variables stored in key vault:

# Access application-client-id token secret:

client\_id = dbutils.secrets.get(

scope="f1-scope", key="application-client-id-demo")

tenant\_id = dbutils.secrets.get(

scope="f1-scope", key="directory-tenant-id-demo")

client\_secret = dbutils.secrets.get(

scope="f1-scope", key="application-client-secret")

storage\_account = "f1dl9072024"

container\_name = 'raw'

scope\_name = 'f1-scope'

csv\_location = "dbfs:/mnt/f1dl9072024/raw/races.csv"

Next configure Spark:

configs = {"fs.azure.account.auth.type": "OAuth",

"fs.azure.account.oauth.provider.type": "org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider",

"fs.azure.account.oauth2.client.id": client\_id,

"fs.azure.account.oauth2.client.secret": client\_secret,

"fs.azure.account.oauth2.client.endpoint": f"https://login.microsoftonline.com/{tenant\_id}/oauth2/token"}

Then mount:

def mount\_adls(storage\_account\_name, container\_name):

# Access secrets from Key Vault:

client\_id = dbutils.secrets.get(

scope="f1-scope", key="application-client-id-demo")

tenant\_id = dbutils.secrets.get(

scope="f1-scope", key="directory-tenant-id-demo")

client\_secret = dbutils.secrets.get(

scope="f1-scope", key="application-client-secret")

# Set spark configurations:

configs = {"fs.azure.account.auth.type": "OAuth",

"fs.azure.account.oauth.provider.type": "org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider",

"fs.azure.account.oauth2.client.id": client\_id,

"fs.azure.account.oauth2.client.secret": client\_secret,

"fs.azure.account.oauth2.client.endpoint": f"https://login.microsoftonline.com/{tenant\_id}/oauth2/token"}

# Check to see if mount exists. Unmount if exists:

if any(mount.mountPoint == f"/mnt/{storage\_account\_name}/{container\_name}" for mount in dbutils.fs.mounts()):

dbutils.fs.unmount(f"/mnt/{storage\_account\_name}/{container\_name}")

# Mount the storage account container:

dbutils.fs.mount(

source=f"abfss://{container\_name}@{storage\_account\_name}.dfs.core.windows.net/",

mount\_point=f"/mnt/{storage\_account\_name}/{container\_name}",

extra\_configs=configs)

mount\_adls(storage\_account, container\_name)

Create a dataframe:

races\_df = spark.read.csv("dbfs:/mnt/f1dl9072024/raw/races.csv", header='true')

Specify the schema:

from pyspark.sql.types import StructType, StructField, IntegerType, StringType, DoubleType, TimestampType, DateType

races\_schema = StructType(fields = [

StructField("raceId", IntegerType(), False),

StructField("year", IntegerType(), True),

StructField("round", IntegerType(), True),

StructField("circuitId", IntegerType(), True),

StructField("name", StringType(), True),

StructField("date", DateType(), True),

StructField("time", StringType(), True),

StructField("url", StringType(), True)

])

Define races dataframe:

races\_df = spark.read \

.option("header", True) \

.schema(races\_schema) \

.csv(csv\_location)

Update ingestion date timestamp. Notice that we are combining two columns into one with the concat method.

from pyspark.sql.functions import current\_timestamp, to\_timestamp, concat, col, lit

races\_with\_timestamp\_df = races\_df \

.withColumn("ingestion\_date", current\_timestamp()) \

.withColumn("race\_timestamp", to\_timestamp(concat(col('date'), lit(' '), col('time')), 'yyyy-MM-dd HH:mm:ss'))

display(races\_with\_timestamp\_df)

Select the columns we need:

from pyspark.sql.functions import col

races\_selected\_df = races\_with\_timestamp\_df.select(

col('raceId').alias("race\_id"),

col('year').alias("race\_year"),

col('round'),

col('circuitId').alias("circuit\_id"),

col('name'),

col('ingestion\_date'),

col('race\_timestamp')

)

Write to Data Lake as parquet:

file\_path = f"/mnt/{storage\_account}/processed/races"

races\_selected\_df.write.mode("overwrite").parquet(file\_path)

**Partition By**

Currently we have written one parquet file, which contains all the races. However, what if we wanted to partition by, say; race\_year? We could get all the data for the year of 2009 for example.

file\_path = f"/mnt/{storage\_account}/processed/races"

races\_selected\_df.write.mode("overwrite").partitionBy('race\_year').parquet(file\_path)

If we look in the storage folder, we can see that everything has been partitioned by the race year, and stored in individual folders:

A screenshot of a computer

Description automatically generated

**Data Ingestion Constructors**

Now we need to ingest a json file.

A screenshot of a computer

Description automatically generated

In our CSV file, in the first row we have the header. This can tell us important information about our column names. JSON’s on the other hand, do not have headers.

{*"constructorId"*:1,*"constructorRef"*:"mclaren",*"name"*:"McLaren",*"nationality"*:"British",*"url"*:"http://en.wikipedia.org/wiki/McLaren"}

{*"constructorId"*:2,*"constructorRef"*:"bmw\_sauber",*"name"*:"BMW Sauber",*"nationality"*:"German",*"url"*:"http://en.wikipedia.org/wiki/BMW\_Sauber"}

Instead, jsons have key/value pairs. Because of this, we need to infer the input schema automatically.

<https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameReader.json.html#pyspark.sql.DataFrameReader.json>

First define schema:

constructors\_schema = "constructorId INT, constructorRef STRING, name STRING, nationality STRING, url STRING"

Next read the file with the schema specified earlier;

constructor\_df = spark.read \

.schema(constructors\_schema) \

.json(f"/mnt/{storage\_account}/raw/constructors.json")

Now we need to drop unwanted columns from the dataframe:

constructor\_dropped\_df = constructor\_df.drop('url')

Now we need to rename a column and add a column; ingestion\_Date:

from pyspark.sql.functions import current\_timestamp

constructor\_final\_df = constructor\_dropped\_df.withColumnRenamed("constructorId", "constructor\_id") \

.withColumnRenamed("constructorRef", "constructor\_ref") \

.withColumn("ingestion\_date", current\_timestamp())

Before finally writing to parquet:

constructor\_final\_df.write.mode("overwrite").parquet(f"/mnt/{storage\_account}/processed/constructors")

**Data Ingestion Drivers**

A screenshot of a computer

Description automatically generatedNow we need to ingest our drivers json file.

Notice that in this file:

{*"driverId"*:1,*"driverRef"*:"hamilton",*"number"*:44,*"code"*:"HAM",*"name"*:{*"forename"*:"Lewis",*"surname"*:"Hamilton"},*"dob"*:"1985-01-07",*"nationality"*:"British",*"url"*:"http://en.wikipedia.org/wiki/Lewis\_Hamilton"}

There is a nested json object. This means we need to approach this ingestion task, slightly differently.

**Data Ingestion – Results**

A screenshot of a computer

Description automatically generated

1. Define the schema:

from pyspark.sql.types import StructType, StructField, IntegerType, StringType, DoubleType, TimestampType, DateType, FloatType

races\_schema = StructType(fields=[

StructField("resultId", IntegerType(), False),

StructField("raceId", IntegerType(), True),

StructField("driverId", IntegerType(), True),

StructField("constructorId", IntegerType(), True),

StructField("number", IntegerType(), True),

StructField("grid", IntegerType(), True),

StructField("position", IntegerType(), True),

StructField("positionText", StringType(), True),

StructField("positionOrder", IntegerType(), True),

StructField("points", FloatType(), True),

StructField("laps", IntegerType(), True),

StructField("time", StringType(), True),

StructField("milliseconds", IntegerType(), True),

StructField("fastestLap", IntegerType(), True),

StructField("rank", IntegerType(), True),

StructField("fastestLapTime", StringType(), True),

StructField("fastestLapSpeed", StringType(), True),

StructField("statusId", IntegerType(), True)

])

2. Read the JSON file:

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

results\_df = spark.read.json(f"/mnt/{storage\_account}/raw/results.json", schema=races\_schema)

3. Drop the Unwanted Column:

df\_drop = results\_df.drop("statusId")

4. Rename Columns:

from pyspark.sql.functions import col

df\_rename = df\_drop.select(

col("resultId").alias("result\_id"),

col("raceId").alias("race\_id"),

col("driverId").alias("driver\_id"),

col("constructorId").alias("constructor\_id"),

col("number"),

col("grid"),

col("position"),

col("positionText").alias("position\_text"),

col("positionOrder").alias("position\_order"),

col("points"),

col("laps"),

col("time"),

col("milliseconds"),

col("fastestLap").alias("fastest\_lap"),

col("rank"),

col("fastestLapTime").alias("fastest\_lap\_time"),

col("fastestLapSpeed").alias("fastest\_lap\_speed")

)

5. Create new Column:

from pyspark.sql.functions import current\_timestamp, to\_timestamp, concat, col, lit

df\_with\_column = df\_rename.withColumn("ingestion\_date", current\_timestamp())

6. Write to Parquet:

df\_with\_column.write.mode("overwrite").parquet(f"/mnt/{storage\_account}/processed/results")

7. Check parquet written:

display(spark.read.parquet(f"/mnt/{storage\_account}/processed/results"))

**Data Ingestion – Pitstops**

A screenshot of a computer

Description automatically generated

Previously we were ingesting 1 json file. This time, we are ingesting an array with multiple json files:

[

{

*"raceId"*:841,

*"driverId"*:153,

*"stop"*:1,

*"lap"*:1,

*"time"*:"17:05:23",

*"duration"*:26.898,

*"milliseconds"*:26898

},

First define schema:

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

pit\_schema = StructType(fields=[

StructField("raceId", IntegerType(), False),

StructField("driverId", IntegerType(), True),

StructField("stop", IntegerType(), True),

StructField("time", StringType(), True),

StructField("duration", StringType(), True),

StructField("milliseconds", IntegerType(), True)

])

Next read multi-line JSON, note that there is an option parameter.

pit\_stops\_df = spark.read \

.schema(pit\_schema) \

.option("multiLine", True) \

.json(json\_location)

Rename and add ingestion date:

from pyspark.sql.functions import current\_timestamp

final\_df = pit\_stops\_df \

.withColumnRenamed("driverId", "driver\_id") \

.withColumnRenamed("raceId", "race\_id") \

.withColumn("ingestion\_date", current\_timestamp())

Write parquet file:

final\_df.write.mode("overwrite").parquet(f"/mnt/{storage\_account}/processed/pit\_stops")

Check parquet file written:

display(spark.read.parquet(f"/mnt/{storage\_account}/processed/pit\_stops"))

**Ingest multiple CSV files**

Compared to our other files, ‘lap\_times’ is a folder.

**A blue folder with a black text

Description automatically generated**

Composed of 5 further csv files:

**A close-up of a document

Description automatically generated**

In order to read from two files there are two methods;

lap\_times\_df = spark.read \

.schema(lap\_times\_schema) \

.csv(f"/mnt/{storage\_account}/{container\_name}/lap\_times/lap\_times\_split\_\*.csv")

**Or:**

lap\_times\_df = spark.read \

.schema(lap\_times\_schema) \

.csv(f"/mnt/{storage\_account}/lap\_times/lap\_times")

The first method is useful if you only want the CSV files. Perhaps the folder contains JSON files too. The second method is useful if you know that the folder only contains CSV files.

**Multiple JSON files**

qualifying\_df = spark.read.schema(qualifying\_schema).option('multiLine', True).json(f"/mnt/{storage\_account}/{container\_name}/qualifying")

It is also possible to read multiple JSON files:

**Importing from other notebooks**

So far we have been re-using functions by writing them multiple times. Databricks allows us to import functions from other files.

from pyspark.sql.functions import current\_timestamp

def add\_ingestion\_date(input\_df):

# Args: dataframe.

# Output: dataframe with additional column; 'ingestion\_date'.

# Ingestion date is set to current time.

output\_df = input\_df.withColumn("ingestion\_date", current\_timestamp())

return output\_df

Here is an example of a function that accepts a dataframe and adds an additional column called ‘ingestion\_date’. Ingestion\_date will be set to the current time.

In our ingestion notebook we have this:

from pyspark.sql.functions import current\_timestamp

circuits\_final\_df = circuits\_renamed\_df.withColumn("ingestion\_date", current\_timestamp())

However if we import:

%run "../includes/common\_functions.py"

**Note the RUN COMMAND**

Must be in a single cell! No other code or comments can be present.

We can now call the function:

circuits\_final\_df = add\_ingestion\_date(circuits\_renamed\_df)

**Parsing Parameters**

Sometimes we may need to add fields ‘on the fly’.

dbutils.widgets.text("p\_data\_source", "")

v\_data\_source = dbutils.widgets.get("p\_data\_source")

This source creates a widget.

When run it will create this widget:A screenshot of a computer

Description automatically generated

v\_data\_source = dbutils.widgets.get("p\_data\_source")

This can then be saved to the variable; ‘v\_data\_source”.

Which can then be used with our dataframe functions to add data to our dataframe:

from pyspark.sql.functions import current\_timestamp, lit

final\_df = qualifying\_df.withColumnRenamed("qualifyId", "qualifying\_id").withColumnRenamed("raceId", "race\_id").withColumnRenamed("driverId", "driver\_id").withColumnRenamed("constructorId", "constructor\_id").withColumn("ingestion\_date", current\_timestamp()).withColumn("data\_source", lit(v\_data\_source))

A screenshot of a computer

Description automatically generatedHere is the dataframe with the added data\_source table:

**Workflows**

A screenshot of a computer

Description automatically generated

Workflows can be used to automate running notebooks. First we can use dbutils.notebook.help.

Notice that there is a run method that we can call:

dbutils.notebook.run("1.ingest\_circuits\_csv", 0, {"p\_data\_source": "Ergast API"})

This code will create a new run. The parameters are; the path to the file, timeout in seconds, and any parameter values we want. Now we can see the new table:

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

You can also use a for loop!

**Databricks Jobs**

A screenshot of a computer

Description automatically generated

Here is the JSON configuration of the job:

{

*"job\_id"*: 166721424043089,

*"creator\_user\_name"*: "xrs@icloud.com",

*"run\_as\_user\_name"*: "xrs@icloud.com",

*"run\_as\_owner"*: true,

*"settings"*: {

*"name"*: "F1-Ingestion",

*"email\_notifications"*: {

*"no\_alert\_for\_skipped\_runs"*: false

},

*"webhook\_notifications"*: {},

*"timeout\_seconds"*: 0,

*"max\_concurrent\_runs"*: 1,

*"tasks"*: [

{

*"task\_key"*: "Run\_all\_file\_ingestion",

*"run\_if"*: "ALL\_SUCCESS",

*"notebook\_task"*: {

*"notebook\_path"*: "/Workspace/Users/xrs@icloud.com/Formula1/ingestion/0.ingest\_all\_files",

*"base\_parameters"*: {

*"p\_data\_source"*: "E API"

},

*"source"*: "WORKSPACE"

},

*"existing\_cluster\_id"*: "0722-063548-8ifsg8p1",

*"timeout\_seconds"*: 0,

*"email\_notifications"*: {},

*"notification\_settings"*: {

*"no\_alert\_for\_skipped\_runs"*: false,

*"no\_alert\_for\_canceled\_runs"*: false,

*"alert\_on\_last\_attempt"*: false

},

*"webhook\_notifications"*: {}

}

],

*"format"*: "MULTI\_TASK",

*"queue"*: {

*"enabled"*: true

}

},

*"created\_time"*: 1721657490934

}

This will run the 0.ingest\_all\_files file to ingest all data.

**Filter Transformation**

A screenshot of a computer

Description automatically generated

Reading from our races parquet, we can see that there are a lot of results! Notice that the race\_id is in descending order.

Perhaps we only want the data from specific years.

# 1. SQL Way:

races\_filterd\_df = races\_df.filter("race\_year = 2019")

# 2. Pythonic Way:

races\_filtered\_df = races\_df.filter(races\_df["race\_year"] == 2019)

Here there are two methods.

So far we have only specified one condition. Multiple conditions might look like this:

# 1. SQL Way:

races\_filterd\_df = races\_df.filter("race\_year = 2019 and round <= 5")

# 2. Pythonic Way:

races\_filterd\_df = races\_df.filter((races\_df["race\_year"] == 2019) & (races\_df["round"] <= 5))

Returning the first 5 rounds, where the race year is equal to 2019.

**Inner Join Transformation**

race\_circuits\_df = circuits\_df.join(races\_df\_filtered, circuits\_df.circuit\_id == races\_df.circuit\_id, "inner")

display(race\_circuits\_df)

This code will perform an inner join, which matches where all values are the same. It does not produce null values. A screenshot of a computer

Description automatically generated

This can also be combined with a select statement, to select the data you need:

# Python syntax:

race\_circuits\_df = circuits\_df.join(races\_df\_filtered, circuits\_df.circuit\_id == races\_df.circuit\_id, "inner") \

.select(circuits\_df.name, circuits\_df.location, circuits\_df.country, races\_df.name, races\_df.round)

Note if name appears twice:

races\_df = spark.read.parquet(f"{processed\_folder\_path}/races") \

.withColumnRenamed("name", "race\_name")

circuits\_df = spark.read.parquet(f"{processed\_folder\_path}/circuits") \

.withColumnRenamed("name", "circuit\_name")

Rename columns, as above. Then edit select statement:

# Python syntax:

race\_circuits\_df = circuits\_df.join(races\_df\_filtered, circuits\_df.circuit\_id == races\_df.circuit\_id, "inner") \

.select(circuits\_df.circuit\_name, circuits\_df.location, circuits\_df.country, races\_df\_filtered.race\_name , races\_df\_filtered.round)

**Join Race Results**

We need to read from three tables.

1. Drivers – to get driver information.
2. Constructors – to get nationality.
3. Results – to get race results.
4. Races – to get race information
5. Circuits – to get circuit information

So there are multiple tables that we need to read from.

Here is an example of the table that we want to create:

|  |  |
| --- | --- |
| **Column Name** | **Source** |
| Race\_year | Races |
| Race\_name | Races |
| Race\_date | Races |
| Circuit\_location | circuits |
| Driver\_name | Drivers |
| Driver\_number | Drivers |
| Driver\_nationality | Drivers |
| Team | Constructors |
| Grid | Results |
| Fastest\_lap | Results |
| Race\_time | Results |
| Points | Results |
| Created\_date | Current\_timestamp |

Steps:

1. Create presentation mount point.
2. Create notebook
3. Write newly created dataframe into new mount point

Establish mount point:

Remember that the mountpoint has been stored in configuration as ‘processed\_folder\_path’.

%run "../includes/configuration"

This will allow us access to it.

Next create separate dataframes:

races\_df = spark.read.parquet(f"{processed\_folder\_path}/races").withColumnRenamed("name", "race\_name").withColumnRenamed("race\_timestamp", "race\_date")

drivers\_df = spark.read.parquet(f"{processed\_folder\_path}/drivers").withColumnRenamed("name", "driver\_name").withColumnRenamed("number", "driver\_number").withColumnRenamed("nationality", "driver\_nationality")

constructors\_df = spark.read.parquet(f"{processed\_folder\_path}/constructors").withColumnRenamed("name", "team")

circuits\_df = spark.read.parquet(f"{processed\_folder\_path}/circuits").withColumnRenamed("location", "circuit\_location")

results\_df = spark.read.parquet(f"{processed\_folder\_path}/results").withColumnRenamed("time", "race\_time")

Now join circuits to races:

race\_circuits\_df = races\_df.join(circuits\_df, races\_df.circuit\_id == circuits\_df.circuit\_id, "inner") \

.select(races\_df.race\_id, races\_df.race\_year, races\_df.race\_name, races\_df.race\_date, circuits\_df.circuit\_location)

Finally perform multiple joins:

race\_results\_df = results\_df.join(race\_circuits\_df, results\_df.race\_id == race\_circuits\_df.race\_id) \

.join(drivers\_df, results\_df.driver\_id == drivers\_df.driver\_id) \

.join(constructors\_df, results\_df.constructor\_id == constructors\_df.constructor\_id)

There might be repeated results, so remember a select statement!

from pyspark.sql.functions import current\_timestamp

final\_df = race\_results\_df.select("race\_year", "race\_name", "race\_date", "circuit\_location", "driver\_name", "driver\_number", "driver\_nationality", "team", "grid", "fastest\_lap", "race\_time", "points").withColumn("created\_date", current\_timestamp())

Final code:

from pyspark.sql.functions import current\_timestamp

final\_df = race\_results\_df.select("race\_year", "race\_name", "race\_date", "circuit\_location", "driver\_name", "driver\_number", "driver\_nationality", "team", "grid", "fastest\_lap", "race\_time", "points").withColumn("created\_date", current\_timestamp())

Note the order by used to sort results by points.

A screenshot of a computer

Description automatically generated

NOTES MISSING

**Hive Metastore**

If an existing file is present because of :

final\_df.write.mode("overwrite").parquet(f"/mnt/{storage\_account}/processed/qualifying")

Then you can use dbutils.rm to remove the file!

end\_path = 'qualifying'

try:

final\_df.write.mode("overwrite").format("parquet").saveAsTable(f"f1\_processed.{end\_path}")

print(f"{end\_path.capitalize()} table successfully created.")

except Exception as e:

print(f"Exception occurred: {e}")

try:

path = f"{processed\_folder\_path}/{end\_path}"

if dbutils.fs.ls(path):

dbutils.fs.rm(path, True)

final\_df.write.mode("overwrite").format("parquet").saveAsTable(f"f1\_processed.{end\_path}")

print(f"{end\_path.capitalize()} table successfully created.")

except Exception as e:

print(f"Exception occured: {e}")

Now the table can be created successfully.

**Incremental Load**

Up until now, we have ingested a small amount of data. Incremental load is a data integration technique used in data warehousing with ETL processes. It involves updating a target data store with only the changes (inserts, updates, and deletes) that have occurred since the last load. This is in contrast to a full load, where the entire dataset is loaded into the target system regardless of any changes. Incremental loading is more efficient and less time-consuming as it minimises the amount of data processed and transferred.