In [5]:
```
val df = spark.read
    .format("csv")
    .option("inferScema", "true")
    .option("header", "true")
    .load("../../../src/201508_station_data.csv")

df.show(2)
```

```
+----------+--------------------+---------+-----------+---------+--------+------------+
|station_id|                name|      lat|       long|dockcount|landmark|installation|
+----------+--------------------+---------+-----------+---------+--------+------------+
|         2|San Jose Diridon ...|37.329732|-121.901782|       27|San Jose|    8/6/2013|
|         3|San Jose Civic Ce...|37.330698|-121.888979|       15|San Jose|    8/5/2013|
+----------+--------------------+---------+-----------+---------+--------+------------+
only showing top 2 rows
```

Out[5]:
```
df: org.apache.spark.sql.DataFrame = [station_id: string, name: string ... 5 more fields]
```

Aggregation Functions

Display the nb of lines, of columns and the schema

In [6]:
```
println(df.count(), df.columns.size)
df.printSchema
```

```
(70,7)
root
 |-- station_id: string (nullable = true)
 |-- name: string (nullable = true)
 |-- lat: string (nullable = true)
 |-- long: string (nullable = true)
 |-- dockcount: string (nullable = true)
 |-- landmark: string (nullable = true)
 |-- installation: string (nullable = true)
```

Change type of various cols (lat & long to long, dockcount & station_id to int and installation to date)

In [7]:
```
import org.apache.spark.sql.types._

val df2 = df.withColumn("station_id", col("station_id").cast("int"))
    .withColumn("lat", col("lat").cast("long"))
    .withColumn("long", col("long").cast(LongType))
    .withColumn("dockcount", col("dockcount").cast("int"))
    .withColumn("installation", col("installation").cast(DateType))

df2.printSchema
```

```
root
 |-- station_id: integer (nullable = true)
 |-- name: string (nullable = true)
 |-- lat: long (nullable = true)
 |-- long: long (nullable = true)
 |-- dockcount: integer (nullable = true)
 |-- landmark: string (nullable = true)
 |-- installation: date (nullable = true)
```

Out[7]:
```
import org.apache.spark.sql.types._
df2: org.apache.spark.sql.DataFrame = [station_id: int, name: string ... 5 more fields]
```

val df4 = df.withColumn("station_id", col("station_id").cast("int")).select("*") will work if you want to change only one col

Alternative way with selectExpr

In [11]:
```
val df3 = df2.selectExpr("*", "cast(dockcount as long) dockcount_long").drop("dockcount")

df3.printSchema
```

```
root
 |-- station_id: integer (nullable = true)
 |-- name: string (nullable = true)
 |-- lat: long (nullable = true)
 |-- long: long (nullable = true)
 |-- landmark: string (nullable = true)
 |-- installation: date (nullable = true)
 |-- dockcount_long: long (nullable = true)
```

Out[11]:
```
df3: org.apache.spark.sql.DataFrame = [station_id: int, name: string ... 5 more fields]
```

Show the nb of lines for the column "landmark"

In [12]:
```
df2.select(count("landmark")).show()
```

```
+---------------+
|count(landmark)|
+---------------+
|             70|
+---------------+
```

Count distinct values of the landmark column

In [13]:
```
df2.select(countDistinct("landmark")).show()
```

```
+------------------------+
|count(DISTINCT landmark)|
+------------------------+
|                       5|
+------------------------+
```

Same thing but without returning a DF, but directly the result as a nb

In [14]:
```
df2.select("landmark").distinct().count()
```

Out[14]:
```
res10: Long = 5
```

You can do the same thing with SQL

In [15]:
```
import org.apache.spark.sql.functions._

df2.createOrReplaceTempView("dfTable")

spark.sql("""
SELECT count(distinct(landmark))
FROM dfTable
""").show()
```

```
+------------------------+
|count(DISTINCT landmark)|
+------------------------+
|                       5|
+------------------------+
```

Out[15]:
```
import org.apache.spark.sql.functions._
```

Count approximatively distinct values of the landmark column

In [16]:
```
df2.select(approx_count_distinct("landmark", 0.1)).show()
```

```
+-------------------------------+
|approx_count_distinct(landmark)|
+-------------------------------+
|                              5|
+-------------------------------+
```

As usual, it can also be done in an SQL fashion (the 2nd param can be omitted : the error is equal to 0.05 by defaut :

In [19]:
```
spark.sql("""
SELECT approx_count_distinct(landmark, 0.1)
FROM dfTable
""").show()
```

```
+------------------------------+
|approx_count_distinct(landmark)|
+------------------------------+
|                             5|
+------------------------------+
```

Now let's retrieve the first and last elements of a specific column:

In [20]:
```
df2.select(first("landmark"), last("name")).show()
```

```
+---------------------+-----------------+
|first(landmark, false)|last(name, false)|
+---------------------+-----------------+
|             San Jose|      Ryland Park|
+---------------------+-----------------+
```

and with an SQL query

In [21]:
```
spark.sql("""
SELECT first(landmark), last(name)
FROM dfTable
""").show()
```

```
+---------------------+-----------------+
|first(landmark, false)|last(name, false)|
+---------------------+-----------------+
|             San Jose|      Ryland Park|
+---------------------+-----------------+
```

In [22]:
```
// https://sparkbyexamples.com/spark/spark-change-dataframe-column-type/
```

You can also be interesting in learning the minimum & maximum of a column:

In [23]:
```
df2.select(min("station_id"), max("station_id")).show()
```

```
+---------------+---------------+
|min(station_id)|max(station_id)|
+---------------+---------------+
|              2|             84|
+---------------+---------------+
```

with a SQL query

In [24]:
```
spark.sql("""
SELECT min(station_id), max(station_id)
FROM dfTable
""").show()
```

```
+---------------+---------------+
|min(station_id)|max(station_id)|
+---------------+---------------+
|              2|             84|
+---------------+---------------+
```

Sum, min & avg of one or more columns

In [25]:
```
df2.select(sum("dockcount"), min("long"), avg("lat")).show()
```

```
+--------------+---------+--------+
|sum(dockcount)|min(long)|avg(lat)|
+--------------+---------+--------+
|          1236|     -122|    37.0|
+--------------+---------+--------+
```

In [26]:
```
spark.sql("""
SELECT min(dockcount), min(long), avg(lat)
FROM dfTable
""").show()
```

```
+--------------+---------+--------+
|min(dockcount)|min(long)|avg(lat)|
+--------------+---------+--------+
|            11|     -122|    37.0|
+--------------+---------+--------+
```

Let's compare the sum of all values with the sum of the distinct values :

In [27]:
```
df2.select(sumDistinct("dockcount"), sum("dockcount")).show()
```

```
+----------------------+--------------+
|sum(DISTINCT dockcount)|sum(dockcount)|
+----------------------+--------------+
|                   120|          1236|
+----------------------+--------------+
```

As always it can be also calculated with a SQL statement :

In [28]:
```
spark.sql("""
SELECT sum(distinct(dockcount)), sum(dockcount)
FROM dfTable
""").show()
```

```
+----------------------+--------------+
|sum(DISTINCT dockcount)|sum(dockcount)|
+----------------------+--------------+
|                   120|          1236|
+----------------------+--------------+
```

Avg calculated with selectExpr (side note : it'll fail with only one "selectExpr") :

In [30]:
```
df2.selectExpr("avg(dockcount) as AVG",
               "sum(dockcount) as TOTAL",
               "count(dockcount) as NB").selectExpr("*", "TOTAL/NB as CALCULATED_AVG").show()
```

```
+-----------------+-----+---+-----------------+
|              AVG|TOTAL| NB|   CALCULATED_AVG|
+-----------------+-----+---+-----------------+
|17.65714285714286| 1236| 70|17.65714285714286|
+-----------------+-----+---+-----------------+
```

and with a nested SQL query

In [31]:
```
spark.sql("""
SELECT *, tot/nb as calculated_avg
FROM (
    SELECT avg(dockcount) AS avg_, sum(dockcount) AS tot, count(dockcount) as nb
    FROM dfTable
    )
""").show()
```

```
+-----------------+----+---+-----------------+
|             avg_| tot| nb|   calculated_avg|
+-----------------+----+---+-----------------+
|17.65714285714286|1236| 70|17.65714285714286|
+-----------------+----+---+-----------------+
```

Variance & standard deviation :

The variance is the average of the squared differences from the mean, and the standard deviation is the square root of the variance. You can calculate these in Spark by using their respective functions.

Spark has both the formula for the sample standard deviation as well as the formula for the population standard deviation. These are fundamentally different statistical formulae, and we need to differentiate between them. By default, Spark performs the formula for the sample standard deviation or variance if you use the variance or stddev functions.

You can also specify these explicitly or refer to the population standard deviation or variance:

In [35]:
```
df2.select(stddev("dockcount"), variance("dockcount")).show()
```

```
+---------------------+------------------+
|stddev_samp(dockcount)|var_samp(dockcount)|
+---------------------+------------------+
|     4.010441857493954| 16.083643892339555|
+---------------------+------------------+
```

In [32]:
```
df2.select(var_pop("dockcount"), var_samp("dockcount"), stddev_pop("dockcount"), stddev_samp("dockcount")).sh
```

```
+------------------+------------------+--------------------+----------------------+
|var_pop(dockcount)|var_samp(dockcount)|stddev_pop(dockcount)|stddev_samp(dockcount)|
+------------------+------------------+--------------------+----------------------+
| 15.85387755102042| 16.083643892339555|    3.981692799679606|     4.010441857493954|
+------------------+------------------+--------------------+----------------------+
```

Skewness & kurtosis

A fundamental task in many statistical analyses is to characterize the location and variability of a data set. A further characterization of the data includes skewness and kurtosis. Skewness is a measure of symmetry, or more precisely, the lack of symmetry. A distribution, or data set, is symmetric if it looks the same to the left and right of the center point.

Kurtosis is a measure of whether the data are heavy-tailed or light-tailed relative to a normal distribution. That is, data sets with high kurtosis tend to have heavy tails, or outliers. Data sets with low kurtosis tend to have light tails, or lack of outliers. A uniform distribution would be the extreme case.

source Engineering Statistics Handbook (https://www.itl.nist.gov/div898/handbook/eda/section3/eda35b.htm#:~:text=Skewness%20is%20a%20measure%20of,relative%20to%20a%20norm

An other interseting blog post on medium : https://codeburst.io/2-important-statistics-terms-you-need-to-know-in-data-science-skewness-and-kurtosis-388fef94eeaa?gi=cefd32e14a26 (https://codeburst.io/2-important-statistics-terms-you-need-to-know-in-data-science-skewness-and-kurtosis-388fef94eeaa?gi=cefd32e14a26)

In [36]:
```
df2.select(skewness("dockcount"), kurtosis("dockcount")).show()
```

```
+------------------+-------------------+
|skewness(dockcount)| kurtosis(dockcount)|
+------------------+-------------------+
| 0.7369248035176993|-0.13950371148566587|
+------------------+-------------------+
```

In [37]:
```
df2.select(skewness("station_id"), kurtosis("station_id")).show()
```

```
+-------------------+-------------------+
|skewness(station_id)|kurtosis(station_id)|
+-------------------+-------------------+
|-0.06719168007171215| -1.1955050641926341|
+-------------------+-------------------+
```

Correlation

In [38]:
```
df2.select(corr("station_id", "dockcount")).show()
// sidenote: it doesn't make sense to use the correlation with the station_id
```

```
+--------------------------+
|corr(station_id, dockcount)|
+--------------------------+
|        0.24015841145323474|
+--------------------------+
```

Covariance

In probability theory and statistics, the mathematical concepts of covariance and correlation are very similar.[1][2] Both describe the degree to which two random variables or sets of random variables tend to deviate from their expected values in similar ways.

Source Wikipedia (https://en.wikipedia.org/wiki/Covariance_and_correlation)

“Covariance” indicates the direction of the linear relationship between variables. “Correlation” on the other hand measures both the strength and direction of the linear relationship between two variables. Correlation is a function of the covariance. What sets them apart is the fact that correlation values are standardized whereas, covariance values are not. You can obtain the correlation coefficient of two variables by dividing the

covariance of these variables by the product of the standard deviations of the same values. If we revisit the definition of Standard Deviation, it essentially measures the absolute variability of a datasets' distribution. When you divide the covariance values by the standard deviation, it essentially scales the value down to a limited range of -1 to +1. This is precisely the range of the correlation values.

Source [Towards Datascience (https://towardsdatascience.com/let-us-understand-the-correlation-matrix-and-covariance-matrix-d42e6b643c22)](https://towardsdatascience.com/let-us-understand-the-correlation-matrix-and-covariance-matrix-d42e6b643c22)

In [39]:
```
df2.select(covar_pop("station_id", "dockcount")).show()
```

```
+-------------------------------+
|covar_pop(station_id, dockcount)|
+-------------------------------+
|              22.942857142857182|
+-------------------------------+
```

In [40]:
```
df2.select(covar_pop("station_id", "dockcount"), covar_samp("station_id", "dockcount")).show()
```

```
+-------------------------------+-------------------------------+
|covar_pop(station_id, dockcount)|covar_samp(station_id, dockcount)|
+-------------------------------+-------------------------------+
|              22.942857142857182|               23.27536231884062|
+-------------------------------+-------------------------------+
```

Aggregating to Complex Types

We can retrieve a set and a list of all values of a specific column :

In [41]:
```
df2.agg(collect_set("landmark"), collect_list("landmark")).show()
```

```
+--------------------+---------------------+
|collect_set(landmark)|collect_list(landmark)|
+--------------------+---------------------+
| [San Jose, San Fr...|  [San Jose, San Jo...|
+--------------------+---------------------+
```

Now what is the first element of both two results :

In [42]:
```
df2.agg(collect_set("landmark")).first()
```

Out[42]:
```
res38: org.apache.spark.sql.Row = [WrappedArray(San Jose, San Francisco, Palo Alto, Redwood City, Mountain
View)]
```

In [49]:
```
df2.agg(collect_list("landmark")).first()(0)
// (0) otherwise one get the Row
```

Out[49]:
```
res45: Any = WrappedArray(San Jose, San Jose, San Jose, San Jose, San Jose, San Jose, San Jose, San Jose, S
an Jose, San Jose, San Jose, San Jose, San Jose, San Jose, Redwood City, Redwood City, Redwood City, Redwoo
d City, Redwood City, Redwood City, Mountain View, Mountain View, Mountain View, Mountain View, Mountain Vi
ew, Mountain View, Mountain View, Palo Alto, Palo Alto, Palo Alto, Palo Alto, Palo Alto, San Francisco, San
Francisco, San Francisco, San Francisco, San Francisco, San Francisco, San Francisco, San Francisco, San Fr
ancisco, San Francisco, San Francisco, San Francisco, San Francisco, San Francisco, San Francisco, San Fran
cisco, San Francisco, San Francisco, San Francisco, San Francisco, San Francisco, San Francisco, San Franci
sco, San Francisco, San Francisco, San Francisc...
```

Grouping

First, let's use expression to calculate the sum of dockcount for each group of landmark, and sort by descending order the results :

In [50]:
```
df2.groupBy("landmark").agg(sum("dockcount")).sort(desc("sum(dockcount)")).show(10)
```

```
+-------------+--------------+
|     landmark|sum(dockcount)|
+-------------+--------------+
|San Francisco|           665|
|     San Jose|           264|
|Mountain View|           117|
| Redwood City|           115|
|    Palo Alto|            75|
+-------------+--------------+
```

Same thing by grouping with two cols : landmark & name :

In [51]:
```
df2.groupBy("landmark", "name").agg(sum("dockcount")).orderBy(desc("sum(dockcount)")).show(10)
/*
df2.groupBy("landmark", "name").agg(sum("dockcount")).sort(desc("sum(dockcount)")).show(10)
same thing with sort but in SQL only ORDER BY works ! */
```

```
+-------------+--------------------+--------------+
|     landmark|                name|sum(dockcount)|
+-------------+--------------------+--------------+
|San Francisco|   Market at Sansome|            27|
|San Francisco|      Market at 10th|            27|
|     San Jose|San Jose Diridon ...|            27|
|San Francisco|     2nd at Townsend|            27|
| Redwood City|Redwood City Calt...|            25|
|San Francisco| Golden Gate at Polk|            23|
|San Francisco|   Steuart at Market|            23|
|Mountain View|Mountain View Cal...|            23|
|    Palo Alto|Palo Alto Caltrai...|            23|
|Mountain View|San Antonio Caltr...|            23|
+-------------+--------------------+--------------+
only showing top 10 rows
```

Same thing in SQL

In [52]:
```
spark.sql("""
SELECT landmark, name, sum(dockcount) as Total
FROM dfTable
GROUP BY landmark, name
ORDER BY total DESC
""").show(10)
```

```
+-------------+--------------------+-----+
|     landmark|                name|Total|
+-------------+--------------------+-----+
|San Francisco|     2nd at Townsend|   27|
|San Francisco|   Market at Sansome|   27|
|     San Jose|San Jose Diridon ...|   27|
|San Francisco|      Market at 10th|   27|
| Redwood City|Redwood City Calt...|   25|
|San Francisco|Harry Bridges Pla...|   23|
|Mountain View|Mountain View Cal...|   23|
|Mountain View|San Antonio Caltr...|   23|
|San Francisco|   Steuart at Market|   23|
|    Palo Alto|Palo Alto Caltrai...|   23|
+-------------+--------------------+-----+
only showing top 10 rows
```

An other way to make group is by using maps :

In [54]:
```
df.groupBy("landmark").agg("dockcount" -> "avg", "dockcount" -> "stddev_pop").show()
```

```
+-------------+------------------+--------------------+
|     landmark|    avg(dockcount)|stddev_pop(dockcount)|
+-------------+------------------+--------------------+
|    Palo Alto|              15.0|    4.381780460041329|
|San Francisco|              19.0|    3.703280399090206|
|     San Jose|              16.5|    3.427827300200522|
| Redwood City|16.428571428571427|   3.4992710611188254|
|Mountain View|16.714285714285715|    4.199125273342591|
+-------------+------------------+--------------------+
```

Window Functions

In [102]:
```
val simpleData = Seq(("James", "Sales", 3000),
    ("Michael", "Sales", 4600),
    ("Robert", "Sales", 4100),
    ("Maria", "Finance", 3000),
    ("James", "Sales", 3000),
    ("Scott", "Finance", 3300),
    ("Jen", "Finance", 3900),
    ("Jeff", "Marketing", 3000),
    ("Kumar", "Marketing", 2000),
    ("Saif", "Sales", 4100)
  )

val df = simpleData.toDF("employee_name", "department", "salary")
df.show()
```

```
+-------------+----------+------+
|employee_name|department|salary|
+-------------+----------+------+
|        James|     Sales|  3000|
|      Michael|     Sales|  4600|
|       Robert|     Sales|  4100|
|        Maria|   Finance|  3000|
|        James|     Sales|  3000|
|        Scott|   Finance|  3300|
|          Jen|   Finance|  3900|
|         Jeff| Marketing|  3000|
|        Kumar| Marketing|  2000|
|         Saif|     Sales|  4100|
+-------------+----------+------+
```

Out[102]:
```
simpleData: Seq[(String, String, Int)] = List((James,Sales,3000), (Michael,Sales,4600), (Robert,Sales,410
0), (Maria,Finance,3000), (James,Sales,3000), (Scott,Finance,3300), (Jen,Finance,3900), (Jeff,Marketing,300
0), (Kumar,Marketing,2000), (Saif,Sales,4100))
df: org.apache.spark.sql.DataFrame = [employee_name: string, department: string ... 1 more field]
```

row_number() window function is used to give the sequential row number starting from 1 to the result of each window partition.

In [103]:
```
val windowSpec  = Window.partitionBy("department").orderBy("salary")

df.withColumn("row_number",row_number().over(windowSpec)).show()
```

```
+-------------+----------+------+----------+
|employee_name|department|salary|row_number|
+-------------+----------+------+----------+
|        James|     Sales|  3000|         1|
|        James|     Sales|  3000|         2|
|       Robert|     Sales|  4100|         3|
|         Saif|     Sales|  4100|         4|
|      Michael|     Sales|  4600|         5|
|        Maria|   Finance|  3000|         1|
|        Scott|   Finance|  3300|         2|
|          Jen|   Finance|  3900|         3|
|        Kumar| Marketing|  2000|         1|
|         Jeff| Marketing|  3000|         2|
+-------------+----------+------+----------+
```

Out[103]:
```
windowSpec: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@459ce
8d3
```

dense_rank() window function is used to get the result with rank of rows within a window partition without any gaps. This is similar to rank() function difference being rank function leaves gaps in rank when there are ties.

In [104]:
```
df.withColumn("dense_rank",dense_rank().over(windowSpec)).show()
```

```
+-------------+----------+------+----------+
|employee_name|department|salary|dense_rank|
+-------------+----------+------+----------+
|        James|     Sales|  3000|         1|
|        James|     Sales|  3000|         1|
|       Robert|     Sales|  4100|         2|
|         Saif|     Sales|  4100|         2|
|      Michael|     Sales|  4600|         3|
|        Maria|   Finance|  3000|         1|
|        Scott|   Finance|  3300|         2|
|          Jen|   Finance|  3900|         3|
|        Kumar| Marketing|  2000|         1|
|         Jeff| Marketing|  3000|         2|
+-------------+----------+------+----------+
```

Grouping Sets

an aggregation across multiple groups. We achieve this by using grouping sets. Grouping sets are a low-level tool for combining sets of aggregations together. They give you the ability to create arbitrary aggregation in their group-by statements.

Reference : https://mungingdata.com/apache-spark/aggregations/ (https://mungingdata.com/apache-spark/aggregations/)

Cube

A cube takes the rollup to a level deeper. Rather than treating elements hierarchically, a cube does the same thing across all dimensions. This means that it won't just go by column over the entire possibilities, but also the other cols.

In [84]:
```
val df = Seq(
  ("bar", 2L),
  ("bar", 2L),
  ("foo", 1L),
  ("foo", 2L)
).toDF("word", "num")

df.show()
```

```
+----+---+
|word|num|
+----+---+
| bar|  2|
| bar|  2|
| foo|  1|
| foo|  2|
+----+---+
```

Out[84]:
```
df: org.apache.spark.sql.DataFrame = [word: string, num: bigint]
```

In [85]:
```
df.cube("word", "num").agg(count("num")).show()
```

```
+----+----+----------+
|word| num|count(num)|
+----+----+----------+
| bar|   2|         2|
|null|null|         4|
| foo|   2|         1|
|null|   1|         1|
| foo|null|         2|
| foo|   1|         1|
|null|   2|         3|
| bar|null|         2|
+----+----+----------+
```

In [ ]:
```
+----+----+-----+
|word| num|count|
+----+----+-----+
|null|null|    4| Total rows in df
|null|   1|    1| Count where num equals 1
|null|   2|    3| Count where num equals 2
| bar|null|    2| Where word equals bar
| bar|   2|    2| Where word equals bar and num equals 2
| foo|null|    2| Where word equals foo
| foo|   1|    1| Where word equals foo and num equals 1
| foo|   2|    1| Where word equals foo and num equals 2
+----+----+-----+
```

Rollups

A rollup is a multidimensional aggregation that performs a variety of group-by style calculations for us. It's a subset of cube that "computes hierarchical subtotals from left to right"

In [88]:
```
df.rollup("word", "num")
  .count()
  .sort(asc("word"), asc("num"))
  .show()
```

```
+----+----+-----+
|word| num|count|
+----+----+-----+
|null|null|    4|
| bar|null|    2|
| bar|   2|    2|
| foo|null|    2|
| foo|   1|    1|
| foo|   2|    1|
+----+----+-----+
```

In [ ]:
```
+----+----+-----+
|word| num|count|
+----+----+-----+
|null|null|    4| Count of all rows
| bar|null|    2| Count when word is bar
| bar|   2|    2| Count when num is 2
| foo|null|    2| Count when word is foo
| foo|   1|    1| When word is foo and num is 1
| foo|   2|    1| When word is foo and num is 2
+----+----+-----+s
```

rollup() returns a subset of the rows returned by cube(). rollup returns 6 rows whereas cube returns 8 rows. Here are the missing rows.

In [ ]:
```
+----+----+-----+
|word| num|count|
+----+----+-----+
|null|   1|    1| Word is null and num is 1
|null|   2|    3| Word is null and num is 2
+----+----+-----+
```

rollup("word", "num") doesn't return the counts when only word is null.

Let's switch around the order of the arguments passed to rollup and view the difference in the results.

In [89]:
```
df.rollup("num", "word")
  .count()
  .sort(asc("word"), asc("num"))
  .select("word", "num", "count")
  .show()
```

```
+----+----+-----+
|word| num|count|
+----+----+-----+
|null|null|    4|
|null|   1|    1|
|null|   2|    3|
| bar|   2|    2|
| foo|   1|    1|
| foo|   2|    1|
+----+----+-----+
```

Here are the rows missing from rollup("num", "word") compared to cube(" $word$ ","num").

In [ ]:
```
+----+----+-----+
|word| num|count|
+----+----+-----+
| bar|null|    2| Word equals bar and num is null
| foo|null|    2| Word equals foo and num is null
+----+----+-----+
```

rollup("num", "word") doesn't return the counts when only num is null.

Pivot

Pivots make it possible for you to convert a row into a column. For example, in our current data we have a Country column. With a pivot, we can aggregate according to some function for each of those given countries and display them in an easy-to-query way:

In [90]:
```
val data = Seq(("Banana",1000,"USA"), ("Carrots",1500,"USA"), ("Beans",1600,"USA"),
      ("Orange",2000,"USA"),("Orange",2000,"USA"),("Banana",400,"China"),
      ("Carrots",1200,"China"),("Beans",1500,"China"),("Orange",4000,"China"),
      ("Banana",2000,"Canada"),("Carrots",2000,"Canada"),("Beans",2000,"Mexico"))

val df = data.toDF("Product","Amount","Country")
df.show()
```

```
+-------+------+-------+
|Product|Amount|Country|
+-------+------+-------+
| Banana|  1000|    USA|
|Carrots|  1500|    USA|
|  Beans|  1600|    USA|
| Orange|  2000|    USA|
| Orange|  2000|    USA|
| Banana|   400|  China|
|Carrots|  1200|  China|
|  Beans|  1500|  China|
| Orange|  4000|  China|
| Banana|  2000| Canada|
|Carrots|  2000| Canada|
|  Beans|  2000| Mexico|
+-------+------+-------+
```

Out[90]:
```
data: Seq[(String, Int, String)] = List((Banana,1000,USA), (Carrots,1500,USA), (Beans,1600,USA), (Orange,20
00,USA), (Orange,2000,USA), (Banana,400,China), (Carrots,1200,China), (Beans,1500,China), (Orange,4000,Chin
a), (Banana,2000,Canada), (Carrots,2000,Canada), (Beans,2000,Mexico))
df: org.apache.spark.sql.DataFrame = [Product: string, Amount: int ... 1 more field]
```

Source : https://sparkbyexamples.com/spark/how-to-pivot-table-and-unpivot-a-spark-dataframe/ (https://sparkbyexamples.com/spark/how-to-pivot-table-and-unpivot-a-spark-dataframe/)

Spark SQL provides pivot function to rotate the data from one column into multiple columns. It is an aggregation where one of the grouping columns values transposed into individual columns with distinct data. To get the total amount exported to each country of each product, will do group by Product, pivot by Country, and the sum of Amount.

This will transpose the countries from DataFrame rows into columns and produces below output. where ever data is not present, it represents as null by default.

In [91]:
```
val pivotDF = df.groupBy("Product").pivot("Country").sum("Amount")
pivotDF.show()
```

```
+-------+------+-----+------+----+
|Product|Canada|China|Mexico| USA|
+-------+------+-----+------+----+
| Orange|  null| 4000|  null|4000|
|  Beans|  null| 1500|  2000|1600|
| Banana|  2000|  400|  null|1000|
|Carrots|  2000| 1200|  null|1500|
+-------+------+-----+------+----+
```

Out[91]:
```
pivotDF: org.apache.spark.sql.DataFrame = [Product: string, Canada: bigint ... 3 more fields]
```

Unpivot is a reverse operation, we can achieve by rotating column values into rows values. Spark SQL doesn't have unpivot function hence will use the stack() function. Below code converts column countries to row.

In [96]:
```
val unPivotDF = pivotDF.select(col("Product"),
    expr("stack(3, 'Canada', Canada, 'China', China, 'Mexico', Mexico) as (Country,Total)"))
    .where("Total is not null")
unPivotDF.show()
```

```
+-------+-------+-----+
|Product|Country|Total|
+-------+-------+-----+
| Orange|  China| 4000|
|  Beans|  China| 1500|
|  Beans| Mexico| 2000|
| Banana| Canada| 2000|
| Banana|  China|  400|
|Carrots| Canada| 2000|
|Carrots|  China| 1200|
+-------+-------+-----+
```

Out[96]:
```
unPivotDF: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [Product: string, Country: string ... 1
more field]
```

User-Defined Aggregation Functions

User-defined aggregation functions (UDAFs) are a way for users to define their own aggregation functions based on custom formulae or business rules. You can use UDAFs to compute custom calculations over groups of input data (as opposed to single rows). Spark maintains a single AggregationBuffer to store intermediate results for every group of input data.

To create a UDAF, you must inherit from the UserDefinedAggregateFunction base class and implement the following methods:

- inputSchema represents input arguments as a StructType
- bufferSchema represents intermediate UDAF results as a StructType
- dataType represents the return DataType
- deterministic is a Boolean value that specifies whether this UDAF will return the same result for a given input
- initialize allows you to initialize values of an aggregation buffer
- update describes how you should update the internal buffer based on a given row
- merge describes how two aggregation buffers should be merged
- evaluate will generate the final result of the aggregation