



BIG DATA ANALYTICS SU VOLI AEREI

Corso di Modelli e tecniche per Big Data

Studentessa
Gaia Assunta Bertolino
Matricola 242590



IL DATASET E IL SOFTWARE

Breve descrizione del dataset e del software

Il dataset assegnato presenta una selezione di dati originariamente riportati nel Reporting Carrier On-Time Performance, un database contenente le prestazioni dei voli domestici americani negli anni fra il 1987 e il 2020, contenete un totale di circa 200 milioni di record.

In particolare, nel dataset assegnato sono presenti i dati afferenti ai voli effettuati nel solo anno 2013 e sono suddivisi in 12 file, uno per ciascun mese, per un totale di 6369482 record da 110 campi ciascuno.

Le informazioni presenti riguardano, tra le altre, orari di partenza e arrivo, minuti di ritardo e relative cause, aeroporti intermedi in caso di deviazioni e informazioni sulla cancellazione.

Per effettuare la big data analysis è stato utilizzato principalmente il software Apache Spark e, specificatamente, la libreria PySpark che permette di utilizzare le sue proprietà in linguaggio Python.

Il progetto ha previsto una parte di analisi preliminare per ottenere le coordinate di tutti gli aeroporti citati eseguita non a runtime a causa del tempo impiegato. L'interfaccia utente, invece, esegue ogni altra interazione col dataset in tempo reale. Inoltre, per ottimizzare alcuni utilizzi ripetuti si è ricorso a dei meccanismi di caching in grado di rendere persistente in memoria la struttura dati.

Operazioni preliminari

Per poter operare sul dataset sono state eseguite delle operazioni preliminari:

- Per facilitare l'intero progetto è stato generato un unico dataframe PySpark ottenuto dalla union dei singoli file in estensione csv.

```
# Create SparkSession
spark = SparkSession.builder.master("local[*]").appName("SparkByExamples.com").getOrCreate()
spark.sparkContext.setLogLevel("OFF")

# Reading dataset
df = ((spark.read.options(delimiter=",") .csv(path1, header=True)) \
    .union(spark.read.options(delimiter=",") .csv(path2, header=True)) \
    .union(spark.read.options(delimiter=",") .csv(path3, header=True)) \
    .union(spark.read.options(delimiter=",") .csv(path4, header=True)) \
    .union(spark.read.options(delimiter=",") .csv(path5, header=True)) \
    .union(spark.read.options(delimiter=",") .csv(path6, header=True)) \
    .union(spark.read.options(delimiter=",") .csv(path7, header=True)) \
    .union(spark.read.options(delimiter=",") .csv(path8, header=True)) \
    .union(spark.read.options(delimiter=",") .csv(path9, header=True)) \
    .union(spark.read.options(delimiter=",") .csv(path10, header=True)) \
    .union(spark.read.options(delimiter=",") .csv(path11, header=True)) \
    .union(spark.read.options(delimiter=",") .csv(path12, header=True))) \
    .na.drop(subset=["Flight_Number_Reported_Airline"]).cache()
```

Inoltre, per velocizzare il funzionamento dell'applicazione è richiamata la funzione di cache sul dataframe ottenuto in precedenza. Il suo scopo è di anticipare il calcolo del contenuto rendendolo disponibile in memoria per gli usi futuri. Tale funzione modifica il comportamento tipico della struttura caratterizzata di default da lazy evaluation, principio che prevede che ogni trasformazione operata su un dataframe sia effettivamente computata solo in seguito al richiamo di un'azione per risparmiare uso di risorse nei momenti in cui non vi è effettivo bisogno del suo uso.

- Alcuni campi sono stati opportunamente trasformati dal tipo di default stringa affinché fossero disponibili sottoforma di stringa oraria, come nel caso dei campi ArrTime e DepTime (rappresentanti rispettivamente orario effettivo di arrivo e partenza), sottoforma di data, come nel caso del campo FlightDate contenente la data di volo, o sottoforma di intero, come nel caso dei campi ArrDelayMinutes o CarriedDelay contenenti dei valori rappresentativi del ritardo del volo. In particolare, la rappresentazione delle stringhe contenti orari e date sottoforma di tipi specifici risulta essere utile per poter operare dei confronti in maniera automatica.

```

# Cast to integer
df = df.withColumn("ArrDelayMinutes", df["ArrDelayMinutes"].cast(IntegerType()))
df = df.withColumn("CarrierDelay", df["CarrierDelay"].cast(IntegerType()))
df = df.withColumn("WeatherDelay", df["WeatherDelay"].cast(IntegerType()))
df = df.withColumn("NASDelay", df["NASDelay"].cast(IntegerType()))
df = df.withColumn("SecurityDelay", df["SecurityDelay"].cast(IntegerType()))
df = df.withColumn("LateAircraftDelay", df["LateAircraftDelay"].cast(IntegerType()))
df = df.withColumn("Distance", df["Distance"].cast(IntegerType()))

# Cast to time
df = df.withColumn('DepTime', to_timestamp("DepTime", "HHmm"))
df = df.withColumn("ArrTime", to_timestamp('ArrTime', 'HHmm'))
df = df.withColumn('CRSArrTime', to_timestamp("CRSArrTime", "HHmm"))
df = df.withColumn("CRSDepTime", to_timestamp('CRSDepTime', 'HHmm'))
df = df.withColumn("DivArrDelay", to_timestamp('DivArrDelay', 'HHmm'))
df = df.withColumn("FlightDate", to_date(col("FlightDate"), "yyy-MM-dd"))

```



L'interfaccia utente è stata realizzata attraverso il framework open-source Streamlit che mette a disposizione componenti in grado di rendere fluida e user-friendly l'interazione fra l'utente e il programma.

Informazioni sui voli in America

Anno 2013

Scegli una pagina da visualizzare

- Homepage
- Informazioni sui voli mensili
- Informazioni sui ritardi
- Informazioni sui dirottamenti
- Ricerca voli

La navigazione fra le pagine avviene attraverso un menu presente nella parte sinistra realizzato attraverso un oggetto Streamlit **radiobutton** che assegna alla variabile **menu** il titolo della finestra da visualizzare.

Il progetto prevede 5 finestre:

- Homepage
- Informazioni mensili
- Informazioni sui ritardi
- Informazioni sui dirottamenti
- Ricerca dei voli da parte dell'utente

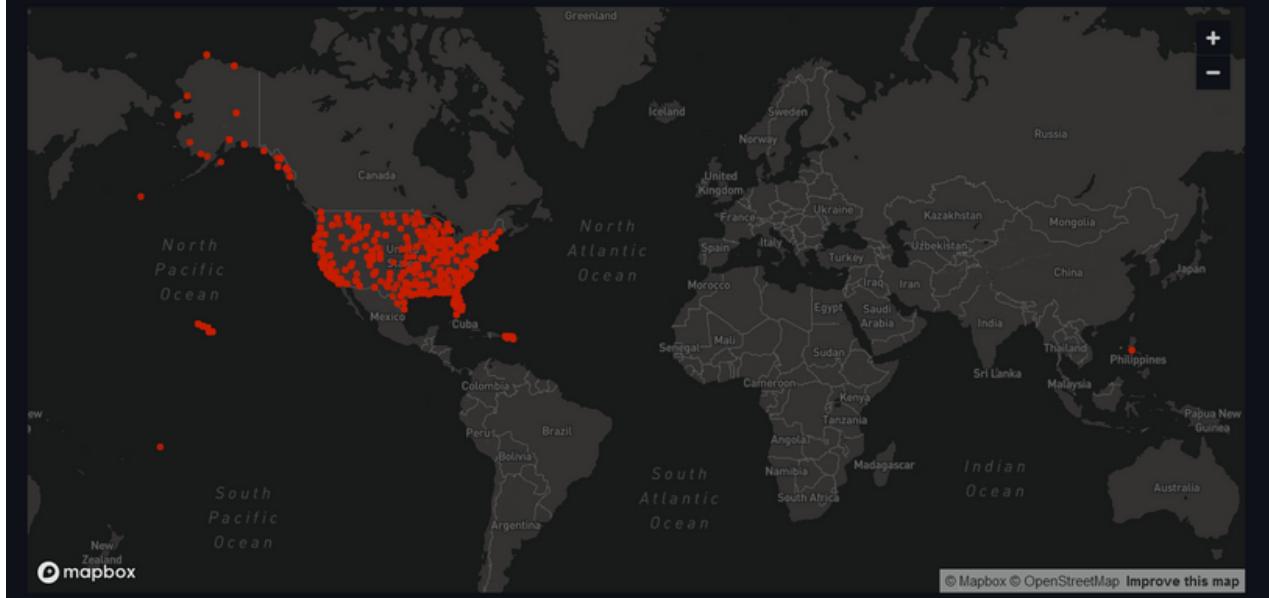
```
# User interface
streamlit.set_page_config(page_title='Analisi dei voli', layout="wide")
streamlit.sidebar.image(image=Image.open(image), width=300)
streamlit.sidebar.header('Informazioni sui voli in America' + '\n' + 'Anno 2013')
menu = streamlit.sidebar.radio("Scegli una pagina da visualizzare", ('Homepage', 'Informazioni sui voli mensili',
) 'Informazioni sui ritardi', 'Informazioni sui dirottamenti', 'Ricerca voli'))
```

Homepage

Il dataset comprende i dati dei voli domestici americani eseguiti nell'arco del 2013. E' riportata la mappa di tutti gli aeroporti. Tra i dati riportati sono presenti la tratta più trafficata e il veivolo che ha compiuto più voli.

Mappa degli aeroporti

Ogni punto sulla mappa rappresenta un aeroporto da cui è partito, atterrato o transitato un aereo



Finestra Homepage

La prima riga della finestra di homepage presenta una mappa generata tramite un oggetto Streamlit map. Esso prende in input un dataframe Pandas contenente le coordinate degli elementi da visualizzare e le rappresenta sottoforma di uno scatter plot.

```
# Data for the map
dfTem = pandas.read_csv(pathCoor, delimiter=",")
dfMap1 = pandas.DataFrame()
dfMap1 = dfTem[['OriginLat', 'OriginLon']]
dfMap1.rename(columns={'OriginLat': 'LAT', 'OriginLon': 'LON'}, inplace=True)
dfMap2 = pandas.DataFrame()
dfMap2 = dfTem[['DestinationLat', 'DestinationLon']]
dfMap2.rename(columns={'DestinationLat': 'LAT', 'DestinationLon': 'LON'}, inplace=True)
dfMap = pandas.concat([dfMap1, dfMap2])

# First row - map
streamlit.markdown(":red[Mappa degli aeroporti]")
streamlit.markdown("Ogni punto sulla mappa rappresenta un aeroporto da cui è partito, atterrato o transitato un aereo")

streamlit.map(data=dfMap, zoom=1)
streamlit.markdown("----")
```

Come anticipato, il file contenente le coordinate è stato generato preliminarmente attraverso un'apposita funzione routes.

Per ogni riga del dataset, la funzione *routes* ricerca le coordinate dell'aeroporto di partenza e di quello di arrivo attraverso il tool Nominatim, il quale usa i dati di OpenStreetMap per individuare delle località attraverso il nome. In particolare, viene operata una ricerca per aeroporto e, se il server non è in grado di individuarlo e restituisce un valore nullo, viene effettuata una seconda ricerca in base al solo nome della relativa città.

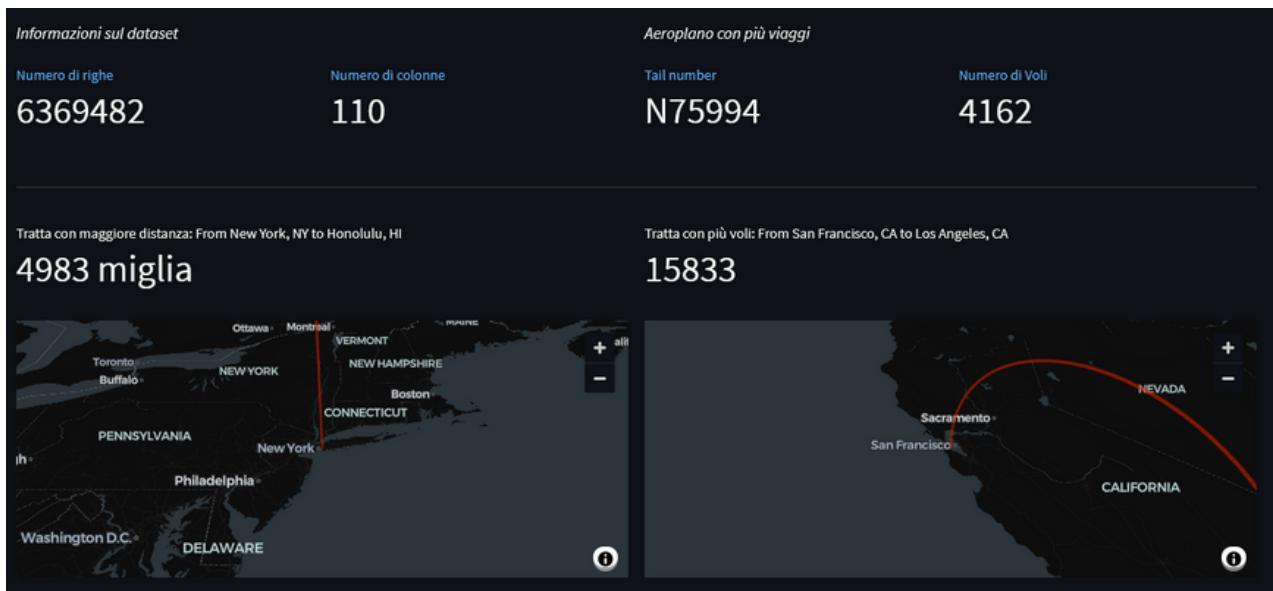
```
# Prints the coordinates of all airports
def routes():
    locator = Nominatim(user_agent="myNewGeocoder")
    df2 = df.dropDuplicates(["Origin", "Dest"]).collect()

    for row in df2:
        # Coordinates for Origin
        time.sleep(1)
        found = locator.geocode(row["Origin"] + " Airport " + row["OriginCityName"], timeout=None)
        if found is None:
            time.sleep(1)
            found = locator.geocode(row["OriginCityName"], timeout=None)
        locationOr = found

        # Coordinates for destination
        time.sleep(1)
        found = locator.geocode(row["Dest"] + " Airport " + row["DestCityName"], timeout=None)
        if found is None:
            time.sleep(1)
            found = locator.geocode(row["DestCityName"], timeout=None)
        locationDes = found

        # Printing of coordinates
        if locationOr is not None and locationDes is not None:
            print(locationOr.latitude + "," + locationOr.longitude + "," + locationDes.latitude + "," + locationDes.longitude)
```

Il server utilizzato fissa un limite al numero di richieste che è possibile effettuare al secondo e ciò ha richiesto l'inserimento di opportune sleep fra le varie chiamate della funzione *routes*. Di conseguenza, la funzione riesce a completare la sua esecuzione dopo diverso tempo; da qui la necessità di eseguire l'azione non in runtime.



La seconda riga presenta, invece, quattro colonne:

- Le due colonne più a sinistra riportano le informazioni relative alla dimensione del dataset ovvero il numero di righe e di colonne
- Le due colonne più a sinistra riportano le informazioni relative l'aereo che ha compiuto più voli ovvero il suo numero identificativo e il numero di voli compiuti nel 2013

```
col1, col2, col3, col4 = streamlit.columns(4)
with col1:
    streamlit.markdown('_Informazioni sul dataset_')
    streamlit.metric(label=":blue[Numero di righe] ", value=dimensionRows())
    streamlit.metric(label=":blue[Numero di colonne] ", value=dimensionCol())

dfTailNumber = mostTravelsAirplane()
with col2:
    streamlit.markdown('_Aeroplano con più viaggi_')
    streamlit.metric(label=":blue[Tail number] ", value=dfTailNumber.take(num=1)[0].__getitem__("Tail_Number"))
    streamlit.metric(label=":blue[Numero di Voli] ", value=dfTailNumber.take(num=1)[0].__getitem__("count"))
```

La funzione `mostTravelsAirplane` ha lo scopo di calcolare quale sia l'aereo che ha compiuto più viaggi. Per fare ciò è utilizzata la funzione `orderBy` che permette di compiere un ordinamento decrescente in base ai valori di una colonna passata come argomento.

```
# Number of rows and columns of dataset
def dimensionRows():
    return str(df.count())
def dimensionCol():
    return str(len(df.columns))

# Airplane which made most flights
def mostTravelsAirplane():
    return df.na.drop(subset=["Tail_Number"]).groupby("Tail_Number").count().orderBy(desc("count"))
```

La terza riga risulta essere invece divisa in due colonne:

- La colonna più a sinistra riporta le informazioni relative alla tratta caratterizzata da una maggiore distanza
- La colonna più a destra riporta le informazioni relative alla tratta che è stata percorsa da più voli

Per ciascuna delle due informazioni è riportata anche la rappresentazione su mappa realizzata attraverso un oggetto PyDeck. Tra i vari tipi di layer realizzabili, è stato utilizzato il tipo ArcLayer che permette di rappresentare un arco fra due coppie di coordinate.

```
# Third row - maps
col5, col6 = streamlit.columns(2)
with col5:
    travelD = biggestDistance().take(num=1)[0]
    lab1 = 'Tratta con maggiore distanza:' + " From " + travelD.__getitem__(
        "OriginCityName") + " to " + travelD.__getitem__("DestCityName")
    streamlit.metric(label=lab1, value=str(travelD.__getitem__("max(Distance)")) + " miglia")
    coorD = coordinates(travelD.__getitem__("Origin"), travelD.__getitem__("OriginCityName"),
                         travelD.__getitem__("Dest"), travelD.__getitem__("DestCityName"))

    col5.pydeck_chart(
        pydeck.Deck(
            # map_style=None,
            initial_view_state=pydeck.ViewState(
                latitude=coorD.iloc[0].__getitem__("OriginLatitude"),
                longitude=coorD.iloc[0].__getitem__("OriginLongitude"),
                zoom=5,
                pitch=50,
                height=250,
            ),
            layers=[
                pydeck.Layer(
                    "ArcLayer",
                    data=coorD,
                    get_source_position=["OriginLongitude", "OriginLatitude"],
                    get_target_position=["DesLongitude", "DesLatitude"],
                    get_source_color=[200, 30, 0, 160],
                    get_target_color=[200, 30, 0, 160],
                    auto_highlight=True,
                    width_scale=0.0001,
                    get_width="outbound",
                    width_min_pixels=3,
                    width_max_pixels=30,
                    radius=200,
                    elevation_scale=4,
                    elevation_range=[0, 1000],
                    pickable=True,
                    extruded=True,
                ),
            ],
        )
    )
```

Per ottenere le coordinate di una specifica coppia di aeroporti origine-destinazione è stata realizzata una funzione simile alla funzione *routes* precedentemente vista dal nome *coordinates* che ne sfrutta lo stesso meccanismo.

```
# Returns the coordinates for a given airport
def coordinates(origin, originCityName, destination, destinationCityName):
    locator = Nominatim(user_agent="myNewGeocoder")

    # Coordinates for Origin
    found = locator.geocode(origin + " Airport " + originCityName, timeout=None)
    if found is None:
        time.sleep(1)
        found = locator.geocode(originCityName, timeout=None)
    result = [found.latitude, found.longitude]

    # Coordinates for destination
    time.sleep(1)
    found = locator.geocode(destination + " Airport " + destinationCityName, timeout=None)
    if found is None:
        time.sleep(1)
        found = locator.geocode(destinationCityName, timeout=None)
    result.append(found.latitude)
    result.append(found.longitude)
    dfRes = pandas.DataFrame({'OriginLatitude': [result[0]], 'OriginLongitude': [result[1]],
                               'DesLatitude': [result[2]], 'DesLongitude': [result[3]]})
    return dfRes
```

Il codice utilizzato per la realizzazione delle mappe è pressoché similare; ciò che varia è la funzione utilizzata per filtrare il dataframe:

- Nel caso della rotta con distanza maggiore si è realizzata la funzione *biggestDistance* che fa uso della funzione *max* per ottenere la riga contenente il massimo valore della colonna Distance, assumendo che per una stessa tratta possano esserci delle variazioni nella distanza percorsa

```
# Route with the biggest distance
def biggestDistance():
    return df.groupby("Origin", "Dest", "OriginCityName", "DestCityName").max("Distance").orderBy(desc("max(Distance)"))
```

- Nel caso della rotta con più voli è stata utilizzata una funzione *mostTravels* che fa uso della funzione *count* per contare le righe di ciascuna tratta, individuate dal raggruppamento per città.

```
# Route with more flights
def mostTravels():
    return df.groupby("Origin", "Dest", "OriginCityName", "DestCityName").count().orderBy(desc("count"))
```

Dati sui voli mensili

Per ciascun mese è riportato il numero di voli eseguiti con l'incremento rispetto al precedente. In blu è evidenziato il mese con più voli di tutto l'anno 2013

Gennaio	Febbraio	Marzo	Aprile
509519	469746	552312	536393
↑ 0	↓ -39773	↑ 82566	↓ -15919
Maggio	Giugno	Luglio Mese con più voli	Agosto
548642	552141	571623	562921
↑ 12249	↑ 3499	↑ 19482	↓ -8702
Settembre	Ottobre	Novembre	Dicembre
510806	535344	503296	516739
↓ -52115	↑ 24538	↓ -32048	↑ 13443

Finestra Informazioni sui voli mensili

La seconda finestra riporta delle informazioni basate sui dati mensili. Per ciascuno sono indicati:

- Il numero di voli eseguiti
- L'incremento nel numero di voli rispetto al precedente.

Per calcolare il numero di voli è utilizzata una funzione *countFlightsMonth* che raggruppa il contenuto del dataframe in base ai valori presenti nella colonna *month* e per ciascuno restituisce il conteggio delle colonne. Dunque vengono tenuti in considerazione anche i voli cancellati e i voli dirottati.

La funzione è definita come segue:

```
# Monthly count of flights
def countFlightsMonth():
    return df.groupby("month").count()
```

Per rappresentare i dati attraverso un ciclo di for è stata resa iterabile la struttura dati ricorrendo alla funzione *collect* che trasforma il dataframe in un array di oggetti di tipo Row

```
# Data
monthsFlightDF = countFlightsMonth()
monthMostFlights = int(monthsFlightDF.orderBy(desc("count")).take(num=1)[0].__getitem__("month")) - 1
monthsFlight = monthsFlightDF.collect()
```

Il ciclo di for genera quattro colonne su tre righe e posiziona all'interno di ogni cella ottenuta un oggetto Streamlit di tipo *metric*. Esso è dotato di una **label**, un **valore** da mostrare (in questo caso il mese) e un **delta** (che rappresenta in questo l'incremento di voli). In particolare, l'oggetto è stato customizzato affinché un aumento dei voli venisse rappresentato in automatico con una freccia verso l'alto e in colore **verde** mentre un decremento (dunque con segno negativo) con una freccia **rossa** verso il basso. Il mese con più voli dell'anno prevede che la label sia seguita da una dicitura in **blu** "Mese con più voli".

```
# Stream
prev = 0
for i in range(0, 12, 4):
    col1, col2, col3, col4 = streamlit.columns(4)

    val = monthsFlight[i].__getitem__("count")
    lab = findMonth(i)
    if prev == 0:
        delta = 0
    else:
        delta = val-prev
    if i == 0:
        deltaCol = "off"
    else:
        deltaCol = "normal"
    if i == monthMostFlights:
        col1.metric(label=lab + ":blue[ Mese con più voli ] ", value=val, delta=delta, delta_color=deltaCol)
    else:
        col1.metric(label=lab, value=val, delta=delta, delta_color="normal")
    prev = val

    val = monthsFlight[i+1].__getitem__("count")
    lab = findMonth(i+1)
    delta = val - prev
    if (i+1) == monthMostFlights:
        col2.metric(label=lab + ":blue[ Mese con più voli ] ", value=val, delta=delta, delta_color="normal")
    else:
        col2.metric(label=lab, value=val, delta=delta, delta_color="normal")
    prev = val

    val = monthsFlight[i+2].__getitem__("count")
    lab = findMonth(i+2)
    delta = val - prev
    if (i+2) == monthMostFlights:
        col3.metric(label=lab + ":blue[ Mese con più voli ] ", value=val, delta=delta, delta_color="normal")
    else:
        col3.metric(label=lab, value=val, delta=delta, delta_color="normal")
    prev = val

    val = monthsFlight[i+3].__getitem__("count")
    lab = findMonth(i + 3)
    delta = val - prev
    if (i+3) == monthMostFlights:
        col4.metric(label=lab + ":blue[ Mese con più voli ] ", value=val, delta=delta, delta_color="normal")
    else:
        col4.metric(label=lab, value=val, delta=delta, delta_color="normal")
    prev = val
```

Informazioni sui ritardi

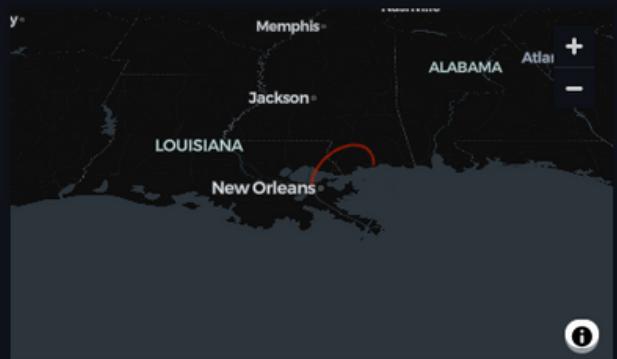
Sono riportate le informazioni inerenti la tratta con ritardo medio maggiore e le motivazioni che causano i ritardi. Puoi visualizzare la tratta con la percentuale di ritardi maggiore e che compie un numero minimo di voli indicando il valore attraverso lo slider

Tratta aerea con ritardo medio maggiore:

Volo da New Orleans, LA a Gulfport/Biloxi, MS

From MSY To GPT

315.0 Minutes



Finestra Informazioni sui ritardi

La terza finestra riporta alcune informazioni ottenute dai dati sui ritardi riportati nel dataset.

La prima riga presenta la tratta aerea con ritardo medio maggiore e la relativa mappa ottenuta col costrutto PyDeck ArcLayer visto in precedenza.

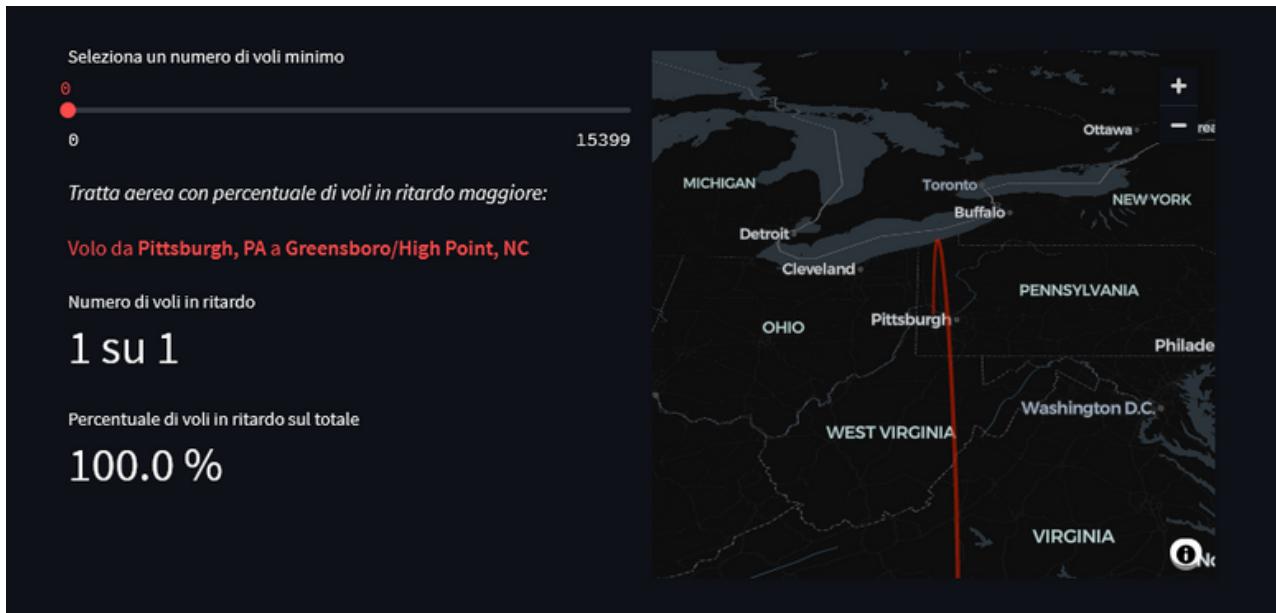
```
col1, col2 = streamlit.columns(2)

# Route with the most average delay
dfAverageDelay = routesAverageDelay().take(num=1)[0]
col1.markdown('_Tratta aerea con ritardo medio maggiore:_')
col1.markdown(":red[Volo da **" + dfAverageDelay.__getitem__("OriginCityName") + "** a **" +
    dfAverageDelay.__getitem__("DestCityName") + "**]")
col1.metric(label="From " + dfAverageDelay.__getitem__("Origin") + " To " + dfAverageDelay.__getitem__("Dest"),
    value=str(dfAverageDelay.__getitem__("Delay")) + " Minutes")

coor1 = coordinates(dfAverageDelay.__getitem__("Origin"), dfAverageDelay.__getitem__("OriginCityName"),
    dfAverageDelay.__getitem__("Dest"), dfAverageDelay.__getitem__("DestCityName"))
```

Per ottenere la tratta, è stata realizzata una funzione `routesAverageDelay` che fa uso della funzione `avg` in grado di calcolare la media dei valori presenti nella colonna passata come argomento. La colonna ottenuta è stata poi rinominata con la funzione `alias` per un più facile accesso.

```
# Routes and their average delay
def routesAverageDelay():
    return df.groupby("Origin", "Dest", "OriginCityName", "DestCityName").agg(avg("ArrDelay").alias("Delay")).orderBy(desc("Delay"))
```



La seconda riga presenta un oggetto **slider** che rende interattiva la visualizzazione. In particolare, tra le tratte che hanno previsto almeno tanti voli quanto è il valore indicato dallo slider, viene riportata quella con la percentuale maggiore di ritardi sul totale dei voli compiuti.

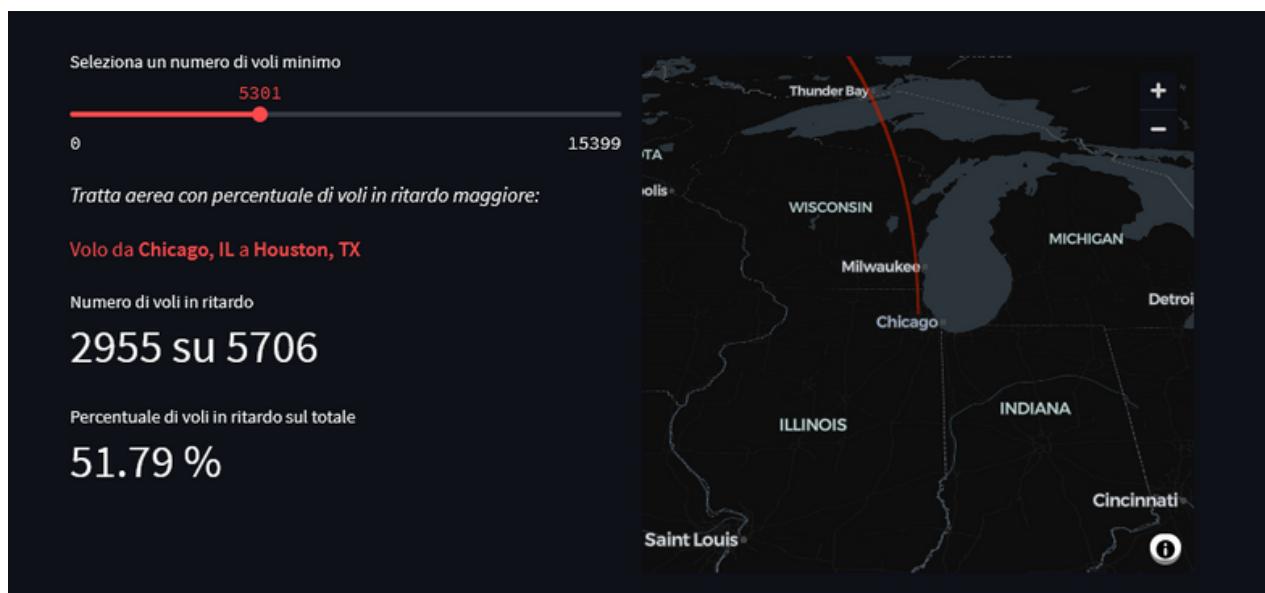
```
# Route with the biggest number of delay
dfNumberDelayTotal = routesNumberDelay()
maxVal = dfNumberDelayTotal.select(max("Total")).take(num=1)[0].__getitem__("max(Total)")
number = col3.slider('Selezione un numero di voli minimo', min_value=0, max_value=maxVal, step=1)
dfNumberDelay = dfNumberDelayTotal.filter(col("Total") >= number).take(num=1)[0]
col3.markdown('_Tratta aerea con percentuale di voli in ritardo maggiore:_')
col3.markdown(
    ":red[Volo da **" + dfNumberDelay.__getitem__("OriginCityName") + "** a **" + dfNumberDelay.__getitem__(
        "DestCityName") + "**]")
col3.metric(label="Numero di voli in ritardo",
            value=str(str(dfNumberDelay.__getitem__("Delay")) + " su " + str(dfNumberDelay.__getitem__("Total"))))
col3.metric(label="Percentuale di voli in ritardo sul totale",
            value=str(dfNumberDelay.__getitem__("Percentage")) + "%")

coor2 = coordinates(dfNumberDelay.__getitem__("Origin"), dfNumberDelay.__getitem__("OriginCityName"),
                     dfNumberDelay.__getitem__("Dest"), dfNumberDelay.__getitem__("DestCityName"))
```

Per ottenere i dati della tratta è stata realizzata una funzione *routesNumberDelay* che fa uso della funzione *agg* per giustapporre le colonne relative ai ritardi (filtrate per tenere in considerazione solo le celle con valori maggiori di zero) al totale dei voli e alla percentuale. Il dataframe ottenuto è poi ordinato in maniera decrescente affinché sulla prima riga si trovino le informazioni ricercate.

```
# Routes and number of delay related to the total amount
def routesNumberDelay():
    return df.groupby("Origin", "Dest", "OriginCityName", "DestCityName").agg(count(when(col("ArrDelayMinutes") > 0, 1)).alias("Delay"),
    count(col("ArrDelayMinutes")).alias("Total"),
    rou((count(when(col("ArrDelayMinutes") > 0, 1)) / (count(col("ArrDelayMinutes")) * 100), 2).alias("Percentage")).orderBy(desc("Percentage")))
```

Il valore iniziale dello slider è impostato su zero. Ad esempio, con un valore dello slider pari a 5301 viene visualizzato la tratta di volo da Chicago a Houston, la quale ha una percentuale di ritardo pari all'51,79% e ha visto un numero di voli totali pari a 5706 nell'arco del 2013.

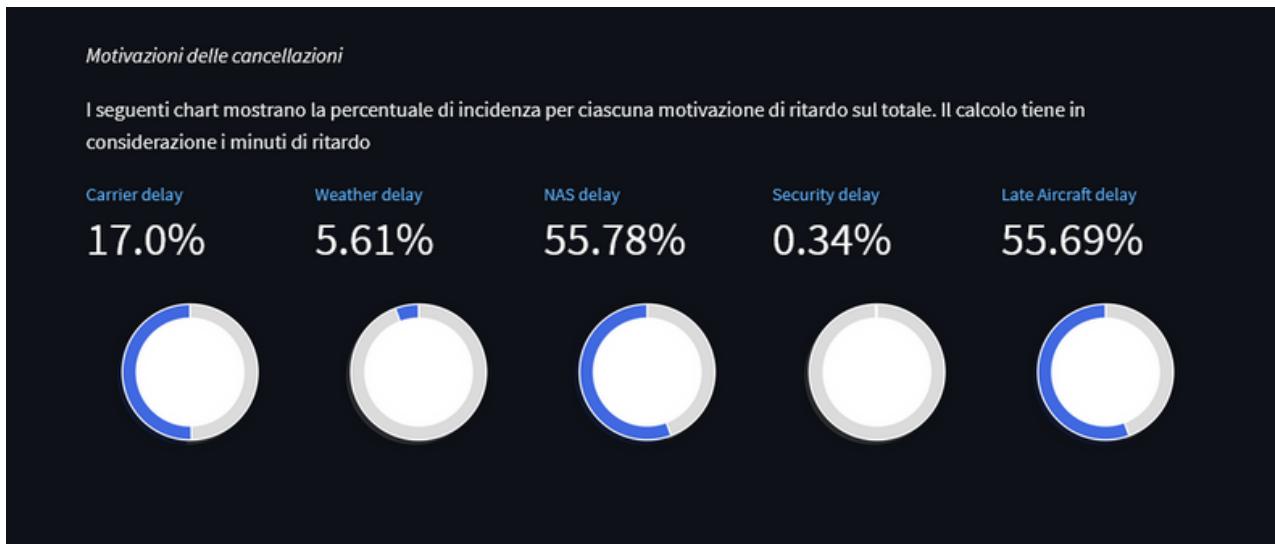


Infine, l'ultima riga della finestra riporta dei chart ottenuti dall'elaborazione dei dati sui ritardi.

In particolare, il dataset presenta cinque colonne relative ai minuti di ritardo:

- Carrier delay
- Weather delay
- NAS delay
- Security delay
- Late aircraft delay

Da una analisi totale, tali colonne presentano dei valori non nulli, quindi uguali o maggiori di zero, solo nel caso in cui vi sia stato un ritardo nel volo di riferimento e, sempre in tal caso, il valore in esse contenuto rappresenta il numero di minuti di ritardo causati da ciascun motivo.



Per la rappresentazione grafica sono state utilizzate cinque colonne ciascuna riportante un oggetto `metric`, per riportare la percentuale calcolata, e un oggetto `pie`, utilizzato per rappresentare i dati sottoforma di pie chart.

A titolo esemplificativo, viene riportato il codice della prima colonna, equivalente alle altre quattro presenti.

```
col5, col6, col7, col8, col9 = streamlit.columns(5)
dfDelay = reasonsDelayAverage().collect()

# Carrier delay
with col5:
    streamlit.metric(label=":blue[Carrier delay]", value= str(dfDelay[0][0]) + "%")
    value = dfDelay[0][1]
    sizes = [value, (100-value)]
    colors = ['#4169E1', '#DCDCDC']
    fig, ax = plt.subplots()
    ax.pie(sizes, shadow=True, startangle=90, labeldistance=1.15,
           wedgeprops = { 'linewidth' : 3, 'edgecolor' : 'white' }, colors=colors)
    ax.axis('equal') # Equal aspect ratio ensures that pie is drawn as a circle.
    p = plt.gcf()
    p.gca().add_artist(plt.Circle((0, 0), 0.8, color='white'))
    fig.patch.set_facecolor('#0e1117')
    streamlit.pyplot(fig)
```

Per ottenere i dati è stata realizzata una funzione *reasonsAverageDelay* che fa uso della funzione *round* rinominata *rou* per approssimare il valore della percentuale ottenuto.

Per accedere più agevolmente al contenuto del dataframe, esso è stato poi convertito in una collezione attraverso la funzione *collect* che ne permette l'iterazione.

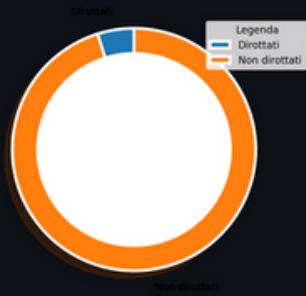
```
# Motive that caused more delay and the average time each time there was a delay
# NO DIVERTED FLIGHTS
def reasonsDelayAverage():
    total = df.where(col("CarrierDelay") >= 0).count()
    result = df.groupby().agg(rou(avg("CarrierDelay")).alias("Carrier delay"),
        rou((count(when(col("CarrierDelay") > 0, 1)) / total * 100), 2).alias("Percentage"),
        rou(avg("WeatherDelay")).alias("Weather delay"),
        rou((count(when(col("WeatherDelay") > 0, 1)) / total * 100), 2).alias("Percentage"),
        rou(avg("NASDelay")).alias("NAS delay"),
        rou((count(when(col("NASDelay") > 0, 1)) / total * 100), 2).alias("Percentage"),
        rou(avg("SecurityDelay")).alias("Security delay"),
        rou((count(when(col("SecurityDelay") > 0, 1)) / total * 100), 2).alias("Percentage"),
        rou(avg("LateAircraftDelay")).alias("Late Aircraft delay"),
        rou((count(when(col("LateAircraftDelay") > 0, 1)) / total * 100), 2).alias("Percentage"))
    )
    return result
```

Dirottamenti

Nella pagina seguente è possibile visualizzare alcune informazioni relativamente alle deviazioni subite nelle tratte aeree. Inoltre, sono riportate le informazioni relative alla tratta aerea con più dirottamenti in valore assoluto.

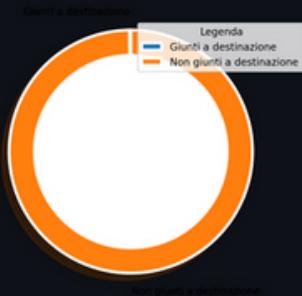
Tratta aerea con il maggior numero di voli dirottati:

83 voli su 1714

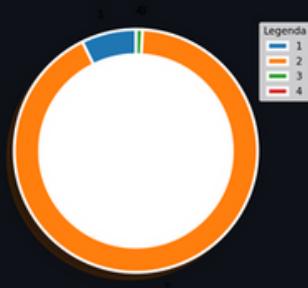


Numero di voli dirottati giunti a destinazione:

11897 su 6369482



Numero di atterraggi deviati:



Volo da Salt Lake City, UT a Sun Valley/Hailey/Ketchum, ID

Finestra *Informazioni sui dirottamenti*

La quarta finestra riporta alcuni dati sulle deviazioni avvenute.

In particolare, le informazioni sono suddivise su tre colonne:

- La prima riporta la tratta aerea con il maggior numero in assoluto di voli deviati
- La seconda riporta il numero di voli dirottati che sono effettivamente giunti a destinazione
- La terza riporta il rapporto sul totale del numero di deviazioni di ogni volo

Per ciascuna delle informazioni è riportato un pie chart.

In particolare, per la prima colonna è stata utilizzata una funzione `routesNumberDiverted` in grado di quantificare per ogni rotta il numero di voli dirottati, la loro percentuale sul totale e il raggiungimento della destinazione per individuare la tratta con numero maggiore di voli deviati

```
# Routes and number of diverted flights
def routesNumberDiverted():
    return df.groupby("Origin", "Dest", "OriginCityName", "DestCityName").agg(count(when(col("Diverted") == "1.00", 1)).alias("Diverted"),
    count(col("Diverted")).alias("Total"), count(col("DivReachedDest") == "1.00").alias("DivReachedDest"),
    rou((count(when(col("Diverted") == "1.00", 1)) / (count(col("Diverted")) * 100), 2).alias("Percentage"))
```

Per la seconda è stata utilizzata una funzione *routesNumberReached* in grado di fornire il conteggio dei voli dirottati e rispettivamente dei voli giunti a destinazione o meno senza raggruppamenti per città.

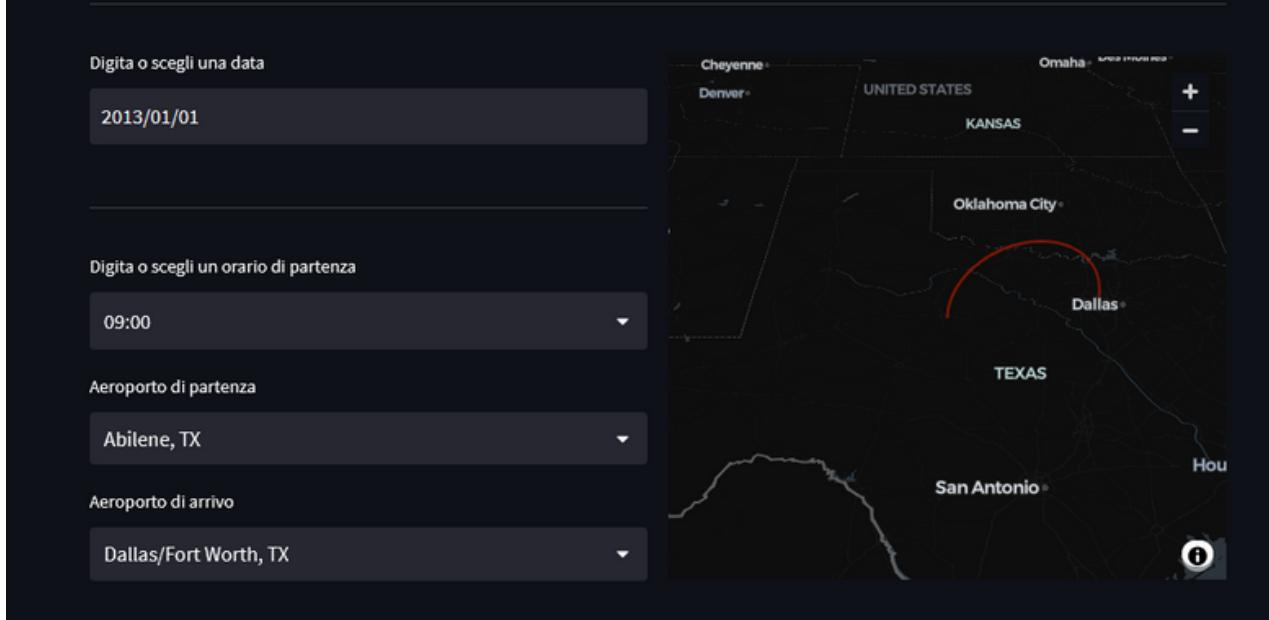
```
# Routes with deverted flights which reached their destination
def routesNumberReached():
    return df.agg(count(when(col("Diverted") == "1.00", 1)).alias("Diverted"),
    count(col("Diverted")).alias("Total"),
    count(when(col("DivReachedDest") == "1.00", 1)).alias("DivReachedDest"))
```

Per la terza è stata utilizzata una funzione *deviatedLandingNumber* che tiene in considerazione soltanto le righe per cui è riportata l'informazione sul numero di aeroporti intermedi e, sulla base dei suoi valori, fornisce un raggruppamento del dataframe.

```
# Number of diverted airport landings
def deviatedLandingsNumber():
    return df.filter(col("DivAirportLandings") != 0).groupby("DivAirportLandings").count()
```

Ricerca voli

Puoi ricercare un volo inserendo la data, l'ora di partenza, l'aeroporto di partenza e quello di arrivo. Ti verranno visualizzate le informazioni relative all'orario di partenza ed arrivo effettivo, la distanza e relativamente alla tratta anche il ritardo medio e la percentuale di voli cancellati



Finestra Ricerca voli

La quinta ed ultima finestra permette all'utente di ricercare un volo inserendo data e ora di partenza e aeroporto di origine e destinazione, restituendo alcune informazioni rilevanti.

```
col1, col2 = streamlit.columns(2)
with col1:
    giorno = streamlit.date_input("Digita o scegli una data", datetime.date(2013, 1, 1))
    streamlit.markdown("----")
    partenza = streamlit.time_input('Digita o scegli un orario di partenza', datetime.time(9, 00))

    p = datetime.datetime.strptime("1970-01-01 " + partenza.strftime("%H:%M:%S"), "%Y-%m-%d %H:%M:%S")
    flightsP = df.filter(col("FlightDate") == giorno).filter(col("CRSDepTime") == p).cache()
    labelsP = flightsP.select(col("OriginCityName")).dropDuplicates().sort(col("OriginCityName"))

    if labelsP.count() == 0:
        streamlit.markdown('Non sono presenti voli per i dati inseriti')
        streamlit.markdown('Attenzione: Scegli un volo del 2013')
    else:
        origin = streamlit.selectbox('Aeroporto di partenza', (labelsP))
        flightsA = flightsP.filter(col("OriginCityName") == origin)
        labelsA = flightsA.select(col("DestCityName")).sort(col("DestCityName"))
        destination = streamlit.selectbox('Aeroporto di arrivo', (labelsA))
        flight = flightsA.filter(col("DestCityName") == destination).take(num=1)[0]
        coor = coordinates(flight.__getitem__("Origin"), flight.__getitem__("OriginCityName"),
                           flight.__getitem__("Dest"), flight.__getitem__("DestCityName"))
```

L'inserimento dei dati avviene attraverso degli oggetti Streamlit di tipo `date_input` per la data, `time_input` per l'orario e `selectbox` per gli aeroporti.

Ogni modifica genera un nuovo filtraggio del dataframe e di conseguenza, per velocizzare le operazioni, è invocata la funzione di `caching` sul dataframe di base che contiene i voli filtrati per data. Da ciò deriva una notevole velocizzazione delle prestazioni.

```
col1, col2 = streamlit.columns(2)
with col1:
    giorno = streamlit.date_input("Digita o scegli una data", datetime.date(2013, 1, 1))
    streamlit.markdown("""---""")
    partenza = streamlit.time_input('Digita o scegli un orario di partenza', datetime.time(9, 00))

    p = datetime.datetime.strptime("1970-01-01 " + partenza.strftime("%H:%M:%S"), "%Y-%m-%d %H:%M:%S")
    flightsP = df.filter(col("FlightDate") == giorno).filter(col("CRSDepTime") == p).cache()
    labelsP = flightsP.select(col("OriginCityName")).dropDuplicates().sort(col("OriginCityName"))
    if labelsP.count() == 0:
        streamlit.markdown('Non sono presenti voli per i dati inseriti')
        streamlit.markdown('Attenzione: Scegli un volo del 2013')
    else:
        origin = streamlit.selectbox('Aeroporto di partenza', (labelsP))
        flightsA = flightsP.filter(col("OriginCityName") == origin)
        labelsA = flightsA.select(col("DestCityName")).sort(col("DestCityName"))
        destination = streamlit.selectbox('Aeroporto di arrivo', (labelsA))
        flight = flightsA.filter(col("DestCityName") == destination).take(num=1)[0]
        coor = coordinates(flight.__getitem__("Origin"), flight.__getitem__("OriginCityName"),
                           flight.__getitem__("Dest"), flight.__getitem__("DestCityName"))
```

La rimanente parte della pagina viene popolata in maniera differente in base all'esito della ricerca, richiamando parzialmente i costrutti precedentemente utilizzati:

- Nessun volo presente per data e orario inseriti: viene visualizzato un warning.

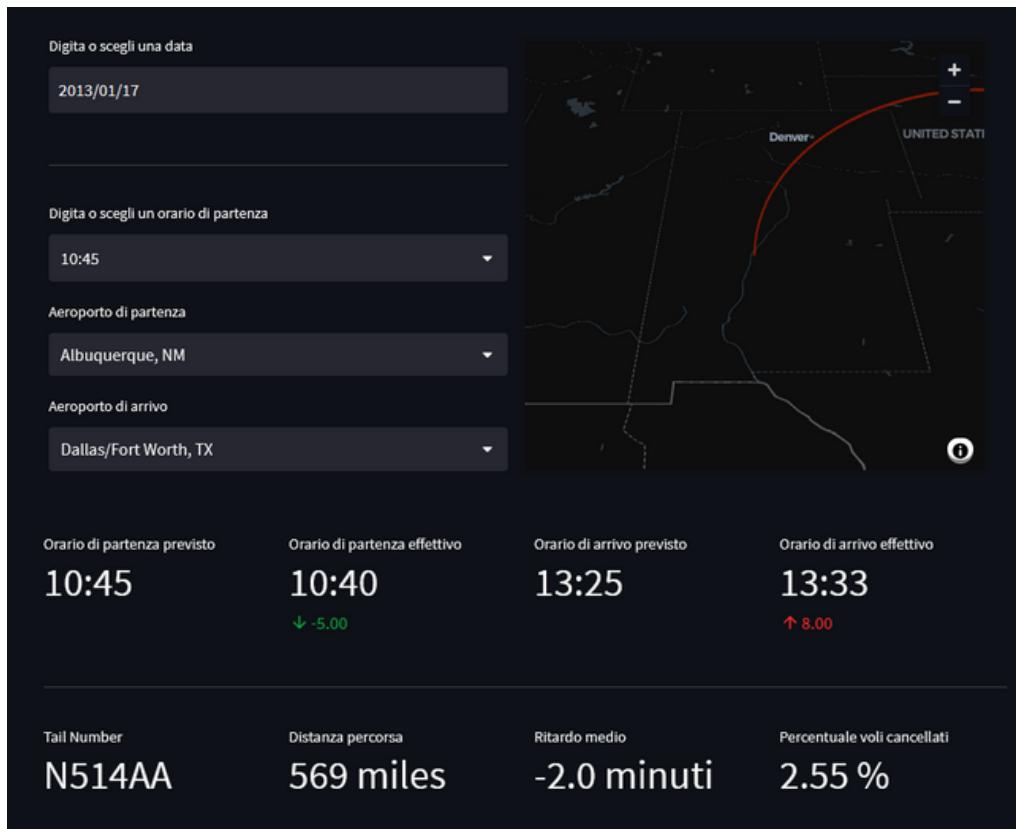
Ad esempio, viene riportato l'esito dell'inserimento della data del 13 dicembre 2012



- Volo generico che è arrivato a destinazione: sono riportati gli orari di partenza previsti ed effettivi e, nel caso in cui l'aereo sia partito e/o giunto in anticipo e/o ritardo, sono riportati anche i relativi minuti di differenza.

Ad esempio, segue la visualizzazione del volo del 17 gennaio 2013 delle 10:45 diretto da Albuquerque a Dallas partito con 5 minuti di anticipo ma atterrato con 8 minuti di ritardo.

In media, la tratta vede un anticipo nell'orario di arrivo pari a 2 minuti e un numero di voli cancellati pari a quasi 3 ogni 100.



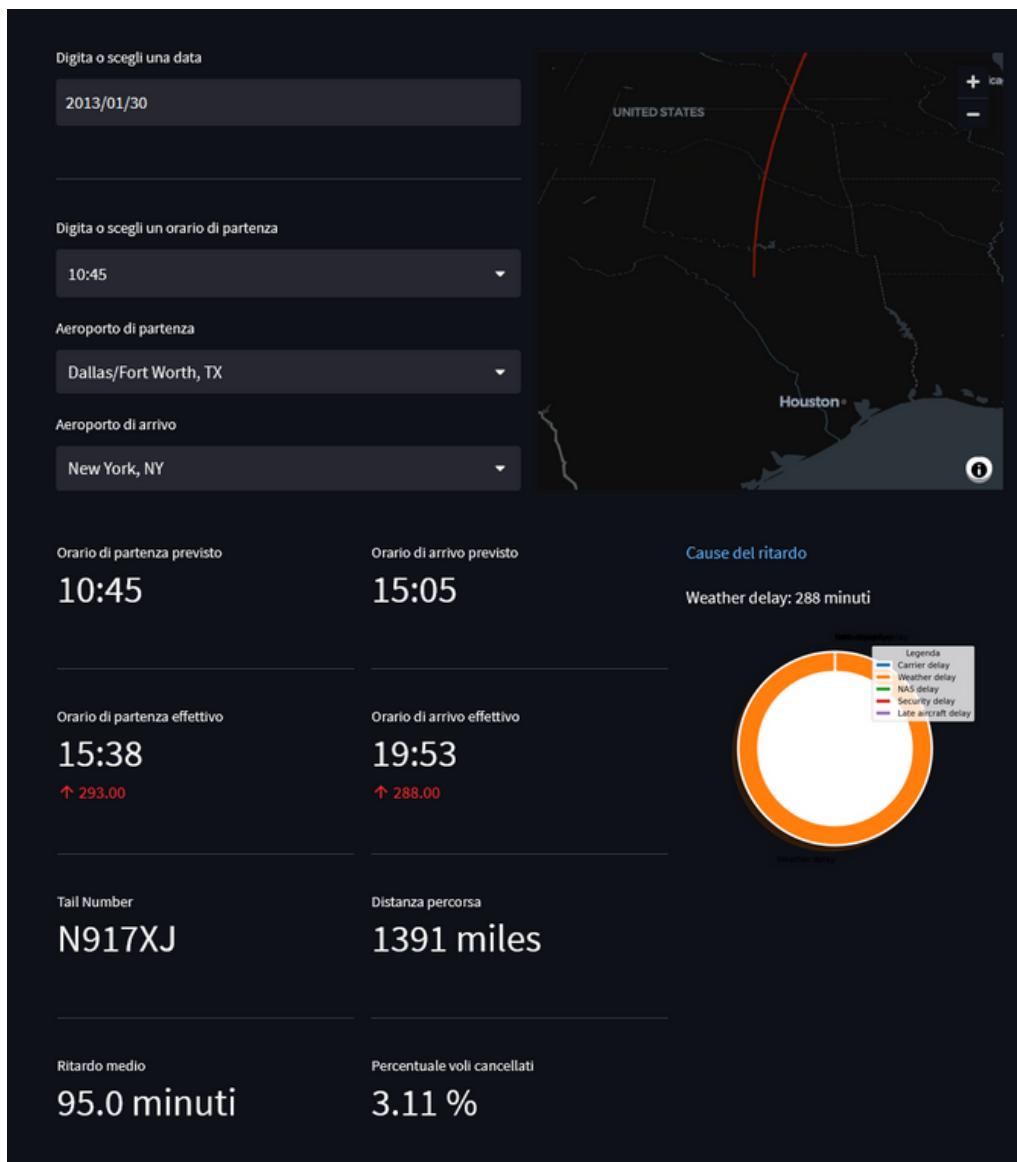
```
# Code for a generic flight
else:
    col3, col4, col5, col6 = streamlit.columns(4)
    col3.metric(label="Orario di partenza previsto", value=flight.__getitem___.("CRSDepTime").strftime("%H:%M"))
    if flight.__getitem___.("DepDelay") == "0.00":
        deltaCol = "off"
    else:
        deltaCol = "inverse"
    col4.metric(label="Orario di partenza effettivo", value=flight.__getitem___.("DepTime").strftime("%H:%M"),
               delta=flight.__getitem___.("DepDelay"), delta_color=deltaCol)

    col5.metric(label="Orario di arrivo previsto", value=flight.__getitem___.("CRSArrTime").strftime("%H:%M"))
    if flight.__getitem___.("ArrDelay") == "0.00":
        deltaCol = "off"
    else:
        deltaCol = "inverse"
    col6.metric(label="Orario di arrivo effettivo", value=flight.__getitem___.("ArrTime").strftime("%H:%M"),
               delta=flight.__getitem___.("ArrDelay"), delta_color=deltaCol)

    streamlit.markdown("-----")
    col7, col8, col9, col10 = streamlit.columns(4)
    col7.metric(label="Tail Number", value=flight.__getitem___.("Tail_Number"))
    col8.metric(label="Distanza percorsa", value=str(flight.__getitem___.("Distance")) + " miles")
    ritardoMedio = flightsA.agg(avg("ArrDelay")).take(num=1)[0].__getitem___.("avg(ArrDelay)")
    col9.metric(label="Ritardo medio", value=str(ritardoMedio) + " minuti")
    dfMedia = df.filter(col("OriginCityName") == origin).filter(col("DestCityName") == destination)
    canc = dfMedia.filter(col("Cancelled") == "1.00").count()
    total = dfMedia.count()
    col10.metric(label="Percentuale voli cancellati", value=str(round(canc/total * 100, 2)) + " %")
```

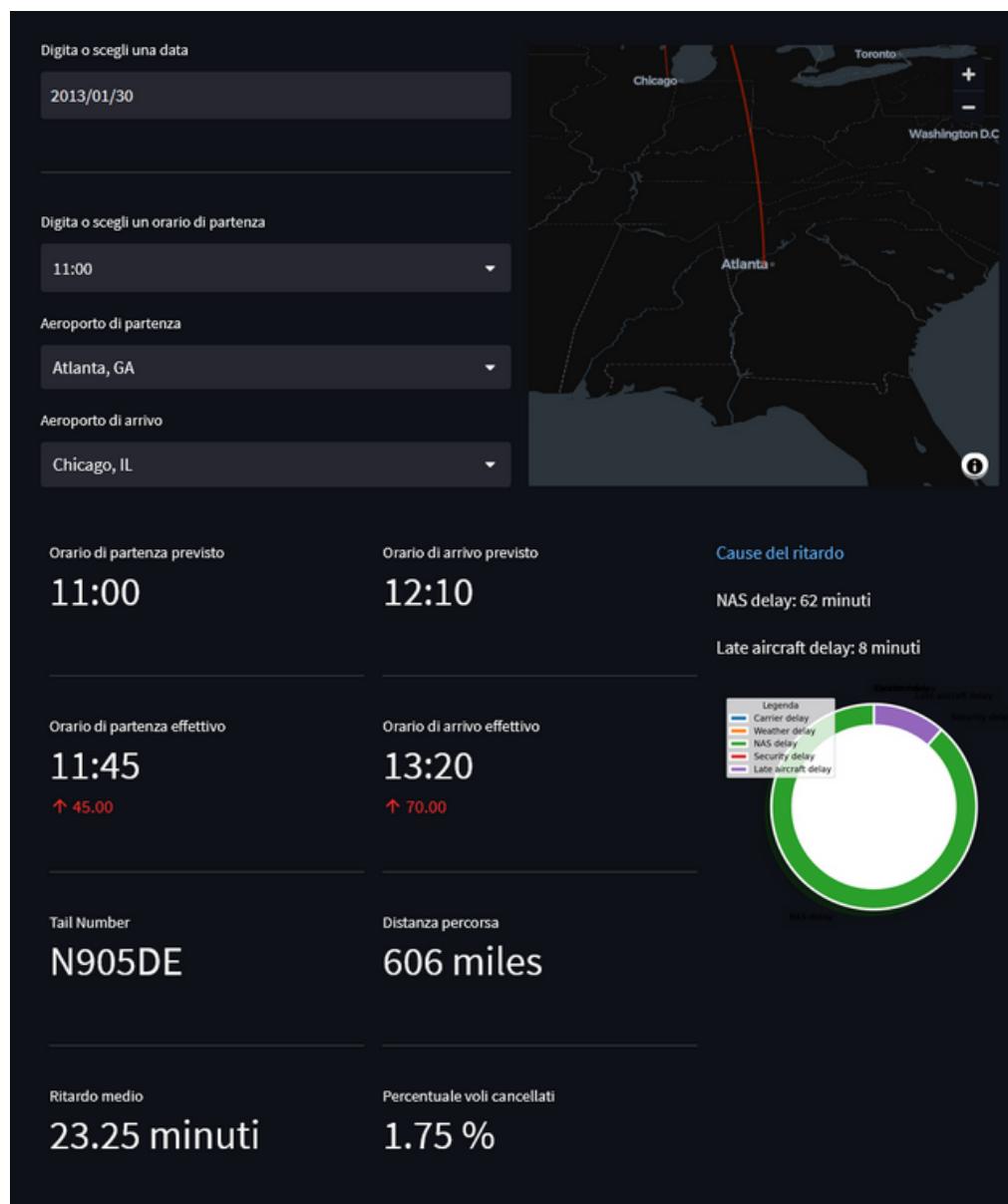
- Volo in ritardo di cui sono specificati i motivi: nel caso in cui siano riportati i motivi del ritardo, viene rappresentata sulla destra un pie chart, il cui codice è stato visto in precedenza, che tiene in considerazione la percentuale di incidenza di ciascuna causa.

Ad esempio, il volo del 30 gennaio 2013 delle 10:45 diretto da Dallas a New York ha visto un ritardo di 288 minuti per motivi legati al meteo

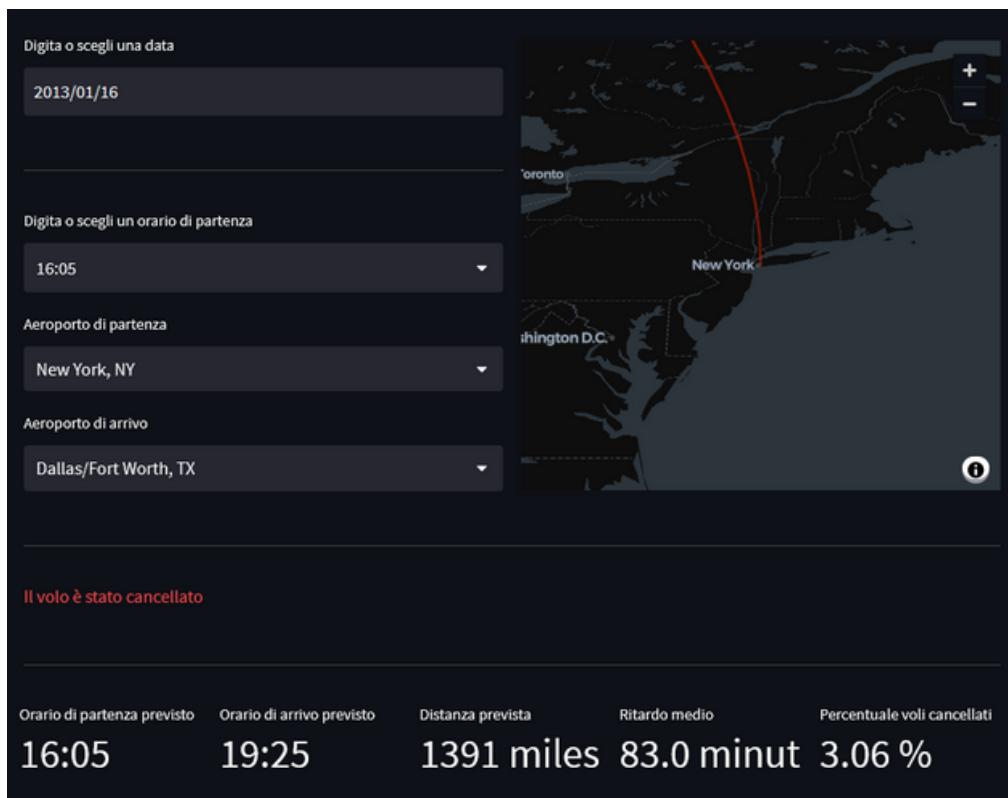


Ad esempio, il volo del 30 gennaio 2013 delle 11:00 da Atlanta a Chicago ha visto un ritardo pari a 70 minuti dovuto per

- 62 minuti da cause generiche legate al sistema di aviazione nazionale americano
- 8 minuti dovuti ad un ritardo dell'aeroplano



- Volo cancellato: nel caso di un volo cancellato viene visualizzato un messaggio di errore ma vengono comunque visualizzate le informazioni associate alla tratta e agli orari di partenza e arrivo previsti



- Volo dirottato

Nel caso di un volo dirottato vengono visualizzati orario di partenza ed arrivo previsti insieme all'orario effettivo di partenza.

Ad esempio, è riportato di seguito il volo del 10 gennaio delle 10:10 da Atlanta a Pensacola

Digit o scegli una data

2013/01/10

Digit o scegli un orario di partenza

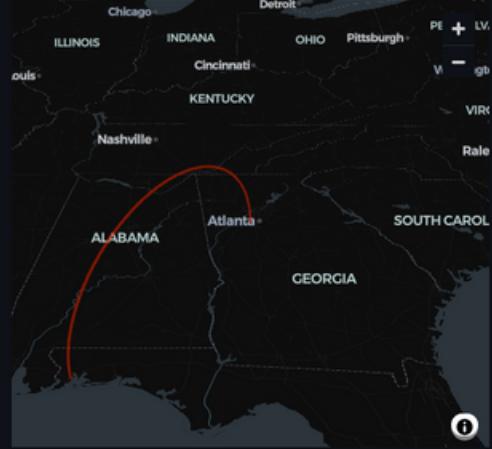
10:10

Aeroporto di partenza

Atlanta, GA

Aeroporto di arrivo

Pensacola, FL



Il volo è stato dirottato

Orario di partenza previsto	Orario di partenza effettivo	Orario di arrivo previsto
10:10	10:48 ↑ 38.00	10:28

Il volo non ha raggiunto la destinazione

Sono riportati gli aeroporti di deviazione

Primo aeroporto Secondo aeroporto

ECP ATL

Nel caso in cui l'aereo non sia giunto a destinazione viene visualizzato un messaggio all'utente.

Inoltre, vengono riportati i nomi degli aeroporti intermedi di deviazione.