1) Explain your choice of processing framework briefly.

We´ve chosen to do the exercises in PySPark. Python libraries have nice plotting features and jupyter notebook is a tool of choice, when it comes to combining ad-hoc analysis and reporting.

However, there are some considerations to do, when using the python API for Spark.

- Python is dynamically typed, which means that... as opposed to other API´s such as Scala or Java which...?
- Parsing objects between Spark and Python (serializing) is a quite expensive task, so this should be done with care. Why ?
- Working with RDD´s vs Spark Dataframes in python.... why are RDD´s not available

TODO: consider when to "persist" dataframes and what it means
TODO: is there a diffrence between "airline" and "carrier" ?

In [1]:
```
#import pyspark as spark
#import findspark
#import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
import pandas as pd
```

In [5]:
```
import matplotlib.pyplot as plt
```

Spark is set up with... Memory settings... Local xecuters (8 cores)...

In [2]:
```
# https://stackoverflow.com/questions/26562033/how-to-set-apache-spark-executor
-memory
sc._conf.get('spark.driver.memory')
```

Out[2]: '30g'

Creating a schema for the data using af Struct type

In [3]:
```
from pyspark.sql.types import StructType, StructField, IntegerType, StringType
schema = StructType([
    StructField("Year", IntegerType(), True),
    StructField("Month", IntegerType(), True),
    StructField("DayofMonth", IntegerType(), True),
    StructField("DayOfWeek", IntegerType(), True),
    StructField("DepTime", IntegerType(), True),
    StructField("CRSDepTime", IntegerType(), True),
    StructField("ArrTime", IntegerType(), True),
    StructField("CRSArrTime", IntegerType(), True),
    StructField("UniqueCarrier", StringType(), True),
    StructField("FlightNum", IntegerType(), True),
    StructField("TailNum", StringType(), True),
    StructField("ActualElapsedTime", IntegerType(), True),
    StructField("CRSElapsedTime", IntegerType(), True),
    StructField("AirTime", IntegerType(), True),
    StructField("ArrDelay", IntegerType(), True),
    StructField("DepDelay", IntegerType(), True),
    StructField("Origin", StringType(), True),
    StructField("Dest", StringType(), True),
    StructField("Distance", IntegerType(), True),
    StructField("TaxiIn", IntegerType(), True),
    StructField("TaxiOut", IntegerType(), True),
    StructField("Cancelled", IntegerType(), True),
    StructField("CancellationCode", StringType(), True),
    StructField("Diverted", IntegerType(), True),
    StructField("CarrierDelay", IntegerType(), True),
    StructField("WeatherDelay", IntegerType(), True),
    StructField("NASDelay", IntegerType(), True),
    StructField("SecurityDelay", IntegerType(), True),
    StructField("LateAircraftDelay", IntegerType(), True)])
flights = spark.read.csv("./data/2008.csv",header=True,schema=schema, nullValue
='NA')
airports = spark.read.csv("./data/airports.csv",header=True,inferSchema=True, n
ullValue='NA')
airlines = spark.read.csv("./data/carriers.csv",header=True,inferSchema=True, n
ullValue='NA')
weather = spark.read.csv("./data/weather/*daily.txt",header=True,inferSchema=Tr
ue, nullValue='NA')
stations = spark.read.option("delimiter", "|").option("header", "True").csv('./
data/weather/*station.txt')
```

2. How many flights were there from JFK to LAX?

Finding the number of flights from JFK to LAX

In [7]:
```
flights.where((col('Origin') == 'JFK') & (col('Dest') == 'LAX')).count()
```

Out[7]: 8078

3. What was the sum and average of all arrival delays for all delayed flights?

Finding the sum and average of all arrival delays for all delayed flights Average could be found using "Describe", but to include sum, we will use select

In [8]:
```
flights.where(col("ArrDelay")>0).select(avg('ArrDelay'), sum('ArrDelay')).show
()
```

```
+-----------------+-------------+
|    avg(ArrDelay)|sum(ArrDelay)|
+-----------------+-------------+
|32.17070626520387 6|     95852748|
+-----------------+-------------+
```

4. What was the average departure delay for each state?

Finding the average departure delay for each state. To do this, we need the airport data from airports.csv. Instead of defining the schema explicitely as above, for illustration purposes, we´ll just "infer" the schema, which means asking Spark to figure it out by presampling rows.

In [9]:
```
airports.show(2)
```

```
+----+--------------------+-----------+-----+-------+-----------+------------+
|iata|             airport|       city|state|country|        lat|        long|
+----+--------------------+-----------+-----+-------+-----------+------------+
| 00M|            Thigpen |Bay Springs|   MS|    USA|31.95376472|-89.23450472|
| 00R|Livingston Municipal| Livingston|   TX|    USA|30.68586111|-95.01792778|
+----+--------------------+-----------+-----+-------+-----------+------------+
only showing top 2 rows
```

Now, lets join the dataframes, group the result on states and calculate the average departure-delay- To illustrate the "agg" function used with a map, we´ll add the average arrival-delays aswell

In [10]:
```
# We´ll do the join and persiste, since we will use this dataframe later on asw
ell
# Broadcast airports if possible
flightsWithAirports = flights.join(airports, flights.Origin == airports.iata)
```

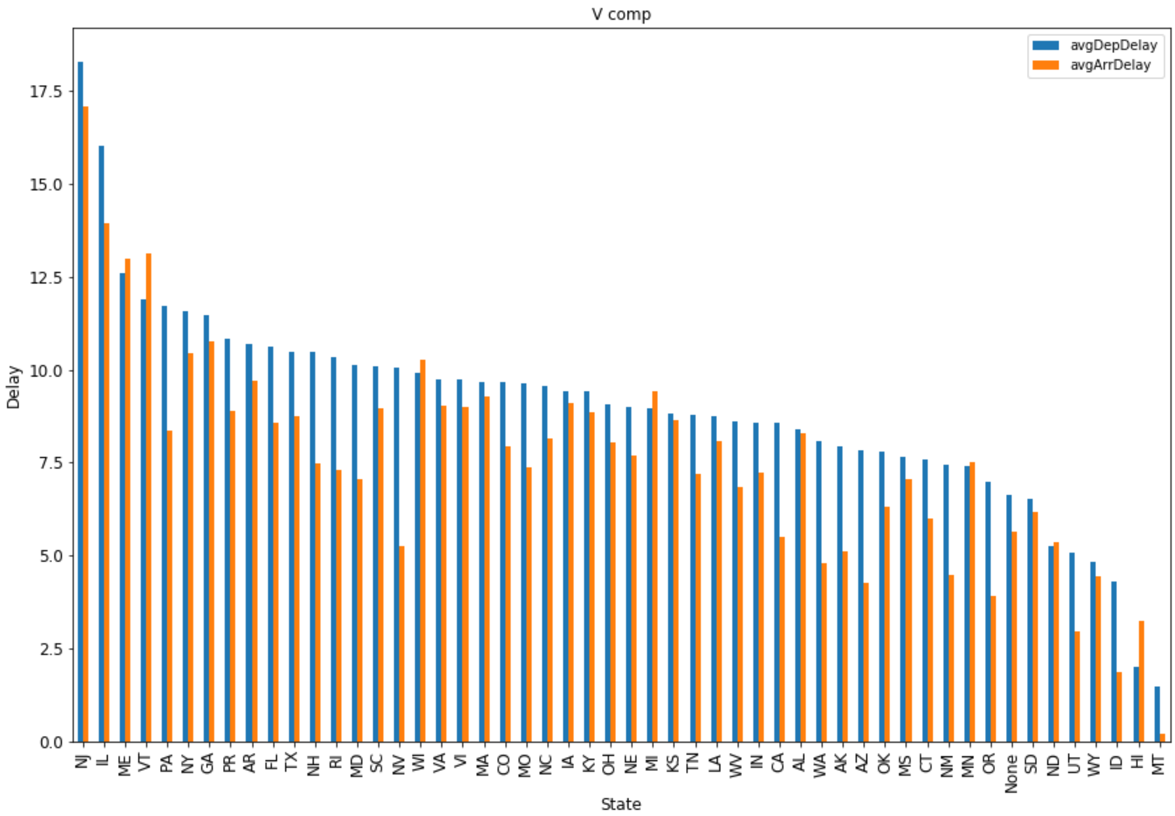
In [11]:
```
delays = flightsWithAirports.\
    groupBy(airports.state).\
        agg({"DepDelay": "avg", "ArrDelay": "avg"}).\
            select(col("state").alias("state"), \
               col("avg(DepDelay)").alias("avgDepDelay"), \
               col("avg(ArrDelay)").alias("avgArrDelay")).\
                    sort(desc("avgDepDelay"))
delays.show()
```

```
+-----+------------------+------------------+
|state|       avgDepDelay|       avgArrDelay|
+-----+------------------+------------------+
|   NJ| 18.28530315230682|17.073619219183303|
|   IL|16.037485162920703|13.927999295439097|
|   ME| 12.60202895487689|12.972307692307693|
|   VT|11.906676449009538| 13.11985294117647|
|   PA|11.706605875610164| 8.359997157696183|
|   NY|11.581353889575762|10.433212329260538|
|   GA| 11.47578943937115|10.746965986839188|
|   PR|10.823683322079676| 8.884239061374899|
|   AR|10.697886119257086| 9.709514325111076|
|   FL|10.617784856557332| 8.554335060599021|
|   TX|10.484268832380778| 8.741412350982355|
|   NH|10.483407140123559| 7.463268777088934|
|   RI|10.345095558668053| 7.284535521603119|
|   MD|10.136788700696506|7.0616724670931115|
|   SC|10.073743016759776| 8.942515845928815|
|   NV|10.047854928293972| 5.234664517182271|
|   WI| 9.898691052537206|10.273451327433628|
|   VA| 9.741461461852408| 9.015987468487651|
|   VI| 9.727703703703703|  9.00453446191052|
|   MA| 9.677755692715417| 9.280603542532255|
+-----+------------------+------------------+
only showing top 20 rows
```

Quite difficult to get a sense of this result, so lets visualize it. PySpark does not have plotting capabillities per se, so we´ll convert the Spark-dataframe to a pandas dataframe (requires installing python Pandas and MatPlotLib libraries). Pandas has several easy-to-use plotting features, and sorting the data by descending departure delay will give us a visual sense of the correlation btw departure delay and arrival delay (state-wise):

In [10]:
```
pdDelays = delays.toPandas()
```

In [10]:
```
%matplotlib inline
states=pdDelays['state'].tolist()
#states
ax = pdDelays[['avgDepDelay','avgArrDelay']].plot(kind='bar', title ="V comp",
figsize=(15, 10), legend=True, fontsize=12)
ax.set_xticklabels(states)
ax.set_xlabel("State", fontsize=12)
ax.set_ylabel("Delay", fontsize=12)
plt.show()
```



5. Which airline performed the worst seen from a customer perspective ?

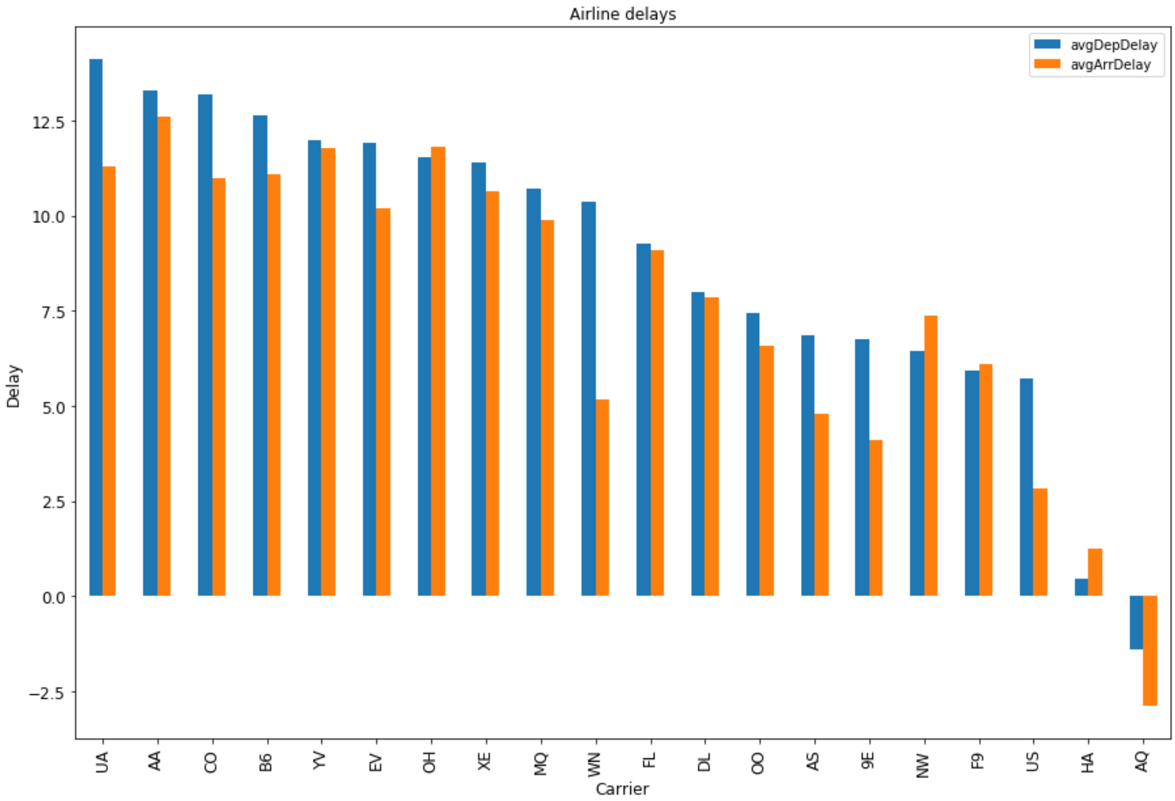
Analysing airlines, lets first load the carriers.csv file, that contains carrier-names instead of just codes. TODO - join for carrier name Broadcast join if possible

In [12]:
```
airlines.show()
```

```
+----+--------------------+
|Code|         Description|
+----+--------------------+
| 02Q|       Titan Airways|
| 04Q|  Tradewind Aviation|
| 05Q| Comlux Aviation, AG|
| 06Q|Master Top Linhas...|
| 07Q| Flair Airlines Ltd.|
| 09Q|      Swift Air, LLC|
| 0BQ|                 DCA|
| 0CQ|ACM AIR CHARTER GmbH|
| 0FQ|Maine Aviation Ai...|
| 0GQ|Inter Island Airw...|
| 0HQ|Polar Airlines de...|
|  0J|          JetClub AG|
| 0JQ|     Vision Airlines|
| 0KQ|Mokulele Flight S...|
| 0LQ|   Metropix UK, LLP.|
| 0MQ|Multi-Aero, Inc. ...|
|  0Q| Flying Service N.V.|
|  16|   PSA Airlines Inc.|
|  17|   Piedmont Airlines|
|  1I|Sky Trek Int'l Ai...|
+----+--------------------+
only showing top 20 rows
```

Now, lets take a look at each airline to determine, which performed the worst, see from a customer perspective. At first, lets plot the same delay-data as above, but with an airline focus instead of airport/state focus:

In [6]:
```
#carrierDelays = flights.join(flights, carriers.Code == flights.UniqueCarrie
r).\
#    groupBy(flights.UniqueCarrier).\

carrierDelays = flights.\
    groupBy(flights.UniqueCarrier).\
        agg({"DepDelay": "avg", "ArrDelay": "avg"}).\
            select(col("UniqueCarrier").alias("UniqueCarrier"), \
               col("avg(DepDelay)").alias("avgDepDelay"), \
               col("avg(ArrDelay)").alias("avgArrDelay")).\
                    sort(desc("avgDepDelay")).toPandas()

carriers=carrierDelays['UniqueCarrier'].tolist()

ax = carrierDelays[['avgDepDelay','avgArrDelay']].plot(kind='bar', title ="Airl
ine delays", figsize=(15, 10), legend=True, fontsize=12)
ax.set_xticklabels(carriers)
ax.set_xlabel("Carrier", fontsize=12)
ax.set_ylabel("Delay", fontsize=12)
plt.show()
print(carrierDelays.to_string(index=False))
```



UniqueCarrier	avgDepDelay	avgArrDelay
UA	14.112577	11.291322
AA	13.280898	12.607194
CO	13.185230	10.979037
B6	12.653396	11.084184
YV	12.000675	11.775181
EV	11.922538	10.208002
OH	11.536153	11.817468
XE	11.395866	10.635405
MQ	10.695642	9.890668
WN	10.383035	5.179678
FL	9.262713	9.091375
DL	8.007766	7.855163
OO	7.456443	6.598885
AS	6.848722	4.804346
9E	6.765860	4.111135
NW	6.463236	7.368539
F9	5.919602	6.108247
US	5.717490	2.848110
HA	0.455201	1.264409
AQ	-1.397783	-2.888674

Now, delays is not all that matters from a customer´s point of view, so lets compute a wide range of statistics to describe the airline performance. Still, all these descriptive measures do not give us "best airline", so lets also choose a couple of them and create a general performance-measure:

- depOnTimePct
- arrOnTimePct
- completedFlightsPct

These are all percentages (eg. values between 0 and 1) describing positive feautures, where 1 is "perfect" and 0 is "worst". If we multiply these measures for each airline, again 1 would describe "perfect performance" and 0 would describe "worst possible performance". Lets rank the airlines according to this airline performance measure:

In [7]:
```
# 1) flight-level feature engeneering
# 2) Grouping by carrier
# 3) Aggregating metrics pr. carrier
# 4) Calculation percentage metrics on carrier level

carrierPerformanceTable = flights.\
    select(flights.UniqueCarrier, \
           flights.DepDelay, \
           when(flights.DepDelay > 0,1).otherwise(0).alias("IsDepDelayed"),\
           when(flights.DepDelay > 0,0).otherwise(1).alias("IsDepOnTime"),\
           when(flights.ArrDelay > 0,1).otherwise(0).alias("IsArrDelayed"),\
           when(flights.ArrDelay > 0,0).otherwise(1).alias("IsArrOnTime"),\
           when(flights.Cancelled== 0,1).otherwise(0).alias("Completed"),\
           flights.DepDelay,
           flights.ArrDelay,
           flights.Cancelled
          ).\
    groupBy(flights.UniqueCarrier). \
    agg(sum("DepDelay").alias("DepDelay"), \
        max("DepDelay").alias("maxDepDelay"), \
        sum("ArrDelay").alias("ArrDelay"), \
        max("ArrDelay").alias("maxArrDelay"), \
        sum("IsDepDelayed").alias("isDepDelayed"), \
        sum("IsDepOnTime").alias("isDepOnTime"), \
        sum("IsArrDelayed").alias("isArrDelayed"), \
        sum("IsArrOnTime").alias("isArrOnTime"), \
        sum("Cancelled").alias("isCancelled"),\
        sum("Completed").alias("isCompleted"),\
        count(lit(1)).alias("numberOfFlights") \
       ). \
    select(col("UniqueCarrier"), \
           ((col("IsCompleted") / col("numberOfFlights"))*\
           (col("IsDepOnTime") / col("numberOfFlights"))*\
           (col("IsArrOnTime") / col("numberOfFlights"))).alias("performanceMea
sure"),\
           round(col("IsDepOnTime") / col("numberOfFlights")*100,2).alias("depO
nTimePct"),\
           round(col("IsArrOnTime") / col("numberOfFlights")*100,2).alias("arrO
nTimePct"),\
           round(col("IsDepDelayed") / col("numberOfFlights")*100,2).alias("dep
DelayedPct"),\
           round(col("IsArrDelayed") / col("numberOfFlights")*100,2).alias("arr
DelayedPct"),\
           round(col("DepDelay") / col("isDepDelayed"),2).alias("AvgDepDelayWhe
nDelayed"),\
           round(col("ArrDelay") / col("isArrDelayed"),2).alias("AvgArrDelayWhe
nDelayed"),\
           round(col("MaxArrDelay"),2).alias("MaxArrDelay"),\
           round(col("MaxDepDelay"),2).alias("MaxDepDelay"),\
           round(col("isCancelled"),2).alias("numberOfCancelledFlights"),\
           round(col("isCancelled") / col("numberOfFlights")*100,2).alias("canc
ellationPct"),\
           round(col("isCompleted") / col("numberOfFlights")*100,2).alias("comp
letedPct")\
          ).sort(desc("performanceMeasure")).toPandas()

carrierPerformanceTable
```

Out[7]:

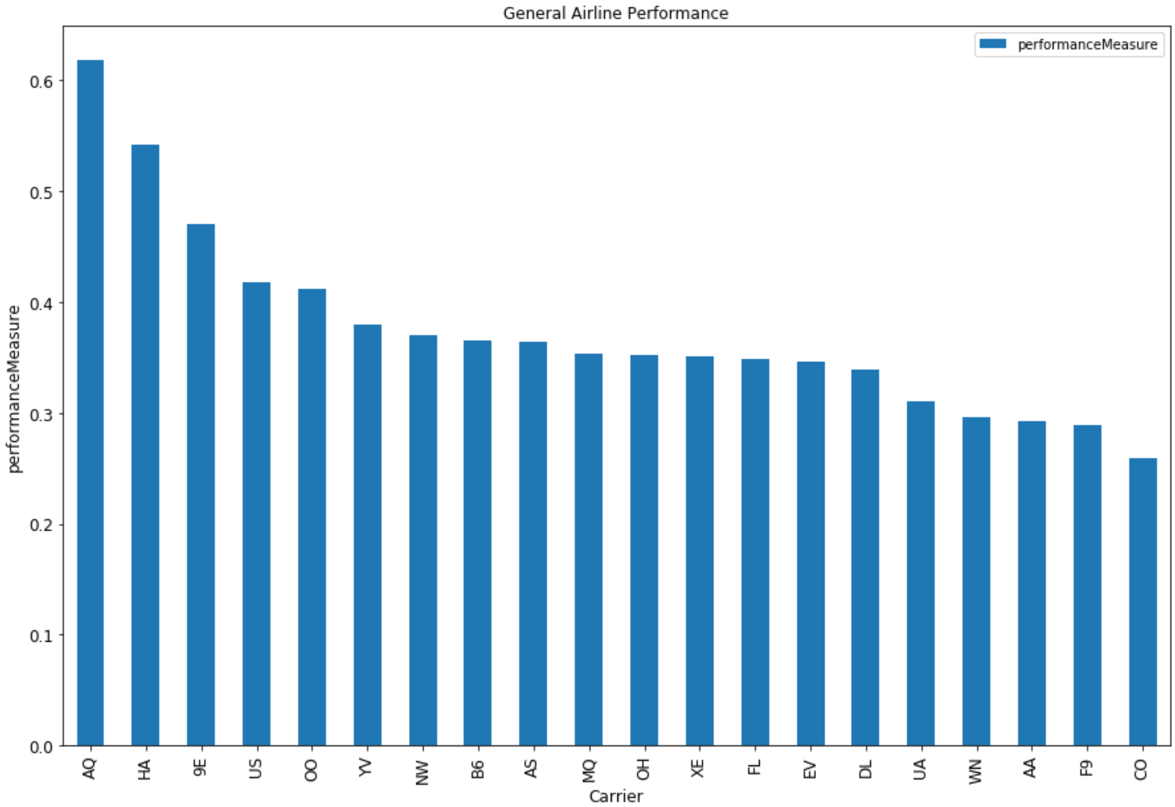
	UniqueCarrier	performanceMeasure	depOnTimePct	arrOnTimePct	depDelayedPct	arrDelay
0	AQ	0.618103	82.27	75.54	17.73	24.46
1	HA	0.542419	78.55	69.70	21.45	30.30
2	9E	0.470375	73.87	65.45	26.13	34.55
3	US	0.418076	67.37	62.97	32.63	37.03
4	OO	0.411363	68.59	61.32	31.41	38.68
5	YV	0.379888	69.81	56.46	30.19	43.54
6	NW	0.370622	68.80	54.32	31.20	45.68
7	B6	0.365074	64.47	57.57	35.53	42.43
8	AS	0.364197	62.82	58.81	37.18	41.19
9	MQ	0.353339	63.21	58.07	36.79	41.93
10	OH	0.352610	71.00	51.34	29.00	48.66
11	XE	0.350882	63.71	56.58	36.29	43.42
12	FL	0.348255	63.81	55.05	36.19	44.95
13	EV	0.346459	62.72	56.25	37.28	43.75
14	DL	0.339081	64.05	53.75	35.95	46.25
15	UA	0.310971	57.48	55.40	42.52	44.60
16	WN	0.296358	49.15	60.93	50.85	39.07
17	AA	0.292314	58.43	51.52	41.57	48.48
18	F9	0.289217	56.79	51.09	43.21	48.91
19	CO	0.258553	49.84	52.53	50.16	47.47

So, all these measures still do not give us "best airline". Lets choose a couple of features:

- depOnTimePct
- arrOnTimePct
- completedFlightsPct

These are all percentages (eg. values between 0 and 1) describing positive feautures, where 1 is "perfect" and 0 is "worst". If we multiply these measures for each airline, again 1 would describe "perfect performance" and 0 would describe "worst possible performance". Lets rank the airlines according to this airline performance measure:

In [12]:
```
carriers=carrierPerformanceTable['UniqueCarrier'].tolist()

ax = carrierPerformanceTable[['performanceMeasure']].plot(kind='bar', title ="General Airline Performance", figsize=(15, 10), legend=True, fontsize=12)
ax.set_xticklabels(carriers)
ax.set_xlabel("Carrier", fontsize=12)
ax.set_ylabel("performanceMeasure", fontsize=12)
plt.show()
```



So, it looks like this years price for best airline goes to AQ !

6. Which airport performed the worst seen from a customer perspective?

Lets do the same kind of analysis on airports, eg. which one performes worst, as seen from a customer´s viewpoint. Airport performance from a customer´s viewpoint could be many things. Some characteristicts could also depend on wether the airport is a origination or a destination for a given flight. However, lets focus on the individual airport and ignore, the origination / destination aspect.

In [18]:
```
airports.show(2)
```

```
+----+--------------------+-----------+-----+-------+-----------+------------+
|iata|             airport|       city|state|country|        lat|        long|
+----+--------------------+-----------+-----+-------+-----------+------------+
| 00M|            Thigpen |Bay Springs|   MS|    USA|31.95376472|-89.23450472|
| 00R|Livingston Municipal| Livingston|   TX|    USA|30.68586111|-95.01792778|
+----+--------------------+-----------+-----+-------+-----------+------------+
only showing top 2 rows
```

In [11]:
```
# Join with airports to get destination airport name
destinationAirports=airports.select(col("iata"),col("airport")).withColumnRenam
ed("iata","destIata").withColumnRenamed("airport","destAirport")
# Join with airports to get origination airport info
flightsWithAirports = flights.join(destinationAirports, flights.Dest == destina
tionAirports.destIata).\
    alias("flightsWithDestinationAirports").\
        join(airports,flights.Origin == airports.iata)

# Pretty print the first 10 rows using pandas
flightsWithAirports.limit(10).toPandas()
```

Out[11]:

	Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime	Unique
0	2008	1	3	4	2003	1955	2211	2225	WN
1	2008	1	3	4	754	735	1002	1000	WN
2	2008	1	3	4	628	620	804	750	WN
3	2008	1	3	4	926	930	1054	1100	WN
4	2008	1	3	4	1829	1755	1959	1925	WN
5	2008	1	3	4	1940	1915	2121	2110	WN
6	2008	1	3	4	1937	1830	2037	1940	WN
7	2008	1	3	4	1039	1040	1132	1150	WN
8	2008	1	3	4	617	615	652	650	WN
9	2008	1	3	4	1620	1620	1639	1655	WN

10 rows × 38 columns

In [15]:
```
pd.set_option('display.max_columns', 250)
#flightsWithDestinationAirports.where(col("iata") == "CYS").limit(10).toPandas
()
```

In [12]:
```
airportPerformanceTable = flightsWithAirports.\
    select(flightsWithAirports.destIata, \
           flightsWithAirports.destAirport, \
           when(flightsWithAirports.Diverted > 0,0).otherwise(1).alias("hasArri
ved"),\
          ).\
    groupBy(flightsWithAirports.destIata, flightsWithAirports.destAirport). \
    agg(sum("hasArrived").alias("hasArrived"),
        count(lit(1)).alias("numberOfFlights") \
       ). \
    select(col("destIata"), col("destAirport"), \
           col("numberOfFlights"),\
           (round(col("hasArrived") / col("numberOfFlights")*100,2)).alias("com
pletedPct")\
          ).sort(asc("completedPct")).limit(10).toPandas()
```

In [12]:
```
airportPerformanceTable
```

Out[12]:

	destIata	destAirport	numberOfFlights	completedPct
0	OGD	Ogden-Hinckley	2	0.00
1	CYS	Cheyenne	2	0.00
2	OME	Nome	1090	95.96
3	TEX	Telluride Regional	194	96.91
4	OTZ	Ralph Wien Memorial	1086	97.42
5	WRG	Wrangell	727	97.66
6	TWF	Joslin Field - Magic Valley	1788	97.76
7	SUN	Friedman Memorial	2905	97.80
8	PSG	James C. Johnson Petersburg	727	98.07
9	HHH	Hilton Head	836	98.09

How about security delay ?

7. On appserver2 (and possibly your laptop), these files are just stored as ordinary files in the OSmanaged file system. How would they be stored in HDFS running on a cluster? Which advantages/disadvantages would that give?

The HDFS (Haddop Distributed File System) is a distributed filesystem that supports parallellism in file reading/writing on multiple machones. This means, that every "logical" file is split into partitions, that are placed on different machines on local storage. This gives us the following benefits:

- Readin the whole logical file can be done in parallel by individual machines
- Having the partitioned data on local storage, some transformnations can be performed directly on the local partion of data
- Being able to store files that are larger than any single local harddrive
- we have fault-tollerance, since all partitions are replicated three (default) times on different nodes The partitioning scheme and replcation however presents a choice between:
- Consistency, Availability and Partition tolerance

This means, that if partitioning tolerance is given in HDFS (meaning, that if one partition-replica is corrupted, the system will still be running), we need to chose between consistency and availability. HDFS offers consistency - thus, we can run into availability-issues, since a write to a file means, that to ensure that consistency, this write needs to be replicated to other replicas before being able to guarantee a consistent read of the same file. If a network (or other) failure prevents this replication, then the system is down.

Basically, this means, that the HDFS is not a high-availability system, because it gives priority to consistency.

How about security delay ?

In [13]:
```
airports.show(2)
```

```
+----+--------------------+-----------+-----+-------+-----------+------------+
|iata|             airport|       city|state|country|        lat|        long|
+----+--------------------+-----------+-----+-------+-----------+------------+
| 00M|            Thigpen |Bay Springs|   MS|    USA|31.95376472|-89.23450472|
| 00R|Livingston Municipal| Livingston|   TX|    USA|30.68586111|-95.01792778|
+----+--------------------+-----------+-----+-------+-----------+------------+
only showing top 2 rows
```

Clustering

In [21]:

```
import urllib.request
import zipfile
import os

def downloadAndUnzip(url, filename):
    downloadFile=url+filename
    targetFile="./data/downloadStaging/"+filename
    print("Downloading and upzipping: "+downloadFile)
    urllib.request.urlretrieve(downloadFile, "./data/downloadStaging/"+filename
)
    zip_ref = zipfile.ZipFile(targetFile, 'r')
    zip_ref.extractall("./data/downloadStaging")
    zip_ref.close()
    # Cleanup
    os.system('cp ./data/downloadStaging/*daily.txt ./data/weather/')
    os.system('cp ./data/downloadStaging/*station.txt ./data/weather/')
    os.system('rm ./data/downloadStaging/*')


years=["2008"]
months=["01","02","03","04","05","06","07","08","09","10","11","12"]
for year in years:
    for month in months:
        downloadAndUnzip("https://www.ncdc.noaa.gov/orders/qclcd/","QCLCD"+year
+month+".zip")
```

```
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200801.
zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200802.
zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200803.
zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200804.
zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200805.
zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200806.
zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200807.
zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200808.
zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200809.
zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200810.
zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200811.
zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200812.
zip
```

In [17]:
```
pd.set_option('display.max_columns', 250)
weatherPD = weather.limit(10).toPandas()
weatherPD
```

Out[17]:

	WBAN	YearMonthDay	Tmax	TmaxFlag	Tmin	TminFlag	Tavg	TavgFlag	Depart	DepartFlag
0	3013	20081201	51		21		36		M	
1	3013	20081202	71		21		46		M	
2	3013	20081203	51		28		40		M	
3	3013	20081204	28		14		21		M	
4	3013	20081205	39		4		22		M	
5	3013	20081206	58		16		37		M	
6	3013	20081207	67		21		44		M	
7	3013	20081208	59		32		46		M	
8	3013	20081209	32		13		23		M	
9	3013	20081210	43		9		26		M	

In [13]:
```
## pd.set_option('display.max_columns', 250)
sqlContext.registerDataFrameAsTable(stations, "stationsTable")
callSigns=sqlContext.sql("SELECT distinct WBAN as stationWBAN, CallSign from st
ationsTable").persist()
callSigns.limit(2).toPandas()
```

Out[13]:

	stationWBAN	CallSign
0	03041	MYP
1	04815	228

We´ll join flight data and weather-station data to translate IATA callsign to WBAN, which is a key in weatherdata. Also, we ´ll construnct a "yearMonthDay" column, that will be used for joining later on

In [55]:
```
flightsWithStations.columns
```

Out[55]:
```
['Year',
 'Month',
 'DayofMonth',
 'DayOfWeek',
 'DepTime',
 'CRSDepTime',
 'ArrTime',
 'CRSArrTime',
 'UniqueCarrier',
 'FlightNum',
 'TailNum',
 'ActualElapsedTime',
 'CRSElapsedTime',
 'AirTime',
 'ArrDelay',
 'DepDelay',
 'Origin',
 'Dest',
 'Distance',
 'TaxiIn',
 'TaxiOut',
 'Cancelled',
 'CancellationCode',
 'Diverted',
 'CarrierDelay',
 'WeatherDelay',
 'NASDelay',
 'SecurityDelay',
 'LateAircraftDelay',
 'stationWBAN',
 'CallSign',
 'yearMonthDay']
```

In [14]:
```
# Start by joining flights with station data to translate IATA-code to WBAN cod
e
flightsWithStations = flights.join(callSigns, flights.Origin==callSigns.CallSig
n,'left_outer').withColumn("yearMonthDay",(concat(col('Year'),lpad(col('Month'
), 2, '0') ,lpad(col('DayofMonth'), 2, '0'))).cast("Integer"))
#flightsWithStations = flights.join(callSigns, flights.Dest==callSigns.CallSig
n,'left_outer').withColumn("yearMonthDay",(concat(col('Year'),lpad(col('Mont
h'), 2, '0') ,lpad(col('DayofMonth'), 2, '0'))).cast("Integer"))
print("Lets check, if all flights have station information")
print("Looks like we loose flights from the following destinations: ")
flightsWithStations.where(col('stationWBAN').isNull()).groupby(col("Origin")).c
ount().show()
#flightsWithStations.where(col('stationWBAN').isNull()).groupby(col("Dest")).co
unt().show()
print("Lets join the weather")
flightsWithStations.limit(10).toPandas()
```

```
Lets check, if all flights have station information
Looks like we loose flights from the following destinations:
+------+-----+
|Origin|count|
+------+-----+
|   PSE|  755|
|   SCE|  645|
|   HHH|  836|
|   FCA| 2762|
|   CLD| 2303|
+------+-----+

Lets join the weather
```

Out[14]:

	Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime	Unique
0	2008	1	3	4	2003	1955	2211	2225	WN
1	2008	1	3	4	754	735	1002	1000	WN
2	2008	1	3	4	628	620	804	750	WN
3	2008	1	3	4	926	930	1054	1100	WN
4	2008	1	3	4	1829	1755	1959	1925	WN
5	2008	1	3	4	1940	1915	2121	2110	WN
6	2008	1	3	4	1937	1830	2037	1940	WN
7	2008	1	3	4	1039	1040	1132	1150	WN
8	2008	1	3	4	617	615	652	650	WN
9	2008	1	3	4	1620	1620	1639	1655	WN

10 rows × 32 columns

In [25]:
```
weather.count()
```

Out[25]: 359377

In [15]:
```
# Now, lets join the weather information for the originating airport, using SQL
 syntax:
# This might be a tough one, joining 7 mill flights with 360K rows of weatherda
ta
# How about the explain plan
#sqlContext.registerDataFrameAsTable(flightsWithStations, "flightsWithStation
s")
#sqlContext.registerDataFrameAsTable(weather, "weather")
#flightsWithOriginWeather=sqlContext.sql("SELECT a.*, b.* \
#                                          from flightsWithStations a left joi
n \
#                                             weather               b on (a.Ca
llSign = b.WBAN & \
#                                       a.yearMonthDay = b.YearMonthDay )")
flightsWithOriginWeather=flightsWithStations.join(weather,(flightsWithStations.
stationWBAN==weather.WBAN) & (flightsWithStations.yearMonthDay == weather.YearM
onthDay),'left_outer').drop('YearMonthDay')
#flightsWithDestinationWeather=flightsWithStations.join(weather,(flightsWithSta
tions.stationWBAN==weather.WBAN) & (flightsWithStations.yearMonthDay == weathe
r.YearMonthDay),'left_outer').drop('YearMonthDay')
print("This join looses the following number of rows:")
flightsWithOriginWeather.where(col('WBAN').isNull()).count()
```

```
This join looses the following number of rows:
```

Out[15]: 71613

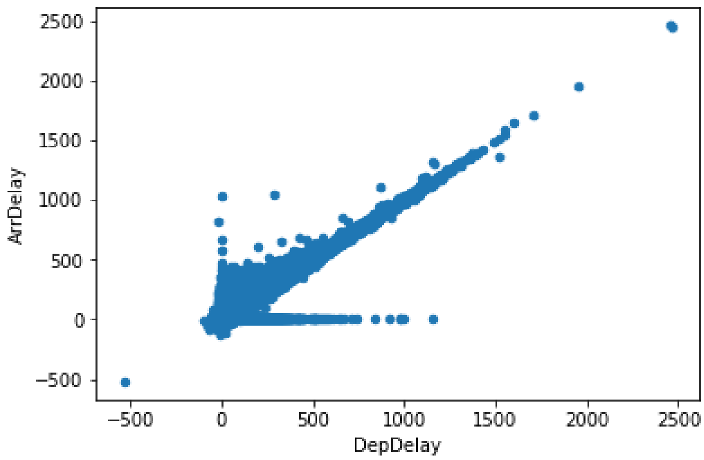
From the above query, it looks like we loose

PySPark ML library - in contrast to Scikit-learn - requires features to be assembled in one column. The ML library supplies a method for doing so, vectorAssembler. Also, no NULL column are allowed, so we´ll replace them with 0 (as in "no delay").

In [16]:
```
dfForClustering = flightsWithOriginWeather.where(col('WBAN').isNotNull()).selec
t(col('UniqueCarrier'),col('DepDelay'),col('ArrDelay')).na.fill(0)
#dfForClustering = flightsWithDestinationWeather.where(col('WBAN').isNotNull
()).select(col('UniqueCarrier'),col('DepDelay'),col('ArrDelay')).na.fill(0)
```

In [28]:
```
dfForClustering.select(col('DepDelay'),col('ArrDelay')).toPandas().plot.scatter
(x='DepDelay',y='ArrDelay')
```

Out[28]: <matplotlib.axes._subplots.AxesSubplot at 0x7fc3950152b0>



Hvordan kan man være 1000 minuter forsinket i afgang men ankomme til tiden ?!? Hvordan kan man være 1000 minuter forsinket i ankomst men være afgået til tiden ?!?

In [17]:
```
dfForClustering.where(col('DepDelay').isNull()).count()
#dfForClustering.where(col('ArrDelay').isNull()).count()
#df.na.fill(0).show()
```

Out[17]: 0

In [71]:
```
from pyspark.ml.stat import Correlation
dfForClustering.stat.corr("DepDelay", "ArrDelay")
```

Out[71]: 0.9269186899131432

We could cluster from a combination of departure- and arrival delay, but that would yield a measure of delay, that is not quite intuitive. As it looks like departure and arrival delays are very much correlated, it would suffice to accept arrivaldelay only as a general measure of delay. This is probably also the most important delay-type, seen from a customer view-point.

Rather that setting up fx low/medium/high delay-groups, we look into the flights data for hidden groups, using the Kmeans clustering method to divide the flights into "natural" delay-groups.

In [19]:
```
from pyspark.ml.clustering import KMeans
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StandardScaler
%matplotlib inline
# VectorAssembler does not accept NULL values
#features = ['DepDelay']
features = ['ArrDelay']
assembler = VectorAssembler(inputCols=features, outputCol="features")
baseClusteringDF = assembler.transform(dfForClustering).cache()
baseClusteringDF.limit(10).toPandas()
```

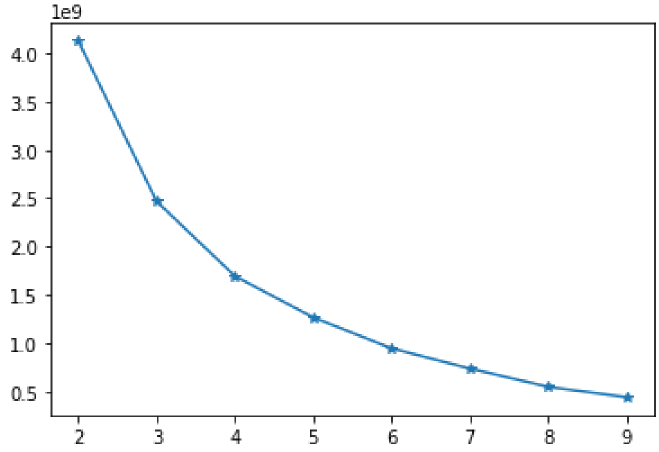
Out[19]:

	UniqueCarrier	DepDelay	ArrDelay	features
0	WN	8	-14	[-14.0]
1	WN	19	2	[2.0]
2	WN	8	14	[14.0]
3	WN	-4	-6	[-6.0]
4	WN	34	34	[34.0]
5	WN	25	11	[11.0]
6	WN	67	57	[57.0]
7	WN	-1	-18	[-18.0]
8	WN	2	2	[2.0]
9	WN	0	-16	[-16.0]

In [24]:
```
from tqdm import tqdm
elbowDict={}
numberOfClusters = range(2,10)
for cluster in tqdm(numberOfClusters):
    #print("calculating cost for k={}".format(cluster))
    kmeans = KMeans(k=cluster, seed=1)  # 2 clusters here
    model = kmeans.fit(baseClusteringDF.select('features'))
    #transformed = model.transform(baseClusteringDF)
    #featuresAndPrediction = transformed.select("features", "prediction")
    WSSSE = model.computeCost(baseClusteringDF.select('features'))
    #print(str(WSSSE))
    elbowDict[cluster]=WSSSE
```

```
100%|██████████| 8/8 [01:23<00:00, 10.49s/it]
```

In [25]:
```
import matplotlib.pylab as plt
lists = sorted(elbowDict.items()) # sorted by key, return a list of tuples
x, y = zip(*lists) # unpack a list of pairs into two tuples
plt.plot(x, y, marker='*')
plt.show()
```



There is no clear "elbow" point, so we´ll choose 5 clusters for the number of delay-groups

In [22]:
```
cluster=5
kmeans = KMeans(k=cluster, seed=1)
model = kmeans.fit(baseClusteringDF.select('features'))
transformed = model.transform(baseClusteringDF)
```

Lets take a look at the groups:

In [23]:
```
from pyspark.sql import functions as F
from pyspark.sql.window import Window
#delayGroups = transformed.groupBy("prediction").agg(avg('DepDelay').alias('avg
DepDelay'), \
#                                                        min('DepDelay').alias
('minDepDelay'), \
#                                                        max('DepDelay').alias
('maxDepDelay'), \
#                                                        sum(lit(1)).alias('num
berOfFlights')).\
#withColumn('delayGroup',F.row_number().over(Window.partitionBy(lit(1)).orderBy
(col("avgDepDelay")))).cache()
#delayGroups.toPandas().sort_values(by=['delayGroup'])

delayGroups = transformed.groupBy("prediction").agg(avg('ArrDelay').alias('avgA
rrDelay'), \
                                                       min('ArrDelay').alias(
'minArrDelay'), \
                                                       max('ArrDelay').alias(
'maxArrDelay'), \
                                                       sum(lit(1)).alias('numb
erOfFlights')).\
withColumn('delayGroup',F.row_number().over(Window.partitionBy(lit(1)).orderBy(
col("avgArrDelay")))).cache()
delayGroups.toPandas().sort_values(by=['delayGroup'])
```

Out[23]:

	prediction	avgArrDelay	minArrDelay	maxArrDelay	numberOfFlights	delayGroup
0	0	-11.843752	-519	-3	3279086	1
1	2	7.662529	-2	29	2739910	2
2	4	51.709830	30	91	669026	3
3	1	132.032380	92	213	215473	4
4	3	297.201704	214	2461	34620	5

In [26]:
```
flightsWithDelayGroup = flightsWithOriginWeather.where(col('WBAN').isNotNull())
.\
                        join(delayGroups,(coalesce(flightsWithOriginWeather.Arr
Delay,lit(0)) >=delayGroups.minArrDelay) & \
                                         (coalesce(flightsWithOriginWeather.Arr
Delay,lit(0)) <=delayGroups.maxArrDelay)).cache()
#flightsWithDelayGroup = flightsWithDestinationWeather.where(col('WBAN').isNotN
ull()).\
#                        join(delayGroups,(coalesce(flightsWithDestinationWeath
er.ArrDelay,lit(0)) >=delayGroups.minArrDelay) & \
#                                         (coalesce(flightsWithDestinationWeath
er.ArrDelay,lit(0)) <=delayGroups.maxArrDelay)).cache()
```

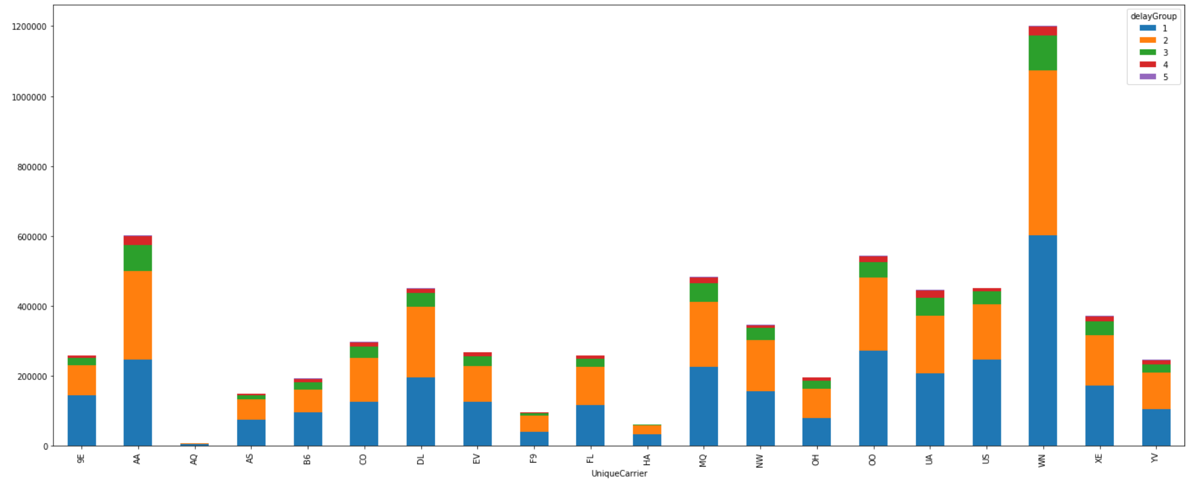
In [23]:
```
#flightsWithDelayGroup.limit(10).toPandas()
# Persist to disk to be able to restart
flightsWithDelayGroup.write.mode('overwrite').parquet("./data/flightsWithDelayG
roup.parquet")
weather.write.mode('overwrite').parquet("./data/weather.parquet")
```

In [24]:
```
flightsWithDelayGroup=sqlContext.read.parquet("./data/flightsWithDelayGroup.par
quet")
weather=sqlContext.read.parquet("./data/weather.parquet")
```

Lets stack the flights with delayGroup for each carrier. First we´ll aggregate the carrierinfo in Spark and then use Pandas to plot.

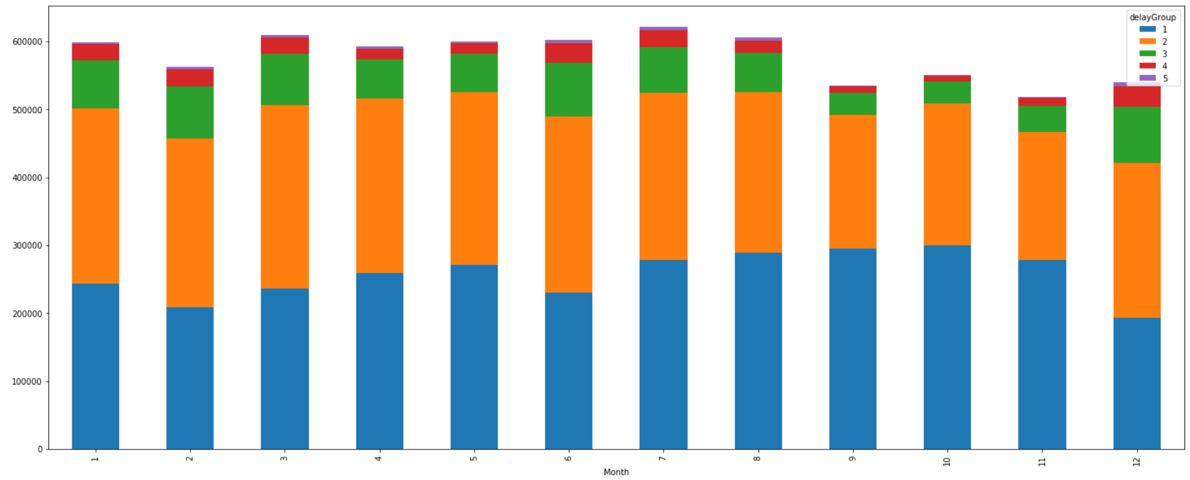
In [27]:
```
groupedUniqueCarriers = flightsWithDelayGroup.\
groupBy(col('UniqueCarrier'),col('delayGroup')).\
agg(sum(lit(1)).alias('numberOfFlights')).\
toPandas()

pt = groupedUniqueCarriers.pivot('UniqueCarrier', 'delayGroup', 'numberOfFlight
s')
pt.plot(kind='bar', stacked=True, figsize=(25,10))
```

Out[27]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fc460239c50>
```



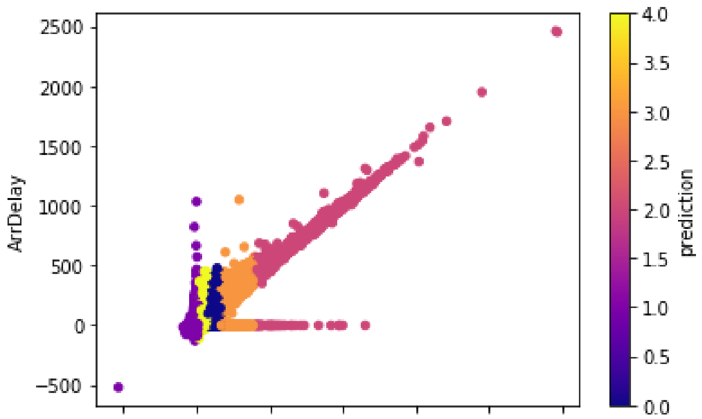
In [26]:
```
groupedUniqueCarriers = flightsWithDelayGroup.\
groupBy(col('Month'),col('delayGroup')).\
agg(sum(lit(1)).alias('numberOfFlights')).\
toPandas()

pt = groupedUniqueCarriers.pivot('Month', 'delayGroup', 'numberOfFlights')
pt.plot(kind='bar', stacked=True, figsize=(25,10))
```

Out[26]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f5351af5ef0>
```



In [42]:
```
transformed = model.transform(baseClusteringDF)
featuresAndPrediction = transformed.select("DepDelay","ArrDelay", "prediction")
featuresAndPredictionPD=featuresAndPrediction.select(col('DepDelay'),col('ArrDe
lay'),col('prediction')).toPandas()
featuresAndPredictionPD.plot.scatter(x='DepDelay',y='ArrDelay',c='prediction',
cmap='plasma')
#featuresAndPredictionPD.head(10)
```

Out[42]: <matplotlib.axes._subplots.AxesSubplot at 0x7fc395077e10>



Classification

In [4]:
```
#weather.columns
```

From https://spark.apache.org/docs/1.6.2/ml-features.html#rformula (https://spark.apache.org/docs/1.6.2/ml-features.html#rformula) : RFormula selects columns specified by an R model formula. It produces a vector column of features and a double column of labels. Like when formulas are used in R for linear regression, string input columns will be one-hot encoded, and numeric columns will be cast to doubles. If not already present in the DataFrame, the output label column will be created from the specified response variable in the formula.

In [5]:
```
#flightsWithDelayGroup.columns
```

Dropping "prediction" column, since this conflicts with below decisionTree, that outputs that column

In [ ]:
```
#dist=flightsWithDelayGroup.agg(
dist=weather.agg(
countDistinct("Tmax"),
countDistinct("TmaxFlag"),
countDistinct("Tmin"),
countDistinct("TminFlag"),
countDistinct("Tavg"),
countDistinct("TavgFlag"),
countDistinct("Depart"),
countDistinct("DepartFlag"),
countDistinct("DewPoint"),
countDistinct("DewPointFlag"),
countDistinct("WetBulb"),
countDistinct("WetBulbFlag"),
countDistinct("Heat"),
countDistinct("HeatFlag"),
countDistinct("Cool"),
countDistinct("CoolFlag"),
countDistinct("Sunrise"),
countDistinct("SunriseFlag"),
countDistinct("Sunset"),
countDistinct("SunsetFlag"),
countDistinct("CodeSum"),
countDistinct("CodeSumFlag"),
countDistinct("Depth"),
countDistinct("DepthFlag"),
countDistinct("Water1"),
countDistinct("Water1Flag"),
countDistinct("SnowFall"),
countDistinct("SnowFallFlag"),
countDistinct("PrecipTotal"),
countDistinct("PrecipTotalFlag"),
countDistinct("StnPressure"),
countDistinct("StnPressureFlag"),
countDistinct("SeaLevel"),
countDistinct("SeaLevelFlag"),
countDistinct("ResultSpeed"),
countDistinct("ResultSpeedFlag"),
countDistinct("ResultDir"),
countDistinct("ResultDirFlag"),
countDistinct("AvgSpeed"),
countDistinct("AvgSpeedFlag"),
countDistinct("Max5Speed"),
countDistinct("Max5SpeedFlag"),
countDistinct("Max5Dir"),
countDistinct("Max5DirFlag"),
countDistinct("Max2Speed"),
countDistinct("Max2SpeedFlag"),
countDistinct("Max2Dir"),
countDistinct("Max2DirFlag")).toPandas()
dist
```

In [28]:
```
flightsWithDelayGroup.limit(10).toPandas()
```

Out[28]:

	Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime	Unique
0	2008	6	29	7	1945	1945	2110	2105	WN
1	2008	6	29	7	1158	1145	1316	1305	WN
2	2008	6	29	7	1527	1530	1636	1650	WN
3	2008	6	29	7	754	800	900	920	WN
4	2008	6	29	7	954	1000	1157	1220	WN
5	2008	6	29	7	1832	1815	2042	2035	WN
6	2008	6	29	7	1639	1635	1933	1945	WN
7	2008	6	29	7	1031	1035	1338	1345	WN
8	2008	6	29	7	1113	1115	1429	1445	WN
9	2008	6	29	7	1759	1655	2119	2025	WN

10 rows × 86 columns

In [29]:
```
# We´ll create a binary classification target (delayedStatus)
# We´ll consider delayGroup 1 and 2 as no delay, since it is such a small delay
 (<29 mins)
flightsWithDelayStatus=flightsWithDelayGroup.withColumn('delayedStatus',when(fl
ightsWithDelayGroup.delayGroup > 2, 1).otherwise(0))
flightsWithDelayStatus.persist()
```

Out[29]:
```
DataFrame[Year: int, Month: int, DayofMonth: int, DayOfWeek: int, DepTime: in
t, CRSDepTime: int, ArrTime: int, CRSArrTime: int, UniqueCarrier: string, Flig
htNum: int, TailNum: string, ActualElapsedTime: int, CRSElapsedTime: int, AirT
ime: int, ArrDelay: int, DepDelay: int, Origin: string, Dest: string, Distanc
e: int, TaxiIn: int, TaxiOut: int, Cancelled: int, CancellationCode: string, D
iverted: int, CarrierDelay: int, WeatherDelay: int, NASDelay: int, SecurityDel
ay: int, LateAircraftDelay: int, stationWBAN: string, CallSign: string, WBAN:
int, Tmax: string, TmaxFlag: string, Tmin: string, TminFlag: string, Tavg: str
ing, TavgFlag: string, Depart: string, DepartFlag: string, DewPoint: string, D
ewPointFlag: string, WetBulb: string, WetBulbFlag: string, Heat: string, HeatF
lag: string, Cool: string, CoolFlag: string, Sunrise: string, SunriseFlag: str
ing, Sunset: string, SunsetFlag: string, CodeSum: string, CodeSumFlag: string,
Depth: string, DepthFlag: string, Water1: string, Water1Flag: string, SnowFal
l: string, SnowFallFlag: string, PrecipTotal: string, PrecipTotalFlag: string,
StnPressure: string, StnPressureFlag: string, SeaLevel: string, SeaLevelFlag:
string, ResultSpeed: string, ResultSpeedFlag: string, ResultDir: string, Resul
tDirFlag: string, AvgSpeed: string, AvgSpeedFlag: string, Max5Speed: string, M
ax5SpeedFlag: string, Max5Dir: string, Max5DirFlag: string, Max2Speed: string,
Max2SpeedFlag: string, Max2Dir: string, Max2DirFlag: string, prediction: int,
avgArrDelay: double, minArrDelay: int, maxArrDelay: int, numberOfFlights: bigi
nt, delayGroup: int, delayedStatus: int]
```

In [30]:
```
flightsWithDelayStatus.groupBy(col('delayedStatus')).count().show()
```

```
+-------------+-------+
|delayedStatus|  count|
+-------------+-------+
|            1| 919119|
|            0|6018996|
+-------------+-------+
```

We create a feature vector column for the classifier along with the label. Once this is done, we´ll drop all other columns, since we do not want to carry all this data around for no use.

In [32]:
```
# https://spark.apache.org/docs/2.2.0/ml-features.html#rformula
from pyspark.ml.feature import RFormula
formula = RFormula(
    #formula="delayedStatus ~ DepDelay + Tmax + TmaxFlag + Tmin + TminFlag + Ta
vg + Depart + DewPoint + WetBulb + Heat + Cool + Sunrise + Sunset + CodeSum  +
 Depth + SnowFall + SnowFallFlag + PrecipTotal + PrecipTotalFlag + StnPressure
 + SeaLevel + ResultSpeed + ResultDir + AvgSpeed + Max5Speed + Max5SpeedFlag +
 Max5Dir + Max2Speed + Max2SpeedFlag + Max2Dir",
    # Lets try non weather data
    formula="delayedStatus ~ DepDelay + DepTime + Distance ",
    featuresCol="features",
    labelCol="label")

output = formula.fit(flightsWithDelayStatus.na.fill(0).na.fill('None')).transfo
rm(flightsWithDelayStatus.na.fill(0).na.fill('None')).select("features","label"
)
```

In [36]:
```
output.limit(5).toPandas()
```

Out[36]:

	Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime	Unique
0	2008	6	29	7	612	615	728	735	WN
1	2008	6	29	7	1323	1255	1436	1415	WN
2	2008	6	29	7	1711	1715	1838	1835	WN
3	2008	6	29	7	930	935	1040	1055	WN
4	2008	6	29	7	1612	1555	1615	1615	WN

5 rows × 89 columns

Splitting into training- and test data. Lets count the occurances of labels in the training data - this gives us an idea of how balanced the dataset is. We might want to balance it befor training, so not to induce artificial bias towards the majority class.

In [33]:
```
#output.limit(10).toPandas()
(trainingData, testData) = output.randomSplit([0.8,0.2], seed = 13234 )
trainingData.groupBy(col('Label')).count().show()
```

```
+-----+-------+
|Label|  count|
+-----+-------+
|  0.0|4815457|
|  1.0| 735037|
+-----+-------+
```

We have an unbalanced trainingset, so we´ll downsample the majority class to get a balanced set. This way, we will avoid the bias in training the model. The testset however should resemble unseen data, thus we´ll that unbalanced.

Reduced to 1/10th of dataset in sample fraction

In [34]:
```
# Downsampling on-time flights (traininset only) to get a balanced dataset
from pyspark.sql import DataFrame
trainingDataBalanced = trainingData.where(col('label')==0).sample(False, (73503
7/4815457)/10, 42).unionAll(trainingData.where(col('label')==1).sample(False, 1
/10, 42))
trainingDataBalanced.groupBy(col('Label')).count().show()
```

```
+-----+-----+
|Label|count|
+-----+-----+
|  0.0|73165|
|  1.0|73456|
+-----+-----+
```

Saving the training set, so that we can restart the process from here later on.

In [35]:
```
trainingDataBalanced.persist()
```

Out[35]: `DataFrame[features: vector, label: double]`

In [34]:
```
#flightsWithDelayGroup.limit(10).toPandas()
# Persist to disk to be able to restart
trainingDataBalanced.write.mode('overwrite').parquet("./data/trainingDataBalanc
ed.parquet")
testData.write.mode('overwrite').parquet("./data/testData.parquet")
```

In [36]:
```
#flightsWithDelayGroup.limit(10).toPandas()
# Read disk-persistent datasets to restart
trainingDataBalanced=sqlContext.read.parquet("./data/trainingDataBalanced.parqu
et").persist()
testData=sqlContext.read.parquet("./data/testData.parquet").persist()
```

In [39]:
```
#trainingDataBalanced.columns
# Important, when training model, because of the iterative nature
testData.columns
```

Out[39]: `['features', 'label']`

In [32]:
```
trainingDataBalanced.limit(5).toPandas()
```

Out[32]:

	features	label
0	(-3.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...	0.0
1	(-1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...	0.0
2	(2.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...	0.0
3	(-5.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...	0.0
4	(-2.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...	0.0

Instatiate and fit the classifier - done using a pipeline.
Explain pipeline

In [33]:
```
# Cross validation - too expensive
#from pyspark.ml.evaluation import BinaryClassificationEvaluator
#from pyspark.ml.classification import DecisionTreeClassifier
#from pyspark.ml import Pipeline
# Evaluate model
#evaluator = BinaryClassificationEvaluator()
# Create ParamGrid for Cross Validation
#from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

#tree = DecisionTreeClassifier(labelCol="label", featuresCol="features", maxDep
th=5,
#                              minInstancesPerNode=20, impurity="gini")


#paramGrid = (ParamGridBuilder()
#             .addGrid(tree.maxDepth, [1, 2, 6, 10])
#             .addGrid(tree.minInstancesPerNode, [10, 20, 40])
#             .build())

# Create 5-fold CrossValidator
#cv = CrossValidator(estimator=tree, estimatorParamMaps=paramGrid, evaluator=ev
aluator, numFolds=5)

# Run cross validations
#cvModel = cv.fit(trainingDataBalanced.select("features","label"))
# Takes ~5 minutes
#print("numNodes = ", cvModel.bestModel.numNodes)
#print("depth = ", cvModel.bestModel.depth)
#print("depth = ", cvModel.bestModel.minInstancesPerNode)
```

In [36]:
```
from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.classification import DecisionTreeClassifier
evaluator = BinaryClassificationEvaluator()

tree = DecisionTreeClassifier(labelCol="label", featuresCol="features", impurit
y="gini")
paramGrid = (ParamGridBuilder()
             .addGrid(tree.maxDepth, [1, 2, 6, 10])
             .addGrid(tree.minInstancesPerNode, [10, 20, 40])
             .build())

# Create trainValidationSplit
tvs = TrainValidationSplit(estimator=tree,
                           estimatorParamMaps=paramGrid,
                           evaluator=evaluator,
                           # 80% of the data will be used for training, 20% for
 validation.
                           trainRatio=0.8)

# Run TrainValidationSplit, and choose the best set of parameters.
model = tvs.fit(trainingDataBalanced.select("features","label"))

# Make predictions on test data. model is the model with combination of paramet
ers
# that performed best.
#model.transform(testData)\
#    .select("features", "label", "prediction")\
#    .show()
```

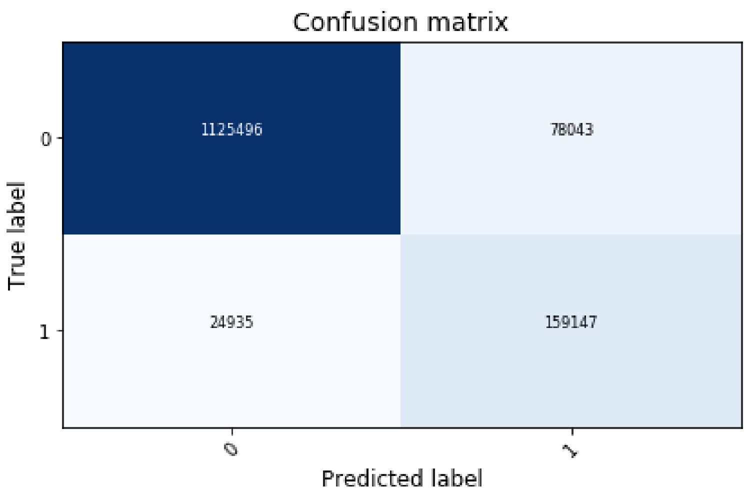
In [38]:
```
treeModel = model.bestModel
treeModel
```

Out[38]:
```
DecisionTreeClassificationModel (uid=DecisionTreeClassifier_4321b4fb87309d6687
f4) of depth 1 with 3 nodes
```

In [35]:
```python
# The not so pretty Spark-way
import matplotlib.pyplot as plt
import numpy as np

from pyspark.mllib.evaluation import MulticlassMetrics
from pyspark.mllib.util import MLUtils
predictions = treeModel.transform(testData).select('prediction','label')
#predictions.show(2)
mcMetrics = MulticlassMetrics(predictions.rdd)
a= mcMetrics.confusionMatrix().toArray().transpose()
a
```

Out[35]:
```
array([[1123990.,   32915.],
       [  61306.,  167014.]])
```

In [44]:
```python
# Pretty printing with Pandas and MatPlotLib
from pyspark.mllib.evaluation import MulticlassMetrics
from pyspark.mllib.util import MLUtils
predictions = treeModel.transform(testData).select('prediction','label')

true=predictions.select('label').toPandas() #Serializing to native Python (Pand
as) dataframe
predicted=predictions.select('prediction').toPandas()
pretty_print_conf_matrix(true, predicted, classes=[0,1],normalize=False,title=
'Confusion matrix',cmap=plt.cm.Blues)
```




```
      precision    recall  f1-score   N Obs

 0.0       0.98      0.94      0.96 1203539
 1.0       0.67      0.86      0.76  184082

avg        0.94      0.93      0.93 1387621
```

In [43]:
```
import matplotlib.pyplot as plt
import itertools
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
import numpy as np

def pretty_print_conf_matrix(y_true, y_pred,
                             classes,
                             normalize=False,
                             title='Confusion matrix',
                             cmap=plt.cm.Blues):
    """
    Mostly stolen from: http://scikit-learn.org/stable/auto_examples/model_sele
ction/plot_confusion_matrix.html#sphx-glr-auto-examples-model-selection-plot-co
nfusion-matrix-py

    Normalization changed, classification_report stats added below plot
    """

    cm = confusion_matrix(y_true, y_pred)
    #cm = confArray

    # Configure Confusion Matrix Plot Aesthetics (no text yet)
    plt.imshow(cm, interpolation='nearest', cmap=cmap, aspect='auto')
    plt.title(title, fontsize=14)
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)
    plt.ylabel('True label', fontsize=12)
    plt.xlabel('Predicted label', fontsize=12)

    # Calculate normalized values (so all cells sum to 1) if desired
    if normalize:
        cm = np.round(cm.astype('float') / cm.sum(),2) #(axis=1)[:, np.newaxis]

    # Place Numbers as Text on Confusion Matrix Plot
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 verticalalignment = 'bottom',
                 color="white" if cm[i, j] > thresh else "black",
                 fontsize=8)


    # Add Precision, Recall, F-1 Score as Captions Below Plot
    rpt = classification_report(y_true, y_pred)
    rpt = rpt.replace('avg / total', '      avg')
    rpt = rpt.replace('support', 'N Obs')

    plt.annotate(rpt,
                 xy = (0,0),
                 xytext = (-50, -200),
                 #xytext = (0, 0),
                 xycoords='axes fraction', textcoords='offset points',
                 fontsize=12, ha='left')

    # Plot
    plt.tight_layout()
```

Lets try an SVM for predicting cancellation

In [40]:
```
flightsWithDelayStatus.columns
```

```
Out[40]: ['Year',
          'Month',
          'DayofMonth',
          'DayOfWeek',
          'DepTime',
          'CRSDepTime',
          'ArrTime',
          'CRSArrTime',
          'UniqueCarrier',
          'FlightNum',
          'TailNum',
          'ActualElapsedTime',
          'CRSElapsedTime',
          'AirTime',
          'ArrDelay',
          'DepDelay',
          'Origin',
          'Dest',
          'Distance',
          'TaxiIn',
          'TaxiOut',
          'Cancelled',
          'CancellationCode',
          'Diverted',
          'CarrierDelay',
          'WeatherDelay',
          'NASDelay',
          'SecurityDelay',
          'LateAircraftDelay',
          'stationWBAN',
          'CallSign',
          'WBAN',
          'Tmax',
          'TmaxFlag',
          'Tmin',
          'TminFlag',
          'Tavg',
          'TavgFlag',
          'Depart',
          'DepartFlag',
          'DewPoint',
          'DewPointFlag',
          'WetBulb',
          'WetBulbFlag',
          'Heat',
          'HeatFlag',
          'Cool',
          'CoolFlag',
          'Sunrise',
          'SunriseFlag',
          'Sunset',
          'SunsetFlag',
          'CodeSum',
          'CodeSumFlag',
          'Depth',
          'DepthFlag',
          'Water1',
          'Water1Flag',
          'SnowFall',
          'SnowFallFlag',
          'PrecipTotal',
          'PrecipTotalFlag',
          'StnPressure',
          'StnPressureFlag',
          'SeaLevel',
          'SeaLevelFlag',
          'ResultSpeed',
          'ResultSpeedFlag',
```

```
 'ResultDir',
 'ResultDirFlag',
 'AvgSpeed',
 'AvgSpeedFlag',
 'Max5Speed',
 'Max5SpeedFlag',
 'Max5Dir',
 'Max5DirFlag',
 'Max2Speed',
 'Max2SpeedFlag',
 'Max2Dir',
 'Max2DirFlag',
 'prediction',
 'avgArrDelay',
 'minArrDelay',
 'maxArrDelay',
 'numberOfFlights',
 'delayGroup',
 'delayedStatus']
```

In [45]:

```
# https://spark.apache.org/docs/2.2.0/ml-features.html#rformula
from pyspark.ml.feature import RFormula
formula = RFormula(
    #formula="Cancelled ~ Tmax + TmaxFlag + Tmin + TminFlag + Tavg + Depart + DewPoint + WetBulb + Heat + Cool + Sunrise + Sunset + CodeSum  + Depth + SnowFall + SnowFallFlag + PrecipTotal + PrecipTotalFlag + StnPressure + SeaLevel + ResultSpeed + ResultDir + AvgSpeed + Max5Speed + Max5SpeedFlag + Max5Dir + Max2Speed + Max2SpeedFlag + Max2Dir",
    #featuresCol="features",
    formula="Cancelled ~ DepDelay + DepTime + Distance ",
    labelCol="label")

outputForSVM = formula.fit(flightsWithDelayStatus.na.fill(0).na.fill('None')).transform(flightsWithDelayStatus.na.fill(0).na.fill('None')).select("features", "label")
```

In [46]:

```
#output.limit(10).toPandas()
(trainingDataForSVM, testDataForSVM) = outputForSVM.randomSplit([0.8,0.2], seed = 13234 )
trainingDataForSVM.groupBy(col('Label')).count().show()
```

```
+-----+-------+
|Label|  count|
+-----+-------+
|  0.0|5442478|
|  1.0| 108016|
+-----+-------+
```

In [48]:

```
trainingDataForSVMBalanced = trainingDataForSVM.where(col('label')==0).sample(False, (108016/5442478), 42).unionAll(trainingDataForSVM.where(col('label')==1))
trainingDataForSVMBalanced.groupBy(col('Label')).count().show()
```

```
+-----+------+
|Label| count|
+-----+------+
|  0.0|107482|
|  1.0|108016|
+-----+------+
```

SVM relies on distance measures. Therefor we need to normalize the data before training, to avoid that large-scaled variables are "over-weighted"

In [47]:
```
trainingDataForSVMBalanced.columns
```

Out[47]: ['features', 'label']

In [52]:
```
# We need to stringidex the label columns ot make the LR behave like a classifi
er
from pyspark.ml.feature import StringIndexer
indexer = StringIndexer(inputCol="label", outputCol="indexedLabel")

from pyspark.ml.feature import Normalizer
# Normalize each Vector using $L^1$ norm.
normalizer = Normalizer(inputCol="features", outputCol="normalizedFeatures", p=
2.0)
normalizedBalancedTrainingData = normalizer.transform(trainingDataForSVMBalance
d).\
    drop("features").\
    withColumnRenamed("normalizedFeatures","features")
indexed = indexer.fit(normalizedBalancedTrainingData).transform(normalizedBalan
cedTrainingData).\
    drop("features").withColumnRenamed("indexedLabel","features")
normalizedBalancedTrainingData.limit(5).toPandas()
```

Out[52]:

	label	features
0	0.0	[-0.010499252701596863, 0.8317234096656296, 0....
1	0.0	[-0.009430697583540077, 0.6945035149021299, 0....
2	0.0	[-0.008439953257911402, 0.9350262502157561, 0....
3	0.0	[-0.0075040356671765935, 0.7087902780178619, 0...
4	0.0	[-0.017844372994792756, 0.9814405147136015, 0....

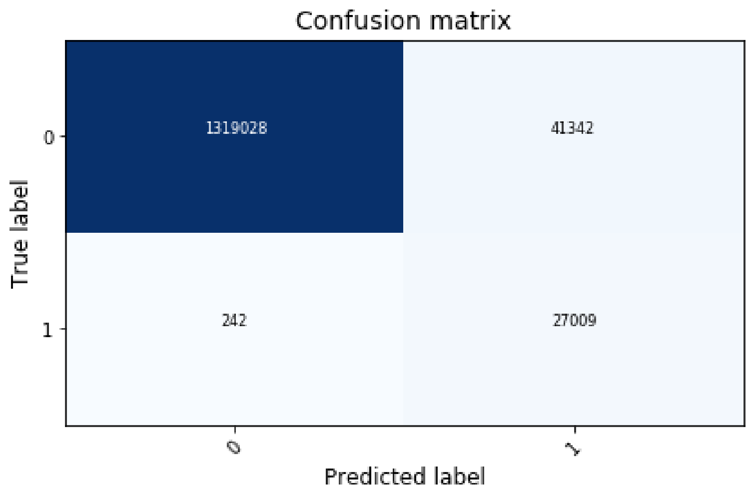
In [53]:
```
from pyspark.ml.classification import LogisticRegression
lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)
# Fit the model
lrModel = lr.fit(normalizedBalancedTrainingData)
```

In [56]:
```
normalizer = Normalizer(inputCol="features", outputCol="normalizedFeatures", p=
2.0)
normalizedTestData = normalizer.transform(testDataForSVM).drop("features").with
ColumnRenamed("normalizedFeatures","features")
#normalizedBalancedTestData.limit(5).toPandas()
# Predict, using the model
predictions = lrModel.transform(normalizedTestData)
predictions.groupby("prediction").count().toPandas()
```

Out[56]:

	prediction	count
0	0.0	1319270
1	1.0	68351

In [59]:
```
# Pretty printing with Pandas and MatPlotLib
#from pyspark.mllib.evaluation import MulticlassMetrics
#from pyspark.mllib.util import MLUtils
#predictions = treeModel.transform(testData).select('prediction','label')

true=predictions.select('label').toPandas() #Serializing to native Python (Pand
as) dataframe
predicted=predictions.select('prediction').toPandas()
pretty_print_conf_matrix(true, predicted, classes=[0,1],normalize=False,title=
'Confusion matrix',cmap=plt.cm.Blues)
```



	precision	recall	f1-score	N Obs
0.0	1.00	0.97	0.98	1360370
1.0	0.40	0.99	0.57	27251
avg	0.99	0.97	0.98	1387621