

Relatório: Pipeline de Dados na Nuvem para Análise Histórica da Fórmula 1 (1950–2024)

Um estudo sobre a construção de um pipeline de dados utilizando tecnologias em nuvem para análise de desempenho de pilotos e corridas na Fórmula 1

Relatório: Pipeline de Dados na Nuvem para Análise Histórica da Fórmula 1 (1950–2024)	1
1. Introdução	3
1.1 Contexto	3
1.2 Objetivo	4
2. Principais Etapas do Pipeline	5
2.1 Busca e Coleta de Dados	5
2.2 Modelagem dos Dados	6
2.2.1 - Catálogo de Dados	6
2.3 Carga - Pipeline de ETL dos dados	14
2.3.1 - Camada Bronze	14
2.3.2 - Camada Silver	15
2.3.3 - Camada Gold	17
2.4 Análise dos Dados	19
2.4.1 Análise de Qualidade dos Dados	19
2.4.2 Resposta às Perguntas do Objetivo	21
3. Discussão dos Resultados e Autoavaliação	33
3.1 Principais Descobertas	33
3.2 Limitações e Desafios	33
3.1 Atingimento dos Objetivos	33
4. Trabalhos Futuros	34
5. Conclusão	34

1. Introdução

1.1 Contexto

A Fórmula 1 (F1) é a categoria mais avançada do automobilismo mundial, regulamentada pela Federação Internacional do Automóvel (FIA) e organizada pelo Formula One Group. Desde sua primeira temporada em 1950, o Campeonato Mundial de Fórmula 1 da FIA consolidou-se como uma das principais competições de esporte a motor, reunindo equipes, pilotos e tecnologias de ponta em corridas realizadas em circuitos profissionais e até mesmo em vias públicas temporárias. O termo "fórmula" refere-se ao conjunto de regras técnicas e esportivas que todos os carros devem seguir, garantindo equilíbrio competitivo e inovação constante.

Com mais de **sete décadas de história**, a Fórmula 1 gera um **enorme volume de dados**, desde resultados de corridas e desempenho de pilotos até estatísticas de construtores, tempos de volta, pit stops e evolução de circuitos. Esses dados representam uma fonte valiosa para análises que podem revelar **padrões de desempenho, tendências históricas e insights estratégicos** sobre o esporte.

Este trabalho visa construir um pipeline de dados na nuvem para coletar, modelar, armazenar e responder às principais perguntas de negócio que foram levantadas tentando recriar uma consultoria de analistas que fornecerá esses dados à rede de TV que irá transmitir a próxima temporada.

1.2 Objetivo

O objetivo principal deste MVP é criar um pipeline de dados que permita responder às perguntas:

- Análise da temporada passada: Qual piloto fez a volta mais rápida de cada Grand Prix em 2024? E quem cada corrida? Quem venceu mais corridas?
 - Quem são os pilotos mais consistentes? (Pilotos com maior porcentagem de chegadas no pódio quando começaram no top 5 do grid)
 - Top 10 países com mais vitórias"? (Evolução das nacionalidades vencedoras ao longo do tempo)
 - Qual nacionalidade produziu os pilotos mais rápidos?
 - Quais são os maiores vencedores da história?
 - Quem são os maiores "começam mal, terminam bem"? (Pilotos com maior diferença média entre posição de largada e chegada)
 - Qual piloto teve a carreira mais longa?
 - Last Heros (Maiores recuperações em uma única corrida - do último grid ao pódio)
 - Tributo Ayrton Senna, todas as provas que ele ganhou
-

2. Principais Etapas do Pipeline

2.1 Busca e Coleta de Dados

- **Fonte de dados:** Dataset histórico da Fórmula 1 (1950–2024).
Link Kaggle:
<https://www.kaggle.com/datasets/rohanrao/formula-1-world-championship-1950-2020>
- **Método de coleta:** web scraping de repositórios públicos do Kaggle.
- **Armazenamento na nuvem:** Utilização do Databricks Community Edition

```
▶ 09:35 AM (3s) 1

!pip install summarytools
from summarytools import dfSummary
from IPython.display import display, HTML
import pandas as pd
import io
import requests

▶ 09:36 AM (1s) 2

url_races = "https://raw.githubusercontent.com/henrique-fes/mvp_data_eng/refs/heads/main/races.csv"
url_drivers = "https://raw.githubusercontent.com/henrique-fes/mvp_data_eng/refs/heads/main/drivers.csv"
url_results = "https://raw.githubusercontent.com/henrique-fes/mvp_data_eng/refs/heads/main/results.csv"

results = pd.read_csv(url_results,delimiter=',')
drivers = pd.read_csv(url_drivers,delimiter=',')
races = pd.read_csv(url_races,delimiter=',')
```

2.2 Modelagem dos Dados

Para estruturar o conjunto de dados históricos da Fórmula 1, foi utilizado o **modelo em Esquema Estrela**, amplamente utilizado em Data Warehouses por sua eficiência em consultas analíticas. A modelagem consiste em:

- **Tabelas de Dimensão** (entidades descritivas):
 - **drivers**: Contém informações sobre os pilotos:
 - **racers**: Armazena dados das corridas
- **Tabela Fato** (eventos mensuráveis):
 - **results**: Registra resultados de corridas

Vantagens do Esquema Estrela

1. **Simplicidade e Performance:**
 - Consultas analíticas são mais rápidas, pois evitam junções complexas (um join central entre a tabela fato e as dimensões).
 - Ideal para perguntas como *"Quantas vitórias o piloto X teve no circuito Y?"*.
2. **Flexibilidade para Análises:**
 - Permite "fatiar" os dados por diferentes dimensões (ex.: análise por década, país ou equipe).
 - Facilita a criação de dashboards e métricas agregadas (ex.: pontos médios por piloto).

2.2.1 - Catálogo de Dados

Um catálogo de dados é essencial para a governança e gestão eficiente de dados em qualquer projeto, pois documenta metadados, estrutura e características dos conjuntos de dados disponíveis. Ele permite que analistas e cientistas de dados compreendam rapidamente a estrutura, os tipos, se há valores faltantes ou outliers e estatísticas descritivas básicas. E pensando em uma solução foi desenvolvido em python a função "sumario_dados()" que organiza:

- **Metadados básicos**: shape do dataframe, número de colunas e linhas
- **Tipos de dados e completeness**: através do `df.info()`
- **Resumo estatístico**: com o `dfSummary()`

E foi feita para todas as tabelas: "drivers", "racers" e "results"

```
Just now (2s) 3 Python
def sumario_dados(df):
    display(HTML("<h2 style='color: #007bff;'>Dataset Info</h2>"))
    display(HTML(f"<h4 style='color: #6c757d;'>Shape: {df.shape}</h4>"))
    display(HTML(f"<h4 style='color: #6c757d;'>Numero de Colunas: {df.shape[1]}</h4>"))
    display(HTML(f"<h4 style='color: #6c757d;'>Numero de Linhas: {df.shape[0]}</h4>"))
    display(HTML("<hr>"))
    display(HTML("<h2 style='color: #28a745;'>Dataset Info </h2>"))
    display(HTML("<h4 style='color: #6c757d;'>Column types and missing data</h4>"))
    display(df.info())
    display(HTML("<hr>"))
    display(HTML("<h2 style='color: #17a2b8;'>Sumario de dados</h2>"))
    return dfSummary(df)

sumario_dados(results)
```

Dataset Info

Shape: (26759, 18)

Numero de Colunas: 18







Numero de Linhas: 26759

Dataset Info

Column types and missing data

0	resultId	26759	non-null	int64
1	raceId	26759	non-null	int64
2	driverId	26759	non-null	int64
3	constructorId	26759	non-null	int64
4	number	26759	non-null	object
5	grid	26759	non-null	int64
6	position	26759	non-null	object
7	positionText	26759	non-null	object
8	positionOrder	26759	non-null	int64
9	points	26759	non-null	float64
10	laps	26759	non-null	int64
11	time	26759	non-null	object
12	milliseconds	26759	non-null	object
13	fastestLap	26759	non-null	object
14	rank	26759	non-null	object
15	fastestLapTime	26759	non-null	object
16	fastestLapSpeed	26759	non-null	object
17	statusId	26759	non-null	int64

Data Frame Summary						
df						
Dimensions: 26,759 x 18						
Duplicates: 0						
No	Variable	Stats / Values	Freqs / (% of Valid)	Graph	Missing	
1	resultId [int64]	Mean (sd) : 13381.0 (7726.1) min < med < max: 1.0 < 13380.0 < 26764.0 IQR (CV) : 13379.0 (1.7)	26,759 distinct values		0 (0.0%)	
2	raceId [int64]	Mean (sd) : 551.7 (313.3) min < med < max: 1.0 < 531.0 < 1144.0 IQR (CV) : 511.0 (1.8)	1,125 distinct values		0 (0.0%)	
3	driverId [int64]	Mean (sd) : 278.7 (282.7) min < med < max: 1.0 < 172.0 < 862.0 IQR (CV) : 342.5 (1.0)	861 distinct values		0 (0.0%)	
4	constructorId [int64]	Mean (sd) : 50.2 (61.6) min < med < max: 1.0 < 25.0 < 215.0 IQR (CV) : 57.0 (0.8)	211 distinct values		0 (0.0%)	
5	number [object]	1. 4 2. 16 3. 11 4. 3 5. 6 6. 8 7. 14 8. 10 9. 20 10. 2 11. other	1,019 (3.8%) 1,005 (3.8%) 1,001 (3.7%) 994 (3.7%) 994 (3.7%) 993 (3.7%) 982 (3.7%) 976 (3.6%) 972 (3.6%) 959 (3.6%) 16,864 (63.0%)		0 (0.0%)	
6	grid [int64]	Mean (sd) : 11.1 (7.2) min < med < max: 0.0 < 11.0 < 34.0 IQR (CV) : 12.0 (1.5)	35 distinct values		0 (0.0%)	
7	position [object]	1. W 2. 3 3. 4 4. 2 5. 5 6. 1 7. 6 8. 7 9. 8 10. 9 11. other	10,953 (40.9%) 1,135 (4.2%) 1,135 (4.2%) 1,135 (4.2%) 1,133 (4.2%) 1,131 (4.2%) 1,128 (4.2%) 1,124 (4.2%) 1,104 (4.1%) 1,076 (4.0%) 1,038 (3.9%) 5,802 (21.7%)		0 (0.0%)	
8	positionText [object]	1. R 2. F 3. 3 4. 4 5. 2 6. 5 7. 1 8. 6 9. 7 10. 8 11. other	8,897 (33.2%) 1,368 (5.1%) 1,135 (4.2%) 1,135 (4.2%) 1,133 (4.2%) 1,131 (4.2%) 1,128 (4.2%) 1,124 (4.2%) 1,104 (4.1%) 1,076 (4.0%) 7,528 (28.1%)		0 (0.0%)	
9	positionOrder [int64]	Mean (sd) : 12.8 (7.7) min < med < max: 1.0 < 12.0 < 39.0 IQR (CV) : 12.0 (1.7)	39 distinct values		0 (0.0%)	
10	points [float64]	Mean (sd) : 2.0 (4.4) min < med < max: 0.0 < 0.0 < 50.0 IQR (CV) : 2.0 (0.5)	39 distinct values		0 (0.0%)	
11	laps [int64]	Mean (sd) : 46.3 (29.5) min < med < max: 0.0 < 53.0 < 200.0 IQR (CV) : 43.0 (1.6)	172 distinct values		0 (0.0%)	
12	time [object]	1. W 2. +8:22.19 3. +46.2 4. +5.7 5. +0.7 6. +1:29.6 7. +24.2 8. +12.8 9. +6.1 10. +58.2 11. other	19,079 (71.3%) 5 (0.0%) 4 (0.0%) 4 (0.0%) 4 (0.0%) 4 (0.0%) 3 (0.0%) 3 (0.0%) 3 (0.0%) 3 (0.0%) 7,647 (28.6%)		0 (0.0%)	
13	milliseconds [object]	1. W 2. 14259460 3. 10928200 4. 5350182 5. 5593660 6. 8867400 7. 4925000 8. 5808819 9. 14507710 10. 14260930 11. other	19,079 (71.3%) 5 (0.0%) 3 (0.0%) 2 (0.0%) 2 (0.0%) 2 (0.0%) 2 (0.0%) 2 (0.0%) 2 (0.0%) 2 (0.0%) 7,658 (28.6%)		0 (0.0%)	

14	fastestLap [object]	1. VN	18,507 (69.2%)		0 (0.0%)
		2. 50	309 (1.2%)		
15	rank [object]	3. 52	289 (1.1%)		0 (0.0%)
		4. 53	287 (1.1%)		
		5. 51	275 (1.0%)		
		6. 48	230 (0.9%)		
		7. 44	224 (0.8%)		
		8. 55	220 (0.8%)		
		9. 49	219 (0.8%)		
		10. 43	217 (0.8%)		
		11. other	5,982 (22.4%)		
16	fastestLapTime [object]	1. VN	18,507 (69.2%)		0 (0.0%)
		2. 1:18.904	4 (0.0%)		
		3. 1:43.026	4 (0.0%)		
		4. 1:14.117	4 (0.0%)		
		5. 1:17.495	4 (0.0%)		
		6. 1:18.262	4 (0.0%)		
		7. 1:18.069	3 (0.0%)		
		8. 1:18.462	3 (0.0%)		
		9. 1:34.090	3 (0.0%)		
		10. 1:35.816	3 (0.0%)		
		11. other	8,220 (30.7%)		
17	fastestLapSpeed [object]	1. VN	18,507 (69.2%)		0 (0.0%)
		2. 207.069	4 (0.0%)		
		3. 202.685	3 (0.0%)		
		4. 188.806	3 (0.0%)		
		5. 201.512	3 (0.0%)		
		6. 201.330	3 (0.0%)		
		7. 209.244	3 (0.0%)		
		8. 206.625	3 (0.0%)		
		9. 202.871	3 (0.0%)		
		10. 225.876	3 (0.0%)		
		11. other	8,224 (30.7%)		
18	statusId [int64]	11. other	0,224 (0.1%)		0 (0.0%)
		Mean (sd) : 17.2 (26.0) min < med < max: 1.0 < 10.0 < 141.0 IQR (CV) : 13.0 (0.7)	137 distinct values		

2 minutes ago (1s)

4

sumario_dados(races)

Dataset Info

Shape: (1125, 18)

Numero de Colunas: 18

Numero de Linhas: 1125

Dataset Info

Column types and missing data

0	raceId	1125	non-null	int64
1	year	1125	non-null	int64
2	round	1125	non-null	int64
3	circuitId	1125	non-null	int64
4	name	1125	non-null	object
5	date	1125	non-null	object
6	time	1125	non-null	object
7	url	1125	non-null	object
8	fp1_date	1125	non-null	object
9	fp1_time	1125	non-null	object
10	fp2_date	1125	non-null	object
11	fp2_time	1125	non-null	object
12	fp3_date	1125	non-null	object
13	fp3_time	1125	non-null	object
14	quali_date	1125	non-null	object
15	quali_time	1125	non-null	object
16	sprint_date	1125	non-null	object
17	sprint_time	1125	non-null	object

dtypes: int64(4), object(14)

memory usage: 158.3+ KB

None

Sumario de dados










Data Frame Summary

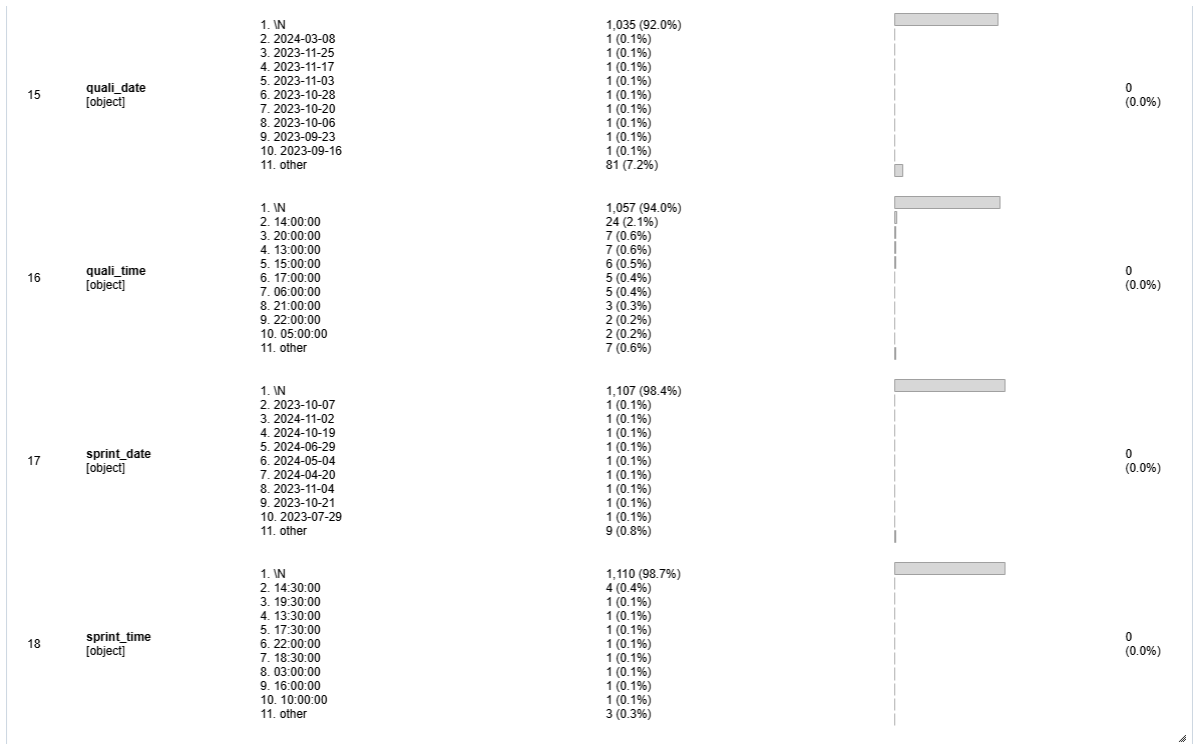
df

Dimensions: 1,125 x 18

Duplicates: 0

No	Variable	Stats / Values	Freqs / (% of Valid)	Graph	Missing
1	raceId [int64]	Mean (sd) : 565.7 (328.8) min < med < max: 1.0 < 563.0 < 1144.0 IQR (CV) : 563.0 (1.7)	1,125 distinct values		0 (0.0%)
2	year [int64]	Mean (sd) : 1992.7 (20.6) min < med < max: 1950.0 < 1994.0 < 2024.0 IQR (CV) : 34.0 (96.7)	75 distinct values		0 (0.0%)
3	round [int64]	Mean (sd) : 8.6 (5.2) min < med < max: 1.0 < 8.0 < 24.0 IQR (CV) : 9.0 (1.7)	24 distinct values		0 (0.0%)
4	circuitId [int64]	Mean (sd) : 23.9 (19.6) min < med < max: 1.0 < 18.0 < 80.0 IQR (CV) : 25.0 (1.2)	77 distinct values		0 (0.0%)
5	name [object]	1. Italian Grand Prix 2. British Grand Prix 3. Monaco Grand Prix 4. Belgian Grand Prix 5. German Grand Prix 6. French Grand Prix 7. Spanish Grand Prix 8. Canadian Grand Prix 9. Brazilian Grand Prix 10. United States Grand Prix 11. other	75 (6.7%) 75 (6.7%) 70 (6.2%) 69 (6.1%) 64 (5.7%) 62 (5.5%) 54 (4.8%) 53 (4.7%) 47 (4.2%) 45 (4.0%) 511 (45.4%)		0 (0.0%)
6	date [object]	1. 2009-03-29 2. 1960-06-06 3. 1960-11-20 4. 1960-09-04 5. 1960-08-14 6. 1960-07-16 7. 1960-07-03 8. 1960-06-19 9. 1960-05-30 10. 1959-05-30 11. other	1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 1,115 (99.1%)		0 (0.0%)

7	time [object]	1. \N	731 (85.0%)		0 (0.0%)
		2. 12:00:00	113 (10.0%)		
8	url [object]	3. 13:00:00	50 (4.4%)		0 (0.0%)
		4. 14:00:00	34 (3.0%)		
		5. 13:10:00	30 (2.7%)		
		6. 06:00:00	21 (1.9%)		
		7. 17:00:00	13 (1.2%)		
		8. 19:00:00	13 (1.2%)		
		9. 05:00:00	13 (1.2%)		
		10. 15:00:00	12 (1.1%)		
		11. other	95 (8.4%)		
		1. http://en.wikipedia.org/wiki/2	1 (0.1%)		
		2. http://en.wikipedia.org/wiki/1	1 (0.1%)		
9	fp1_date [object]	3. http://en.wikipedia.org/wiki/1	1 (0.1%)		0 (0.0%)
		4. http://en.wikipedia.org/wiki/1	1 (0.1%)		
		5. http://en.wikipedia.org/wiki/1	1 (0.1%)		
		6. http://en.wikipedia.org/wiki/1	1 (0.1%)		
		7. http://en.wikipedia.org/wiki/1	1 (0.1%)		
		8. http://en.wikipedia.org/wiki/1	1 (0.1%)		
		9. http://en.wikipedia.org/wiki/1	1 (0.1%)		
		10. http://en.wikipedia.org/wiki/1	1 (0.1%)		
		11. other	1,115 (99.1%)		
		1. \N	1,035 (92.0%)		
		2. 2024-03-07	1 (0.1%)		
10	fp1_time [object]	3. 2023-11-24	1 (0.1%)		0 (0.0%)
		4. 2023-11-16	1 (0.1%)		
		5. 2023-11-03	1 (0.1%)		
		6. 2023-10-27	1 (0.1%)		
		7. 2023-10-20	1 (0.1%)		
		8. 2023-10-06	1 (0.1%)		
		9. 2023-09-22	1 (0.1%)		
		10. 2023-09-15	1 (0.1%)		
		11. other	81 (7.2%)		
		1. \N	1,057 (94.0%)		
		2. 11:30:00	18 (1.6%)		
11	fp2_date [object]	3. 12:00:00	8 (0.7%)		0 (0.0%)
		4. 09:30:00	6 (0.5%)		
		5. 10:30:00	4 (0.4%)		
		6. 13:30:00	4 (0.4%)		
		7. 17:30:00	4 (0.4%)		
		8. 02:30:00	3 (0.3%)		
		9. 18:30:00	3 (0.3%)		
		10. 18:00:00	3 (0.3%)		
		11. other	15 (1.3%)		
		1. \N	1,035 (92.0%)		
		2. 2024-03-07	1 (0.1%)		
12	fp2_time [object]	3. 2023-11-24	1 (0.1%)		0 (0.0%)
		4. 2023-11-16	1 (0.1%)		
		5. 2023-11-04	1 (0.1%)		
		6. 2023-10-27	1 (0.1%)		
		7. 2023-10-21	1 (0.1%)		
		8. 2023-10-07	1 (0.1%)		
		9. 2023-09-22	1 (0.1%)		
		10. 2023-09-15	1 (0.1%)		
		11. other	81 (7.2%)		
		1. \N	1,057 (94.0%)		
		2. 15:00:00	22 (2.0%)		
13	fp3_date [object]	3. 13:00:00	8 (0.7%)		0 (0.0%)
		4. 06:00:00	5 (0.4%)		
		5. 10:30:00	4 (0.4%)		
		6. 14:00:00	4 (0.4%)		
		7. 21:00:00	4 (0.4%)		
		8. 17:00:00	3 (0.3%)		
		9. 21:30:00	3 (0.3%)		
		10. 22:00:00	3 (0.3%)		
		11. other	12 (1.1%)		
		1. \N	1,053 (93.6%)		
		2. 2022-10-29	1 (0.1%)		
14	fp3_time [object]	3. 2023-11-17	1 (0.1%)		0 (0.0%)
		4. 2023-10-28	1 (0.1%)		
		5. 2023-09-23	1 (0.1%)		
		6. 2023-09-16	1 (0.1%)		
		7. 2023-09-02	1 (0.1%)		
		8. 2023-08-26	1 (0.1%)		
		9. 2023-07-22	1 (0.1%)		
		10. 2023-07-08	1 (0.1%)		
		11. other	63 (5.6%)		
		1. \N	1,072 (95.3%)		
		2. 10:30:00	14 (1.2%)		
		3. 11:00:00	9 (0.8%)		0 (0.0%)
		4. 09:30:00	4 (0.4%)		
		5. 02:30:00	3 (0.3%)		
		6. 17:00:00	3 (0.3%)		
		7. 16:30:00	3 (0.3%)		
		8. 01:30:00	2 (0.2%)		
		9. 17:30:00	2 (0.2%)		
		10. 13:30:00	2 (0.2%)		
		11. other	11 (1.0%)		



Just now (1s)

5

Python

sumario_dados(drivers)

Dataset Info

Shape: (861, 9)

Numero de Colunas: 9

Numero de Linhas: 861

Dataset Info

Column types and missing data

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 861 entries, 0 to 860
Data columns (total 9 columns):
#   Column      Non-Null Count  Dtype
---  -
0   driverId    861 non-null   int64
1   driverRef   861 non-null   object
2   number      861 non-null   object
3   code        861 non-null   object
4   forename    861 non-null   object
5   surname     861 non-null   object
6   dob         861 non-null   object
7   nationality  861 non-null   object
8   url         861 non-null   object
dtypes: int64(1), object(8)
memory usage: 60.7+ KB
None
```










Sumario de datos

Data Frame Summary

df

Dimensions: 861 x 9

Duplicates: 0

No	Variable	Stats / Values	Freqs / (% of Valid)	Graph	Missing
1	driverId [int64]	Mean (sd) : 431.1 (248.8) min < med < max: 1.0 < 431.0 < 862.0 IQR (CV) : 430.0 (1.7)	861 distinct values		0 (0.0%)
2	driverRef [object]	1. hamilton 2. shelby 3. fontes 4. ashdawn 5. bill_moss 6. dennis_taylor 7. blanchard 8. tomaso 9. constantine 10. said 11. other	1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 851 (98.8%)		0 (0.0%)
3	number [object]	1. \N 2. 22 3. 99 4. 21 5. 2 6. 10 7. 4 8. 9 9. 88 10. 28 11. other	802 (93.1%) 2 (0.2%) 2 (0.2%) 2 (0.2%) 2 (0.2%) 2 (0.2%) 2 (0.2%) 2 (0.2%) 2 (0.2%) 2 (0.2%) 41 (4.8%)		0 (0.0%)
4	code [object]	1. \N 2. BIA 3. MSC 4. HAR 5. MAG 6. VER 7. ALB 8. DOO 9. BOT 10. NAS 11. other	757 (87.9%) 2 (0.2%) 2 (0.2%) 2 (0.2%) 2 (0.2%) 2 (0.2%) 2 (0.2%) 2 (0.2%) 1 (0.1%) 1 (0.1%) 88 (10.2%)		0 (0.0%)
5	forename [object]	1. John 2. Mike 3. Peter 4. Bill 5. Tony 6. Bob 7. David 8. Paul 9. Johnny 10. George 11. other	14 (1.6%) 14 (1.6%) 13 (1.5%) 11 (1.3%) 11 (1.3%) 10 (1.2%) 10 (1.2%) 9 (1.0%) 9 (1.0%) 8 (0.9%) 752 (87.3%)		0 (0.0%)
6	surname [object]	1. Taylor 2. Wilson 3. Fittipaldi 4. Stewart 5. Russo 6. Schumacher 7. Brown 8. Brabham 9. Hill 10. Winkelhock 11. other	5 (0.6%) 4 (0.5%) 4 (0.5%) 3 (0.3%) 3 (0.3%) 3 (0.3%) 3 (0.3%) 3 (0.3%) 3 (0.3%) 3 (0.3%) 827 (96.1%)		0 (0.0%)
7	dob [object]	1. 1919-10-28 2. 1946-12-12 3. 1932-05-05 4. 1919-09-30 5. 1929-06-13 6. 1915-10-26 7. 1920-02-16 8. 1942-05-27 9. 1931-05-18 10. 1937-04-26 11. other	2 (0.2%) 2 (0.2%) 2 (0.2%) 2 (0.2%) 2 (0.2%) 2 (0.2%) 2 (0.2%) 2 (0.2%) 2 (0.2%) 2 (0.2%) 841 (97.7%)		0 (0.0%)
8	nationality [object]	1. British 2. American 3. Italian 4. French 5. German 6. Brazilian 7. Argentine 8. Belgian 9. Swiss 10. South African 11. other	166 (19.3%) 158 (18.4%) 99 (11.5%) 73 (8.5%) 50 (5.8%) 32 (3.7%) 24 (2.8%) 23 (2.7%) 23 (2.7%) 23 (2.7%) 190 (22.1%)		0 (0.0%)
9	url [object]	1. http://en.wikipedia.org/wiki/L 2. http://en.wikipedia.org/wiki/C 3. http://en.wikipedia.org/wiki/A 4. http://en.wikipedia.org/wiki/P 5. http://en.wikipedia.org/wiki/B 6. http://en.wikipedia.org/wiki/D 7. http://en.wikipedia.org/wiki/H 8. http://en.wikipedia.org/wiki/I 9. http://en.wikipedia.org/wiki/G 10. http://en.wikipedia.org/wiki/B 11. other	1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 1 (0.1%) 851 (98.8%)		0 (0.0%)

2.3 Carga - Pipeline de ETL dos dados

Foi utilizado uma pipeline baseado em estruturas como o “Medallion Architecture”, organizando os dados em 3 camadas principais: Bronze, Silver e Gold, cada uma com funções e objetivos específicos.

Fluxo do Pipeline

1. **Ingestão** (Bronze) → 2. **Transformação** (Silver) → 3. **Modelagem** (Gold) → 4. **Consumo**.

Essa abordagem garante **escalabilidade**, **qualidade** e **confiabilidade** dos dados em cada etapa.

2.3.1 - Camada Bronze

Armazena os dados brutos, exatamente como foram coletados da fonte.

- **Objetivos:**
 - Preservar os dados originais sem modificações.
 - Garantir rastreabilidade e auditoria.
 - Permitir reproprocessamento em caso de falhas.

```
▶ Last execution failed 7
1 %sql DROP DATABASE bronze CASCADE;
```

❌ > AnalysisException: [SCHEMA_NOT_FOUND] The schema `bronze` cannot be found. Verify the spelling and correctness of the schema and catalog. If you did not qualify the name with a catalog, verify the current_schema() output, or qualify the name with the correct catalog. To tolerate the error on drop use DROP SCHEMA IF EXISTS.

```
▶ 09:37 AM (1s) 8
%sql CREATE DATABASE bronze;
```

_sqlDf: pyspark.sql.dataframe.DataFrame
OK

```
▶ 09:37 AM (33s) 9
from pyspark.sql.functions import current_timestamp, lit
import uuid

batch_id = str(uuid.uuid4())

name_races_spark_df = spark.createDataFrame(races)
bronze_races_df = name_races_spark_df.withColumn("ingestion_timestamp", current_timestamp()) \
    .withColumn("batch_id", lit(batch_id))

(bronze_races_df.write
 .format("delta")
 .mode("overwrite") # ou "append" para pipelines incrementais
 .option("overwriteSchema", "true")
 .option("delta.enableChangeDataFeed", "true") # Habilita CDC para futuras integrações
 .saveAsTable("bronze.races"))
```

▶ (10) Spark Jobs

```
09:37 AM (8s) 11
name_drivers_spark_df = spark.createDataFrame(drivers)
bronze_drivers_df = name_drivers_spark_df.withColumn("ingestion_timestamp", current_timestamp()) \
                                          .withColumn("batch_id", lit(batch_id))

(bronze_drivers_df.write
 .format("delta")
 .mode("overwrite") # ou "append" para pipelines incrementais
 .option("overwriteSchema", "true")
 .option("delta.enableChangeDataFeed", "true") # Habilita CDC para futuras integrações
 .saveAsTable("bronze.drivers"))

(10) Spark Jobs
bronze_drivers_df: pyspark.sql.dataframe.DataFrame = [driverId: long, driverRef: string ... 9 more fields]
name_drivers_spark_df: pyspark.sql.dataframe.DataFrame = [driverId: long, driverRef: string ... 7 more fields]
```

```
09:38 AM (11s) 13
name_results_spark_df = spark.createDataFrame(results)
bronze_results_df = name_results_spark_df.withColumn("ingestion_timestamp", current_timestamp()) \
                                              .withColumn("batch_id", lit(batch_id))

(bronze_results_df.write
 .format("delta")
 .mode("overwrite") # ou "append" para pipelines incrementais
 .option("overwriteSchema", "true")
 .option("delta.enableChangeDataFeed", "true") # Habilita CDC para futuras integrações
 .saveAsTable("bronze.results"))

(10) Spark Jobs
bronze_results_df: pyspark.sql.dataframe.DataFrame = [resultId: long, raceId: long ... 18 more fields]
name_results_spark_df: pyspark.sql.dataframe.DataFrame = [resultId: long, raceId: long ... 16 more fields]
```

Na camada Bronze de um data lake ou pipeline de dados, a inclusão de colunas como `ingestion_time` e `batch_id` é uma prática comum e tem vários propósitos importantes:

- 1. Rastreabilidade e Auditoria
- 2. Controle de Processamento
- 3. Detecção de Dados Duplicados ou Atrasados
- 4. Separação de Responsabilidades

A camada Bronze armazena os dados **cru** (raw), sem transformações, então metadados como `ingestion_time` e `batch_id` garantem que a origem e o contexto sejam preservados.

Nas camadas Silver/Gold, esses campos podem ser removidos ou usados para filtros.

2.3.2 - Camada Silver

Transforma os dados brutos em um formato estruturado e limpo.

- **Objetivos:**

- Aplicar validação, limpeza e normalização.
- Realizar enriquecimento (junção de fontes, deduplicação).
- Estruturar em schemas otimizados (Delta Lake).

```
1 %sql DROP DATABASE silver CASCADE
```

❌ Last execution failed 15

❌ > AnalysisException: [SCHEMA_NOT_FOUND] The schema `silver` cannot be found. Verify the spelling and correctness of the schema and catalog. If you did not qualify the name with a catalog, verify the current_schema() output, or qualify the name with the correct catalog. To tolerate the error on drop use DROP SCHEMA IF EXISTS.

```
%sql CREATE DATABASE silver
```

✓ 09:38 AM (<1s) 16

_sqlidf: pyspark.sql.dataframe.DataFrame

OK

```
silver_races_df = bronze_races_df.withColumn("processing_timestamp", current_timestamp())\
    .drop("circuitId", "time", "url", "fp1_date", "fp1_time", "fp2_date", "fp2_time", "fp3_date", "fp3_time",
         "quali_date", "quali_time", "sprint_date", "sprint_time")

(silver_races_df.write
 .format("delta")
 .mode("overwrite") # Ou append
 .option("overwriteSchema", "true")
 .option("delta.enableChangeDataFeed", "true")
 .saveAsTable("silver.races"))
```

✓ 09:38 AM (5s) 17 Python

```
silver_drivers_df = (bronze_drivers_df
 .withColumn("processing_timestamp", current_timestamp())
 .withColumn("name", concat(col("forename"), lit(" "), col("surname")))
 .drop("url")
)

(silver_drivers_df.write
 .format("delta")
 .mode("overwrite")
 .option("overwriteSchema", "true")
 .option("delta.enableChangeDataFeed", "true")
 .saveAsTable("silver.drivers"))
```

▶ (10) Spark Jobs

▶ silver_drivers_df: pyspark.sql.dataframe.DataFrame = [driverId: long, driverRef: string ... 10 more fields]



```

silver_results_df = (
    bronze_results_df
    .withColumn("processing_timestamp", current_timestamp())
    .withColumn(
        "positionText",
        when(col("positionText") == "R", "Retired")
        .when(col("positionText") == "D", "Disqualified")
        .when(col("positionText") == "E", "Excluded")
        .when(col("positionText") == "W", "Withdrawn")
        .when(col("positionText") == "F", "Failed to qualify")
        .when(col("positionText") == "N", "Not Classified")
        .otherwise("Finished")
    )
    .drop("number", "position", "statusId", "alt")
)

(
    silver_results_df.write
    .format("delta")
    .mode("overwrite")
    .option("mergeSchema", "true")
    .option("delta.enableChangeDataFeed", "true")
    .saveAsTable("silver.results")
)

```

▶ (10) Spark Jobs

▶  silver_results_df: pyspark.sql.dataframe.DataFrame = [resultId: long, raceId: long ... 16 more fields]

2.3.3 - Camada Gold

Disponibilizar dados agregados e modelados para consumo final (BI, relatórios, ML).

- **Objetivos:**

- Criar métricas, KPIs e modelos de negócio.
- Otimizar para consultas rápidas (OLAP).
- Garantir conformidade com regras de governança.

```
▶ Last execution failed 22
1 %sql DROP DATABASE gold CASCADE

> AnalysisException: [SCHEMA_NOT_FOUND] The schema 'gold' cannot be found. Verify the spelling and correctness of the schema and catalog.
If you did not qualify the name with a catalog, verify the current_schema() output, or qualify the name with the correct catalog.
To tolerate the error on drop use DROP SCHEMA IF EXISTS.
```

```
▶ 09:42 AM (<1s) 23
%sql CREATE DATABASE gold

_sqldf: pyspark.sql.dataframe.DataFrame
OK
```

```
▶ 09:42 AM (7s) 24

from pyspark.sql.functions import monotonically_increasing_id

dim_drivers = silver_drivers_df.select(
    "driverId", "name", "dob", "nationality").dropDuplicates()

# Adicionar chave substituta (surrogate keys)
dim_drivers = dim_drivers.withColumn("sk_driver", monotonically_increasing_id()+1)

(dim_drivers.write
 .format("delta")
 .mode("overwrite") # Ou "append" para cargas incrementais
 .option("overwriteSchema", "true") # Permite evolução do schema
 .option("delta.enableChangeDataFeed", "true") # Habilita Change Data Feed
 .saveAsTable("gold.dim_drivers"))

▶ (11) Spark Jobs
dim_drivers: pyspark.sql.dataframe.DataFrame = [driverId: long, name: string ... 3 more fields]
```

```
▶ 09:43 AM (7s) 26

from pyspark.sql.functions import monotonically_increasing_id

dim_races = silver_races_df.select(
    "raceId", "year", "round", "name", "date",).dropDuplicates()

# Adicionar chave substituta (surrogate keys)
dim_races = dim_races.withColumn("sk_races", monotonically_increasing_id()+1)

(dim_races.write
 .format("delta")
 .mode("overwrite") # Ou "append" para cargas incrementais
 .option("overwriteSchema", "true") # Permite evolução do schema
 .option("delta.enableChangeDataFeed", "true") # Habilita Change Data Feed
 .saveAsTable("gold.dim_races"))

▶ (11) Spark Jobs
dim_races: pyspark.sql.dataframe.DataFrame = [raceId: long, year: long ... 4 more fields]
```

```
09:43 AM (13s) 28

from pyspark.sql.functions import broadcast, year, month

fato_results = (silver_results_df.alias("s")
    .join(
        broadcast(dim_drivers.select("driverId", "sk_driver").alias("ddriver")),
        "driverId"
    )
    .join(
        broadcast(dim_races.select("raceId", "sk_races").alias("drace")),
        "raceId"
    )
    .select(
        "sk_races",
        "sk_driver",
        col("s.grid"),
        col("s.positionText"),
        col("s.positionOrder"),
        col("s.points"),
        col("s.laps"),
        col("s.time"),
        col("s.milliseconds"),
        col("s.fastestLap"),
        col("s.rank"),
        col("s.fastestLapTime"),
        col("s.fastestLapSpeed")
    )
)

(fato_results.write
    .format("delta")
    .mode("overwrite") # Ou "append" para cargas incrementais
    .option("overwriteSchema", "true") # Permite evolução do schema
    .option("delta.enableChangeDataFeed", "true") # Habilita Change Data Feed
    .saveAsTable("gold.fato_results"))
```

▶ (14) Spark Jobs

▶ fato_results: pyspark.sql.dataframe.DataFrame = [sk_races: long, sk_driver: long ... 11 more fields]

2.4 Análise dos Dados

2.4.1 Análise de Qualidade dos Dados

- Diante da análise de todos os atributos nas tabelas apresentadas no módulo: 2.2.1 - Catálogo de Dados podemos inferir que não apresentam dados nulos e nenhum outlier muito evidente que pode comprometer as respostas e resultados que esperamos solucionar. Trata-se de um dataset disponibilizado no Kaggle e provavelmente já passou por alguma curadoria e tratamento de dados, tomamos proveito disso e facilitou o nosso trabalho. Com certeza se estivéssemos trabalhando com dados vindo direto de alguma API o nosso ETL de limpeza seria bem mais complexo.
- Validação de domínios das tabelas:
Nesse momento buscamos encontrar alguma inconsistência nas chaves, se há alguma SK (id) duplicada ou sem “join”. Segue código:

```
%python
df = spark.sql("""
SELECT
    COUNT(DISTINCT fr.sk_driver) AS total_drivers_na_fato,
    COUNT(DISTINCT dd.sk_driver) AS drivers_com_match,
    COUNT(DISTINCT CASE WHEN dd.sk_driver IS NULL THEN fr.sk_driver END) AS drivers_orfaos,
    ROUND(100.0 * COUNT(DISTINCT CASE WHEN dd.sk_driver IS NULL THEN fr.sk_driver END) /
        NULLIF(COUNT(DISTINCT fr.sk_driver), 0), 2) AS percentual_orfaos
FROM gold.fato_results fr
LEFT JOIN gold.dim_drivers dd ON fr.sk_driver = dd.sk_driver
""")
display(df)
```

▶ (6) Spark Jobs

▶ df: pyspark.sql.dataframe.DataFrame = [total_drivers_na_fato: long, drivers_com_match: long ... 2 more fields]

	¹ total_drivers_na_fato	¹ drivers_com_match	¹ drivers_orfaos	.00 percentual_orfaos
1	861	861	0	0.00

1 row | 10.65s runtime

Refreshed 9 minutes ago

01:47 PM (3s)
20

```

%python
df = spark.sql("""
SELECT
    COUNT(DISTINCT fr.sk_races) AS total_races_na_fato,
    COUNT(DISTINCT dd.sk_races) AS races_com_match,
    COUNT(DISTINCT CASE WHEN dd.sk_races IS NULL THEN fr.sk_races END) AS races_orfaos,
    ROUND(100.0 * COUNT(DISTINCT CASE WHEN dd.sk_races IS NULL THEN fr.sk_races END) /
        NULLIF(COUNT(DISTINCT fr.sk_races), 0), 2) AS percentual_orfaos
FROM gold.fato_results fr
LEFT JOIN gold.dim_races dd ON fr.sk_races = dd.sk_races
""")
display(df)

```

(4) Spark Jobs

df: pyspark.sql.dataframe.DataFrame = [total_races_na_fato: long, races_com_match: long ... 2 more fields]

Table

	total_races_na_fato	races_com_match	races_orfaos	.00 percentual_orfaos
1	1125	1125	0	0.00

1 row | 3.40s runtime

Refreshed 7 minutes ago

2.4.2 Resposta às Perguntas do Objetivo

- ☐ Análise da temporada passada: Qual piloto fez a volta mais rápida de cada Grand Prix em 2024? E quem ganhou cada corrida? Quem venceu mais corridas?


O piloto que fez a volta mais rápida de cada circuito em 2024 foi:

01:43 PM (8s)21SQL

```
%sql
SELECT
  t1.fastestLapTime as voltaRapida,
  t3.name as grandPrix,
  t2.name as piloto
FROM gold.fato_results AS t1
INNER JOIN gold.dim_drivers AS t2 ON t1.sk_driver = t2.sk_driver
INNER JOIN gold.dim_races AS t3 ON t1.sk_races = t3.sk_races
WHERE t3.year = 2024
AND t1.fastestLapTime = (
  SELECT MIN(t1_sub.fastestLapTime)
  FROM gold.fato_results AS t1_sub
  WHERE t1_sub.sk_races = t1.sk_races
)
ORDER BY t3.name
```

▶ (6) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [voltaRapida: string, grandPrix: string ... 1 more field]

▶  _sqldf: pyspark.sql.dataframe.DataFrame = [voltaRapida: string, grandPrix: string ... 1 more field]

Table  

	 voltaRapida	 grandPrix	 piloto
1	1:25.637	Abu Dhabi Grand Prix	Kevin Magnuss...
2	1:19.813	Australian Grand Prix	Charles Leclerc
3	1:07.694	Austrian Grand Prix	Fernando Alonso
4	1:45.255	Azerbaijan Grand Prix	Lando Norris
5	1:32.608	Bahrain Grand Prix	Max Verstappen
6	1:44.701	Belgian Grand Prix	Sergio Pérez
7	1:28.293	British Grand Prix	Carlos Sainz
8	1:14.856	Canadian Grand Prix	Lewis Hamilton
9	1:37.810	Chinese Grand Prix	Fernando Alonso
10	1:13.817	Dutch Grand Prix	Lando Norris
11	1:18.589	Emilia Romagna Grand P...	George Russell
12	1:20.305	Hungarian Grand Prix	George Russell
13	1:21.432	Italian Grand Prix	Lando Norris
14	1:33.706	Japanese Grand Prix	Max Verstappen
15	1:34.876	Las Vegas Grand Prix	Lando Norris
16	1:18.336	Mexico City Grand Prix	Charles Leclerc
17	1:30.634	Miami Grand Prix	Oscar Piastri

	Δ_C^B voltaRapida	Δ_C^B grandPrix	Δ_C^B piloto
27	\N	Monaco Grand Prix	Fernando Alonso
28	\N	Monaco Grand Prix	Pierre Gasly
29	\N	Monaco Grand Prix	Alexander Albon
30	\N	Monaco Grand Prix	Yuki Tsunoda
31	\N	Monaco Grand Prix	Lewis Hamilton
32	\N	Monaco Grand Prix	Max Verstappen
33	\N	Monaco Grand Prix	George Russell
34	\N	Monaco Grand Prix	Lando Norris
35	\N	Monaco Grand Prix	Carlos Sainz
36	\N	Monaco Grand Prix	Oscar Piastri
37	\N	Monaco Grand Prix	Charles Leclerc
38	1:22.384	Qatar Grand Prix	Lando Norris
39	1:31.632	Saudi Arabian Grand Prix	Charles Leclerc
40	1:34.486	Singapore Grand Prix	Daniel Ricciardo
41	1:17.115	Spanish Grand Prix	Lando Norris
42	1:20.472	São Paulo Grand Prix	Max Verstappen
43	1:37.330	United States Grand Prix	Esteban Ocon

↓ 43 rows | 8.09s runtime

O Vencedor de cada corrida de 2024:

▶ 01:43 PM (3s) 23

```
%sql
SELECT
  t1.positionOrder,
  t3.name as grandPrix,
  t2.name as piloto
FROM gold.f1_results AS t1
INNER JOIN gold.dim_drivers AS t2 ON t1.sk_driver = t2.sk_driver
INNER JOIN gold.dim_races AS t3 ON t1.sk_races = t3.sk_races
WHERE t3.year = 2024
AND t1.positionOrder = 1
```

▶ (5) Spark Jobs

▶ `_sqldf: pyspark.sql.dataframe.DataFrame = [positionOrder: long, grandPrix: string ... 1 more field]`

Table ▾ +

	positionOrder	grandPrix	piloto
10	1	Spanish Grand Prix	Max Verstappen
11	1	Austrian Grand Prix	George Russell
12	1	British Grand Prix	Lewis Hamilton
13	1	Hungarian Grand Prix	Oscar Piastri
14	1	Belgian Grand Prix	Lewis Hamilton
15	1	Dutch Grand Prix	Lando Norris
16	1	Italian Grand Prix	Charles Leclerc
17	1	Azerbaijan Grand Prix	Oscar Piastri
18	1	Singapore Grand Prix	Lando Norris
19	1	United States Grand Prix	Charles Leclerc
20	1	Mexico City Grand Prix	Carlos Sainz
21	1	São Paulo Grand Prix	Max Verstappen
22	1	Las Vegas Grand Prix	George Russell
23	1	Qatar Grand Prix	Max Verstappen
24	1	Abu Dhabi Grand Prix	Lando Norris

Quem mais venceu corridas em 2024 foi:

01:43 PM (4s)24

```
%sql
SELECT
  count(t1.positionOrder),
  t2.name as piloto
FROM gold.f1_results AS t1
INNER JOIN gold.dim_drivers AS t2 ON t1.sk_driver = t2.sk_driver
INNER JOIN gold.dim_races AS t3 ON t1.sk_races = t3.sk_races
WHERE t3.year = 2024
AND t1.positionOrder = 1
GROUP BY t2.name
order by count(t1.positionOrder) desc
```

(4) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [count(positionOrder): long, piloto: string]

Table +

	count(positionOrder)	piloto
1	9	Max Verstappen
2	4	Lando Norris
3	3	Charles Leclerc
4	2	Carlos Sainz
5	2	Oscar Piastri
6	2	Lewis Hamilton
7	2	George Russell

7 rows | 3.62s runtime

☐ Quem são os pilotos mais consistentes da história? (Pilotos com maior percentagem de chegadas no pódio quando começaram no top 5 do grid)

01:43 PM (4s)28

```
%sql
SELECT
  t2.name AS piloto,
  COUNT(CASE WHEN t1.positionOrder <= 3 THEN 1 END) * 100.0 / COUNT(*) AS percentagem_podio,
  COUNT(*) AS corridas_top5_grid
FROM gold.f1_results t1
JOIN gold.dim_drivers t2 ON t1.sk_driver = t2.sk_driver
JOIN gold.dim_races t3 ON t1.sk_races = t3.sk_races
WHERE t1.grid <= 5
GROUP BY t2.name
HAVING COUNT(*) >= 20 -- Mínimo de 20 corridas para ser relevante
ORDER BY percentagem_podio DESC
LIMIT 10
```

(4) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [piloto: string, percentagem_podio: decimal(38,14) ... 1 more field]

Table ▼ +

	A ^B _C piloto	.00 porcentagem_podio	1 ² ₃ corridas_top5_grid
1	Michael Schumach...	67.44186046511628	215
2	Lewis Hamilton	67.15328467153285	274
3	Max Verstappen	64.00000000000000	150
4	Juan Fangio	62.74509803921569	51
5	Fernando Alonso	62.67605633802817	142
6	Sebastian Vettel	62.28571428571429	175
7	Eddie Irvine	60.00000000000000	30
8	Alan Jones	60.00000000000000	30
9	Nico Rosberg	59.78260869565217	92
10	Alain Prost	58.75000000000000	160

↓ 10 rows | 3.71s runtime

☐ Top 10 países com mais vitórias"? (Evolução das nacionalidades vencedoras ao longo do tempo

```

▶ 01:43 PM (3s) 30

%sql
SELECT
  t2.nationality AS nacionalidade,
  COUNT(*) AS vitorias
FROM gold.fato_results t1
JOIN gold.dim_drivers t2 ON t1.sk_driver = t2.sk_driver
JOIN gold.dim_races t3 ON t1.sk_races = t3.sk_races
WHERE t1.positionOrder = 1
GROUP BY nacionalidade
ORDER BY vitorias DESC
limit 10

▶ (4) Spark Jobs

▶ _sqldf: pyspark.sql.dataframe.DataFrame = [nacionalidade: string, vitorias: long]

```

	A ^B _C nacionalidade	1 ² ₃ vitorias
1	British	317
2	German	179
3	Brazilian	101
4	French	81
5	Dutch	63
6	Finnish	57
7	Australian	45
8	Italian	43
9	Austrian	41
10	Argentine	38

↓ 10 rows | 3.28s runtime

☐ Qual nacionalidade produziu os pilotos mais rápidos?

```

▶ 01:43 PM (4s) 40

%sql
SELECT
  t2.nationality AS nacionalidade,
  ROUND(AVG(t1.fastestLapSpeed), 2) AS velocidade_media_kmh,
  COUNT(DISTINCT t2.name) AS pilotos
FROM gold.f1o_results t1
JOIN gold.dim_drivers t2 ON t1.sk_driver = t2.sk_driver
GROUP BY t2.nationality
HAVING COUNT(*) >= 30
ORDER BY velocidade_media_kmh DESC

▶ (4) Spark Jobs

▶ _sqldf: pyspark.sql.dataframe.DataFrame = [nacionalidade: string, velocidade_media_kmh: double ... 1 more fi

```

Table  

	A_C^B nacionalidade	1.2 velocidade_media_kmh	1_3^2 pilotos
1	Monegasque	211.79	4
2	Colombian	211.38	3
3	Thai	210.95	2
4	Canadian	209.12	14
5	Chinese	209.02	1
6	Dutch	207.72	18
7	Austrian	205.7	15
8	Danish	205.63	5
9	New Zealander	205.62	10
10	Australian	205.27	19
11	Finnish	204.96	9
12	British	204.82	166
13	Spanish	204.81	15
14	Mexican	204.3	6
15	American	204.25	158

☐ Quais são os maiores vencedores da história?

▶

✓ 01:43 PM (3s)

```
%sql
SELECT
  t2.name,
  COUNT(*) AS vitorias
FROM gold.fato_results t1
JOIN gold.dim_drivers t2 ON t1.sk_driver = t2.sk_driver
JOIN gold.dim_races t3 ON t1.sk_races = t3.sk_races
WHERE t1.positionOrder = 1
GROUP BY t2.name
ORDER BY vitorias DESC
limit 10
```

▶ (4) Spark Jobs

▶ _sqldf: pyspark.sql.dataframe.DataFrame = [name: string, vitorias: long]

Table ▼ +

	^A _C name	¹ ₃ vitorias
1	Lewis Hamilton	105
2	Michael Schumach...	91
3	Max Verstappen	63
4	Sebastian Vettel	53
5	Alain Prost	51
6	Ayrton Senna	41
7	Fernando Alonso	32
8	Nigel Mansell	31
9	Jackie Stewart	27
10	Jim Clark	25

⬇ 10 rows | 2.61s runtime

- ☐ Quem são os maiores "começam mal, terminam bem"? (Pilotos com maior diferença média entre posição de largada e chegada)

01:43 PM (3s) 36

```
%sql
SELECT
  t2.name AS piloto,
  AVG(t1.grid - t1.positionOrder) AS media_ultrapassagem,
  COUNT(*) AS corridas,
  MAX(t1.grid - t1.positionOrder) AS maior_ganho_unico
FROM gold.f1_results t1
JOIN gold.dim_drivers t2 ON t1.sk_driver = t2.sk_driver
WHERE t1.positionOrder > 0 -- Exclui DNFs
GROUP BY t2.name
HAVING COUNT(*) >= 50
ORDER BY media_ultrapassagem DESC
LIMIT 10
```

▶ (3) Spark Jobs

▶ _sqldf: pyspark.sql.dataframe.DataFrame = [piloto: string, media_ultrapassagem: double ... 2 more fields]

Table +

	^A _C piloto	1.2 media_ultrapassagem	¹ ₃ corridas	¹ ₃ maior_ganho_unico
1	Jonathan Palm...	4.6477272727272725	88	19
2	Philippe Streiff	4.481481481481482	54	19
3	Luca Badoer	4.344827586206897	58	17
4	Marc Surer	3.0568181818181817	88	20
5	Mika Salo	2.5225225225225225	111	15
6	Pedro Diniz	2.474747474747475	99	15
7	Érik Comas	2.1904761904761907	63	18
8	Henri Pescarolo	2.1016949152542375	59	17
9	Jos Verstappen	2.0841121495327104	107	14
10	Marcus Ericsson	1.9175257731958764	97	11

❑ Qual piloto teve a carreira mais longa?

▶

✓ 01:43 PM (3s)

38

```
%sql
SELECT
  t1.name AS piloto,
  MIN(t2.year) AS primeiro_ano,
  MAX(t2.year) AS ultimo_ano,
  MAX(t2.year) - MIN(t2.year) AS anos_de_carreira
FROM gold.dim_drivers t1
JOIN gold.fato_results f ON t1.sk_driver = f.sk_driver
JOIN gold.dim_races t2 ON f.sk_races = t2.sk_races
GROUP BY t1.name
ORDER BY anos_de_carreira DESC
LIMIT 10
```

▶ (4) Spark Jobs

▶

_sqldf: pyspark.sql.dataframe.DataFrame = [piloto: string, primeiro_ano: long ... 2 more fields]

Table ▾

+

	^A _C piloto	¹ ₃ primeiro_ano	¹ ₃ ultimo_ano	¹ ₃ anos_de_carreira
1	Fernando Alonso	2001	2024	23
2	Michael Schumach...	1991	2012	21
3	Kimi Räikkönen	2001	2021	20
4	Rubens Barrichello	1993	2011	18
5	Lewis Hamilton	2007	2024	17
6	Jenson Button	2000	2017	17
7	Graham Hill	1958	1975	17
8	Luca Badoer	1993	2009	16
9	Riccardo Patrese	1977	1993	16
10	Jack Brabham	1955	1970	15

↓

10 rows | 3.44s runtime

- ☐ Last Heros (Maiores recuperações em uma única corrida - do último grid ao pódio)

▶ ✓ 01:43 PM (2s) 42

```
%sql
SELECT
    t3.year,
    t3.name AS circuito,
    t2.name AS piloto,
    t1.grid AS posicao_largada,
    t1.positionOrder AS posicao_chegada,
    t1.grid - t1.positionOrder AS posicoes_ganhas,
    t1.time AS tempo_total
FROM gold.fato results t1
JOIN gold.dim_drivers t2 ON t1.sk_driver = t2.sk_driver
JOIN gold.dim_races t3 ON t1.sk_races = t3.sk_races
WHERE t1.positionOrder = 1
AND t1.grid >= 15
ORDER BY posicoes_ganhas DESC
LIMIT 20
```

▶ (3) Spark Jobs

▶ _sqlidf: pyspark.sql.dataframe.DataFrame = [year: long, circuito: string ... 5 more fields]

Table ▾ +

	² ₃ year	⁰ ₀ circuito	⁰ ₀ piloto	² ₃ posicao_largada	² ₃ posicao_chegada	² ₃ posicoes_ganhas	⁰ ₀ tempo_total
1	1983	United States Grand Prix West	John Watson	22	1	21	1:53:34.889
2	1954	Indianapolis 500	Bill Vukovich	19	1	18	3:49:17.27
3	2000	German Grand Prix	Rubens Barrichello	18	1	17	1:25:34.418
4	2024	São Paulo Grand Prix	Max Verstappen	17	1	16	2:06:54.430
5	1982	Detroit Grand Prix	John Watson	17	1	16	1:58:41.043
6	2005	Japanese Grand Prix	Kimi Räikkönen	17	1	16	1:29:02.212
7	1995	Belgian Grand Prix	Michael Schumach...	16	1	15	1:36:47.875
8	1973	South African Grand Prix	Jackie Stewart	16	1	15	1:43:11.07
9	2008	Singapore Grand Prix	Fernando Alonso	15	1	14	1:57:16.304

↓ 9 rows | 2.50s runtime

□ Tributo Ayrton Senna, todas as provas que ele ganhou:

▶
01:43 PM (3s)
26

```

%sql
SELECT
    count(t1.positionOrder),
    t3.name as grandPrix,
    t2.name as piloto
FROM gold.f1o1_results AS t1
INNER JOIN gold.dim_drivers AS t2 ON t1.sk_driver = t2.sk_driver
INNER JOIN gold.dim_races AS t3 ON t1.sk_races = t3.sk_races
WHERE t2.name = "Ayrton Senna"
AND t1.positionOrder = 1
GROUP BY t2.name,
t3.name
order by count(t1.positionOrder) desc

```

▶ (4) Spark Jobs

▶ _sqldf: pyspark.sql.dataframe.DataFrame = [count(positionOrder): long, grandPrix: string ... 1 more field]

Table ▼ +

	¹ ₂ ³ count(positionOrder)	^A _B ^C grandPrix	^A _B ^C piloto
1	6	Monaco Grand Prix	Ayrton Senna
2	5	Belgian Grand Prix	Ayrton Senna
3	3	San Marino Grand Prix	Ayrton Senna
4	3	Detroit Grand Prix	Ayrton Senna
5	3	Hungarian Grand Prix	Ayrton Senna
6	3	German Grand Prix	Ayrton Senna
7	2	Australian Grand Prix	Ayrton Senna
8	2	Italian Grand Prix	Ayrton Senna
9	2	Canadian Grand Prix	Ayrton Senna
10	2	Spanish Grand Prix	Ayrton Senna
11	2	United States Grand P...	Ayrton Senna
12	2	Japanese Grand Prix	Ayrton Senna
13	2	Brazilian Grand Prix	Ayrton Senna
14	1	British Grand Prix	Ayrton Senna
15	1	Portuguese Grand Prix	Ayrton Senna
16	1	Mexican Grand Prix	Ayrton Senna
17	1	European Grand Prix	Ayrton Senna

3. Discussão dos Resultados e Autoavaliação

3.1 Principais Descobertas

- Comparação de desempenho entre pilotos em diferentes décadas.
- Relação entre pole position e vitória na corrida.
- Evolução dos pilotos dominantes ao longo do tempo.
- Quais são os países que mais se destacam no esporte.
- Países que possuem os pilotos mais agressivos (pela média de velocidade km/h).

3.2 Limitações e Desafios

- Dados incompletos em algumas temporadas antigas e até em temporadas recentes, que são “mascarados” com /N e dificultam o tratamento e limpeza pois ficam mais difíceis de serem encontrados, como no exemplo da pergunta: “Análise da temporada passada: Qual piloto fez a volta mais rápida de cada Grand Prix em 2024?”, somente na parte de resultados descobrimos que alguns valores de tempo estão sem preenchimento correto, e que passaram despercebidos durante todo o ETL.
- Necessidade de normalização de nomes de pilotos e equipes.
- Falta de conhecimento da ferramenta: como nunca tinha tido contato com o Databricks e o tempo era um fator determinante o trabalho teve de ser simplificado ao máximo visando atender os requisitos mínimos, fica como um trabalho futuro adicionar mais recursos que a ferramenta dispõe e até mais tabelas dimensões para enriquecer as consultas e respostas

3.1 Atingimento dos Objetivos

- Quais perguntas foram respondidas?

Mesmo com todas as dificuldades encontradas e com falta de conhecimento de todos os recursos da ferramenta todas as perguntas propostas no objetivo foram respondidas com queries SQL

- Quais ficaram pendentes e por quê?

A única parte de ficou sem solução foi quem fez a volta mais rápida no GranPrix de Mônaco 2024, já que para todos os pilotos a coluna de “fastestLap” estava com “\N”

4. Trabalhos Futuros

- Inclusão de mais fontes de dados (ex.: circuitos, pit stops, construtores, equipes e sprints).
 - Aplicação de machine learning para prever resultados de corridas.
-

5. Conclusão

Este MVP comprovou a viabilidade de um **pipeline de dados na nuvem** seguindo a abordagem **Medallion Architecture**, organizado em três camadas (**Bronze, Silver e Gold**), mesmo que em ferramentas gratuitas para análise histórica da Fórmula 1. A estrutura em **Esquema Estrela** otimizou a modelagem dimensional dos dados, facilitando consultas analíticas e agregações eficientes. Além disso, a implementação de um **Catálogo de Dados** garantiu a governança e a rastreabilidade dos metadados, melhorando a documentação e a usabilidade do pipeline. Os resultados obtidos fornecem **insights valiosos** para entusiastas e analistas do esporte, demonstrando a eficácia da arquitetura proposta. Este trabalho também serve como base para futuras expansões, como a inclusão de novas fontes de dados, análises preditivas ou integração com ferramentas de visualização mais avançadas.

Assim, o projeto não apenas atende aos objetivos iniciais, mas também estabelece um **framework escalável e bem documentado** para o processamento de dados na nuvem