# Using New York City Taxi Data to illistrate using Arkouda with Pandas/NumPy

This notebook shows some examples of how to interoperate between Pandas and Arkouda at a small scale to allow it to be run on a 16GB laptop. Remember, Arkouda is not trying to replace Pandas but to allow for some Pandas-style operation at a much larger scale. In our experience Pandas can handle dataframes up to about **500 million rows** before performance becomes a real issue, this is provided that you run on a sufficently capable compute server. Arkouda breaks the shared memory paradigm and scales its operations to dataframes with over **200 billion rows**, maybe even a trillion. In practice we have run Arkouda server operations on columns of one trillion elements running on 512 compute nodes. This yielded a **>20TB dataframe** in Arkouda.

- Import Arkouda package and connect to the Arkouda server
- import other useful packages
- Define some python helper functions for ETL (Extract/Transform/Load)
- Define a python function to transfer dataframes from Pandas to Arkouda
- Read NYC taxi csv into Pandas
- Put dataframe columns into the Arkouda server
- Compute taxi ride duration in Pandas and in Arkouda and histogram data
- Read NYC Taxi Zone Lookup Table (tzlut) into Pandas
- Transfer tzlut to Arkouda
- Compute something with Groupby/aggregate in Pandas and in Arkouda
    - Groupby on pickup and dropoff location ids
    - use groupby/aggregate on edge list to compute different things
        - min/max/mean/std distance between location ids
        - min/max/mean/std time between location ids
        - number of trips between location ids
        - other things
    -
    - other things
- model number of taxis at a given time
- model taxis as specific entities (Kalman filter?)
    - use time and location ids
    - probability of paths of taxis
    - ...
- other things?

## New York City Taxi Data

[Yellow Trips Data Dictionary (https://www1.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_yellow.pdf)](https://www1.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_yellow.pdf)

[NYC Yellow Taxi Trip Records Jan 2020 (https://s3.amazonaws.com/nyc-tlc/trip+data/yellow_tripdata_2020-01.csv)](https://s3.amazonaws.com/nyc-tlc/trip+data/yellow_tripdata_2020-01.csv)

[Green Trips Data Dictionary (https://www1.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_green.pdf)](https://www1.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_green.pdf)

NYC Green Taxi Trip Records Jan 2020 (https://s3.amazonaws.com/nyc-tlc/trip+data/green_tripdata_2020-01.csv)

NYC Taxi Zone Lookup Table (https://s3.amazonaws.com/nyc-tlc/misc/taxi+_zone_lookup.csv)

### Import Arkouda package and connect to the server

In [1]:

```
import arkouda as ak
ak.connect(connect_url="tcp://localhost:5555")
```

### Import the other packages

In [2]:

```
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
import gc
```

### Conversion functions for ETL

In [3]:

```
# conversion from csv field to int64
# try to convert to int, on exception (empty or other string) convert
 to 0
def cvt_to_int64(v):
    try:
        return np.int64(v)
    except:
        return np.int64(0)

# conversion from csv field to string
# try to convert to int, on exception (empty or other string) convert
 to 0
def cvt_to_string(v):
    try:
        if v == '':
            return 'N/A'
        else:
            return str(v)
    except:
        return 'N/A'

# conversion from csv field (Y,N,empty) to bool
# on Y convert to True, on N or empty convert to False
def cvt_YN_to_bool(v):
    if v == 'Y':
        return True
    else:
        return False
```

## Define function to transfer dataframe to dictionary of Arkouda arrays

In [4]:

```
# check all objects in iterable for instance of str
# this is a crutch to get a pandas column of str into Arkouda
def is_all_str(a):
    ret = True
    for v in a:
        if isinstance(v, str):
            ret = True
        else:
            ret = False
            break
    return ret

# put data frame columns into arkouda server and return a dict of the
 pdarrays
# convert some columns into data types the server can understand
def ak_create_akdict_from_df(df):
    akdict = {}
    for cname in df.keys():
        a = df[cname].values

        # int64, float64, and np.bool should go over fine
        if a.dtype in [np.int64, np.float64, np.bool]:
            akdict[cname] = ak.array(a)
            print(cname, " : ", a.dtype, "->", a.dtype)
        # time needs to be converted to int64
        elif a.dtype in ["datetime64[ns]"]:
            akdict[cname] = ak.array(a.astype(np.int64))
            print(cname, " : ", a.dtype, "->", akdict[cname].dtype)
        # convert to arkouda Strings object if whole column is instanc
e of str
        elif is_all_str(a):
            # convert to python list of str then ak.array can convert
 to ak.Strings object
            akdict[cname] = ak.array(list(a))
            print(cname, " : ", a.dtype, "->", 'ak.Strings')
        # something I don't understand how to convert to a server data
type
        else:
            print("don't know how to convert ", a.dtype, " !!!")
    return akdict
```

## Helper functions

In [5]:

```
def ns_to_min(v):
    return (v / (1e9 * 60.0))

def min_to_ns(v):
    return (int(v * 1_000_000_000 * 60))
```

# Yellow taxi trip data

## Read in the data

In [6]:
```
# Read in yellow taxi data
# per yellow data dictionary convert to data types Arkouda can handle
# int64, float64, bool
cvt = {'VendorID': cvt_to_int64, 'passenger_count': cvt_to_int64, 'RatecodeID': cvt_to_int64,
       'store_and_fwd_flag': cvt_YN_to_bool,
       'PULocationID': cvt_to_int64, 'DOLocationID':cvt_to_int64, 'payment_type': cvt_to_int64}
# explicitly parse date-time fields
parse_dates_lst = ['tpep_pickup_datetime','tpep_dropoff_datetime']
# call read_csv to parse data with these options
ydf = pd.read_csv("../Downloads/yellow_tripdata_2020-01.csv",
                  converters=cvt, header=0, low_memory=False,
                  parse_dates=parse_dates_lst, infer_datetime_format=True)
```

## Print out the dataframe

In [7]:
```
#print out dataframe
ydf
```

Out[7]:

| | VendorID | tpep_pickup_datetime | tpep_dropoff_datetime | passenger_count | trip_distance |
|---|---|---|---|---|---|
| **0** | 1 | 2020-01-01 00:28:15 | 2020-01-01 00:33:03 | 1 | 1.20 |
| **1** | 1 | 2020-01-01 00:35:39 | 2020-01-01 00:43:04 | 1 | 1.20 |
| **2** | 1 | 2020-01-01 00:47:41 | 2020-01-01 00:53:52 | 1 | 0.60 |
| **3** | 1 | 2020-01-01 00:55:23 | 2020-01-01 01:00:14 | 1 | 0.80 |
| **4** | 2 | 2020-01-01 00:01:58 | 2020-01-01 00:04:16 | 1 | 0.00 |
| **...** | ... | ... | ... | ... | ... |
| **6405003** | 0 | 2020-01-31 22:51:00 | 2020-01-31 23:22:00 | 0 | 3.24 |
| **6405004** | 0 | 2020-01-31 22:10:00 | 2020-01-31 23:26:00 | 0 | 22.13 |
| **6405005** | 0 | 2020-01-31 22:50:07 | 2020-01-31 23:17:57 | 0 | 10.51 |
| **6405006** | 0 | 2020-01-31 22:25:53 | 2020-01-31 22:48:32 | 0 | 5.49 |
| **6405007** | 0 | 2020-01-31 22:44:00 | 2020-01-31 23:06:00 | 0 | 11.60 |

6405008 rows × 18 columns

## Which columns did we read in?

```
In [8]: # see which keys we read in from first line of csv data file
        #print(ydf.keys())
        print(ydf.columns)
```

```
Index(['VendorID', 'tpep_pickup_datetime', 'tpep_dropoff_datetime',
       'passenger_count', 'trip_distance', 'RatecodeID', 'store_and_fw
d_flag',
       'PULocationID', 'DOLocationID', 'payment_type', 'fare_amount',
'extra',
       'mta_tax', 'tip_amount', 'tolls_amount', 'improvement_surcharg
e',
       'total_amount', 'congestion_surcharge'],
      dtype='object')
```

## Convert the dataframe to a dictionary of Arkouda arrays

```
In [9]: # put data frame columns into arkouda server
        # convert some columns into data types the server can understand
        akdict = ak_create_akdict_from_df(ydf)
```

```
VendorID  :  int64 -> int64
tpep_pickup_datetime  :  datetime64[ns] -> int64
tpep_dropoff_datetime  :  datetime64[ns] -> int64
passenger_count  :  int64 -> int64
trip_distance  :  float64 -> float64
RatecodeID  :  int64 -> int64
store_and_fwd_flag  :  bool -> bool
PULocationID  :  int64 -> int64
DOLocationID  :  int64 -> int64
payment_type  :  int64 -> int64
fare_amount  :  float64 -> float64
extra  :  float64 -> float64
mta_tax  :  float64 -> float64
tip_amount  :  float64 -> float64
tolls_amount  :  float64 -> float64
improvement_surcharge  :  float64 -> float64
total_amount  :  float64 -> float64
congestion_surcharge  :  float64 -> float64
```

## Show which columns got transfered to the Arkouda server

In [10]:
```
# which keys made it over to the server
print(akdict.keys())
```

```
dict_keys(['VendorID', 'tpep_pickup_datetime', 'tpep_dropoff_datetim
e', 'passenger_count', 'trip_distance', 'RatecodeID', 'store_and_fwd_f
lag', 'PULocationID', 'DOLocationID', 'payment_type', 'fare_amount',
'extra', 'mta_tax', 'tip_amount', 'tolls_amount', 'improvement_surchar
ge', 'total_amount', 'congestion_surcharge'])
```

## Look at the symbol table of the Arkouda server

In [11]:
```
# print out the arkouda server symbol table
print(ak.info(ak.AllSymbols))
```

```
name:"id_61" dtype:"int64" size:6405008 ndim:1 shape:(6405008) itemsiz
e:8
name:"id_60" dtype:"int64" size:6405008 ndim:1 shape:(6405008) itemsiz
e:8
name:"id_63" dtype:"int64" size:6405008 ndim:1 shape:(6405008) itemsiz
e:8
name:"id_62" dtype:"float64" size:6405008 ndim:1 shape:(6405008) items
ize:8
name:"id_65" dtype:"int64" size:6405008 ndim:1 shape:(6405008) itemsiz
e:8
name:"id_64" dtype:"bool" size:6405008 ndim:1 shape:(6405008) itemsiz
e:1
name:"id_67" dtype:"int64" size:6405008 ndim:1 shape:(6405008) itemsiz
e:8
name:"id_66" dtype:"int64" size:6405008 ndim:1 shape:(6405008) itemsiz
e:8
name:"id_69" dtype:"float64" size:6405008 ndim:1 shape:(6405008) items
ize:8
name:"id_68" dtype:"float64" size:6405008 ndim:1 shape:(6405008) items
ize:8
name:"id_70" dtype:"float64" size:6405008 ndim:1 shape:(6405008) items
ize:8
name:"id_71" dtype:"float64" size:6405008 ndim:1 shape:(6405008) items
ize:8
name:"id_72" dtype:"float64" size:6405008 ndim:1 shape:(6405008) items
ize:8
name:"id_73" dtype:"float64" size:6405008 ndim:1 shape:(6405008) items
ize:8
name:"id_74" dtype:"float64" size:6405008 ndim:1 shape:(6405008) items
ize:8
name:"id_75" dtype:"float64" size:6405008 ndim:1 shape:(6405008) items
ize:8
name:"id_58" dtype:"int64" size:6405008 ndim:1 shape:(6405008) itemsiz
e:8
name:"id_59" dtype:"int64" size:6405008 ndim:1 shape:(6405008) itemsiz
e:8
```

## Do a simple computation in the Arkouda server about pickup-time indexed logically by the store-and-forward flag

In [12]:
```
# how many records made it to the server?
numTotal = akdict['tpep_pickup_datetime'].size

# use the store_and_forward column to index tpep_pickup_datetime
# see how many time was false
numFalse = akdict['tpep_pickup_datetime'][~akdict['store_and_fwd_flag'
]].size

# use the store_and_forward column to index tpep_pickup_datetime
# see how many time was true
numTrue = akdict['tpep_pickup_datetime'][akdict['store_and_fwd_flag']]
.size

numTotal == numFalse+numTrue
```

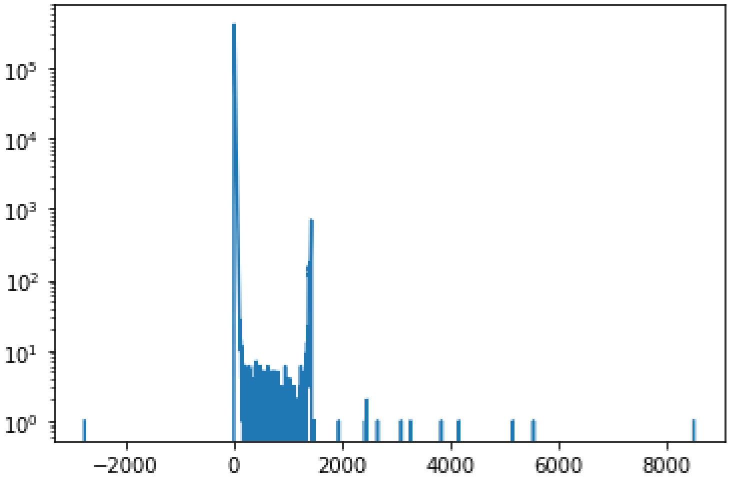
Out[12]:
```
True
```

## Ride duration in Pandas

In [13]:

```
#Pandas ride duration
ride_duration = ydf['tpep_dropoff_datetime'] - ydf['tpep_pickup_dateti
me']
# pull out ride duration in minutes
ride_duration = ride_duration.dt.total_seconds() / 60 # in minutes

print("min = ", ride_duration.min(),"max = ", ride_duration.max())
print("mean = ",ride_duration.mean(),"stdev = ",ride_duration.std())

# how long was the min/max ride to the next integer minute
min_ride = math.floor(ride_duration.min())
print("min_ride = ", min_ride)
max_ride = math.ceil(ride_duration.max())
print("max_ride = ", max_ride)

# histogram the ride time bin by the minute
nBins = max_ride - min_ride
cnts,binEdges = np.histogram(ride_duration, bins=nBins)

print(cnts.size,    "cnts      = ", cnts)
print(binEdges.size,"bin edges = ", binEdges)

# plot the histogram the ride time, bin by the minute
plt.plot(binEdges[:-1],cnts)
plt.yscale('log')
plt.xscale('linear')
plt.show()
```

```
min =  -2770.366666666667 max =  8525.116666666667
mean =  15.950077949417679 stdev =  63.2299915765068
min_ride =  -2771
max_ride =  8526
11297 cnts      =  [1 0 0 ... 0 0 1]
11298 bin edges =  [-2770.36666667 -2769.36680092 -2768.36693517 ...
8523.11693517
  8524.11680092  8525.11666667]
```

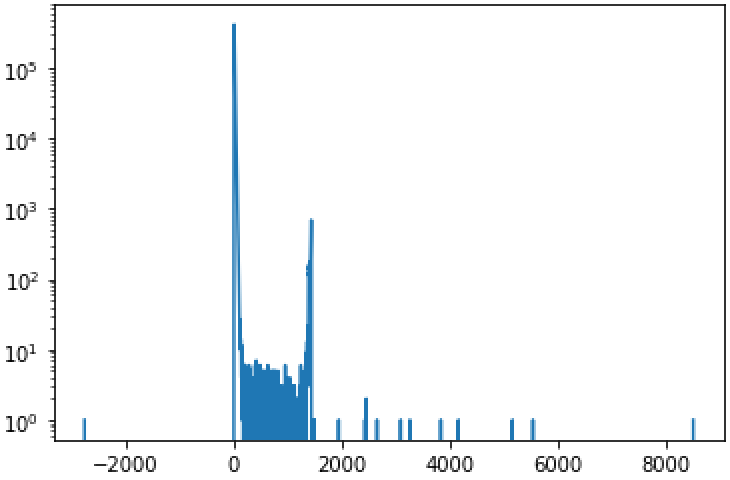


## Ride duration in Arkouda

In [14]:

```
# take delta for ride duration
ride_duration = akdict['tpep_dropoff_datetime'] - akdict['tpep_pickup_datetime']
# pull out ride duration in minutes
ride_duration = ns_to_min(ride_duration)

print("min = ", ride_duration.min(),"max = ", ride_duration.max())
print("mean = ",ride_duration.mean(),"stdev = ",ride_duration.std())

# how long was the min/max ride to the next integer minute
min_ride = math.floor(ride_duration.min())
print("min_ride = ", min_ride)
max_ride = math.ceil(ride_duration.max())
print("max_ride = ", max_ride)

# histogram the ride time bin by the minute
nBins = max_ride - min_ride
cnts = ak.histogram(ride_duration, bins=nBins)

# create bin edges because ak.histogram doesn't
binEdges = np.linspace(ride_duration.min(), ride_duration.max(), nBins+1)
print(binEdges)

print(cnts.size,    "cnts      = ", cnts)
print(binEdges.size,"bin edges = ", binEdges)

# plot the histogram the ride time, bin by the minute
plt.plot(binEdges[:-1],cnts.to_ndarray())
plt.yscale('log')
plt.xscale('linear')
plt.show()
```

```
min =  -2770.366666666667 max =  8525.116666666667
mean =  15.95007794942065 stdev =  63.229986640440934
min_ride =  -2771
max_ride =  8526
[-2770.36666667 -2769.36680092 -2768.36693517 ...  8523.11693517
  8524.11680092  8525.11666667]
11297 cnts      =  [1 0 0 ... 0 0 1]
11298 bin edges =  [-2770.36666667 -2769.36680092 -2768.36693517 ...
8523.11693517
  8524.11680092  8525.11666667]
```

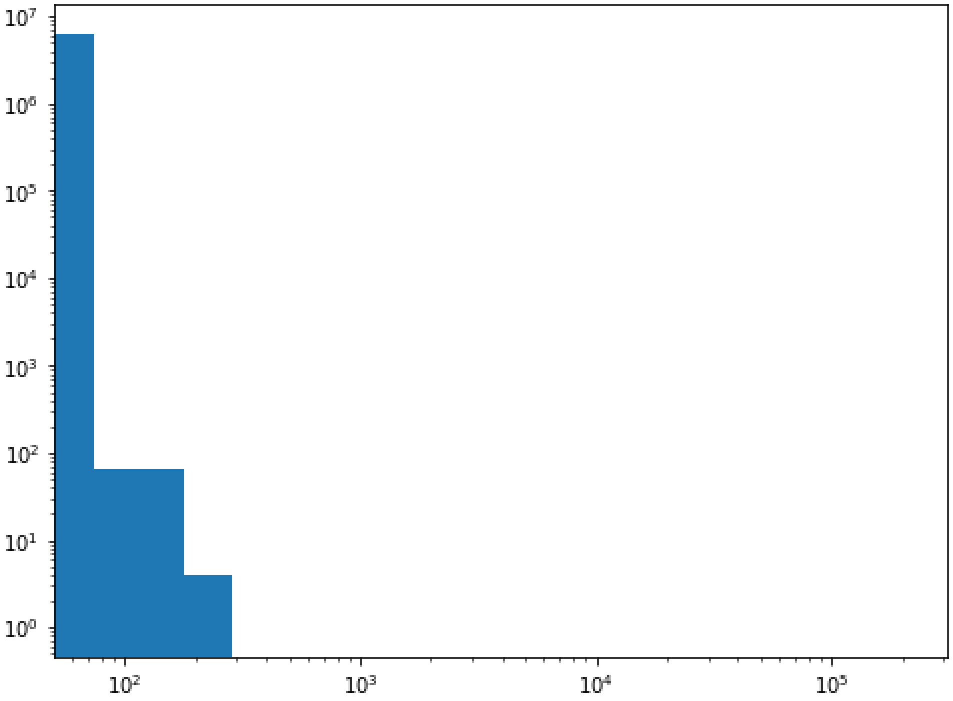


## Compute somehting with trip distance in Pandas

In [15]:

```
print(ydf['trip_distance'].min(), ydf['trip_distance'].max())
print(ydf['trip_distance'].mean(), ydf['trip_distance'].std())

plt.figure(figsize=(8,6))
plt.hist(ydf['trip_distance'],bins=2000)
#ax = plt.gca()
#ax.set_xlim((ydf['trip_distance'].min(),ydf['trip_distance'].max()))
plt.yscale('log')
plt.xscale('log')
plt.show()
```

```
-30.62 210240.07
2.92964393330939 83.15910597325018
```



### Compute somehting with trip distance in Arkouda

In [ ]:

## Taxi Zone Lookup Table

### Read in the data

In [16]:

```
# read the taxi-zone-lookup-table
cvt = {'Borough':cvt_to_string, 'Zone':cvt_to_string, 'service_zone':cvt_to_string}
tzlut = pd.read_csv("../Downloads/taxi+_zone_lookup.csv",converters=cvt)
# print out the tzlut which was read from file
print(tzlut)

# location id is 1-based, index is 0-based
# fix it up to be aligned with index in data frame
# which means add row zero
top_row = pd.DataFrame({'LocationID': [0], 'Borough': ['N/A'], 'Zone': ['N/A'], 'service_zone': ['N/A']})
tzlut = pd.concat([top_row, tzlut]).reset_index(drop = True)
# print fixed up tzlut
tzlut
```

```
     LocationID        Borough                     Zone service_zone
0             1            EWR           Newark Airport          EWR
1             2         Queens              Jamaica Bay    Boro Zone
2             3          Bronx  Allerton/Pelham Gardens    Boro Zone
3             4      Manhattan            Alphabet City  Yellow Zone
4             5  Staten Island            Arden Heights    Boro Zone
..          ...            ...                      ...          ...
260         261      Manhattan       World Trade Center  Yellow Zone
261         262      Manhattan           Yorkville East  Yellow Zone
262         263      Manhattan           Yorkville West  Yellow Zone
263         264        Unknown                       NV          N/A
264         265        Unknown                       NA          N/A

[265 rows x 4 columns]
```

Out[16]:

| | LocationID | Borough | Zone | service_zone |
|---|---|---|---|---|
| **0** | 0 | N/A | N/A | N/A |
| **1** | 1 | EWR | Newark Airport | EWR |
| **2** | 2 | Queens | Jamaica Bay | Boro Zone |
| **3** | 3 | Bronx | Allerton/Pelham Gardens | Boro Zone |
| **4** | 4 | Manhattan | Alphabet City | Yellow Zone |
| **...** | ... | ... | ... | ... |
| **261** | 261 | Manhattan | World Trade Center | Yellow Zone |
| **262** | 262 | Manhattan | Yorkville East | Yellow Zone |
| **263** | 263 | Manhattan | Yorkville West | Yellow Zone |
| **264** | 264 | Unknown | NV | N/A |
| **265** | 265 | Unknown | NA | N/A |

266 rows × 4 columns

## Check the columns for all strings

In [17]:

```
# check the columns for all strings
print(["{} is all str -> {}".format(name, is_all_str(tzlut[name].value
s)) for name in tzlut.keys()])
```

```
['LocationID is all str -> False', 'Borough is all str -> True', 'Zone
is all str -> True', 'service_zone is all str -> True']
```

## Convert dataframe to dictionary of Arkouda arrays

In [18]:

```
# convert data frame with strings and int64 data
aktzlut = ak_create_akdict_from_df(tzlut)
```

```
LocationID  :  int64 -> int64
Borough  :  object -> ak.Strings
Zone  :  object -> ak.Strings
service_zone  :  object -> ak.Strings
```

## what did we get on the server side

In [19]:

```
print(aktzlut)
```

```
{'LocationID': array([0, 1, 2, ..., 263, 264, 265]), 'Borough': array
(['N/A', 'EWR', 'Queens', ... , Manhattan, Unknown, Unknown]), 'Zone':
array(['N/A', 'Newark Airport', 'Jamaica Bay', ... , Yorkville West, N
V, NA]), 'service_zone': array(['N/A', 'EWR', 'Boro Zone', ... , Yello
w Zone, N/A, N/A])}
```

## GroupBy pickup and dropoff location id

- Compute something with Groupby/aggregate in Pandas and in Arkouda
    - Groupby on pickup and dropoff location ids

In [20]:

```
# groupby puckup(PU) and dropoff(DO) location ids(LID)
byLIDs = ak.GroupBy((akdict['PULocationID'], akdict['DOLocationID']))
# print unique keys
print('unique_keys (PU,DO): ', byLIDs.unique_keys)
```

```
unique_keys (PU,DO):  [array([1, 1, 1, ..., 265, 265, 265]), array([1,
50, 68, ..., 263, 264, 265])]
```

## Use groupby/aggregate on edge list to construct a condensed graph

In [21]:

```
# create a condensed graph of LID (PU,DO) pairs with different weights
graphLID = {}
 # vertex names (integer)
graphLID['V']    = aktzlut['LocationID']
# unique edges
graphLID['E']    = byLIDs.unique_keys
# edge weight: count of each unique edge
graphLID['W_CT'] = byLIDs.count()[1]
# edge weight: mean trip distance per edge
graphLID['W_TD'] = byLIDs.mean(akdict['trip_distance'])[1]
# edge weight: mean ride duration per edge
graphLID['W_RD'] = byLIDs.mean(ride_duration)[1]
# edge weight: mean fare amount per edge
graphLID['W_FA'] = byLIDs.mean(akdict['fare_amount'])[1]
# print the graph
print("graphLID = ", graphLID)
```

```
graphLID =  {'V': array([0, 1, 2, ..., 263, 264, 265]), 'E': [array
([1, 1, 1, ..., 265, 265, 265]), array([1, 50, 68, ..., 263, 264, 26
5])], 'W_CT': array([638, 1, 1, ..., 4, 317, 2508]), 'W_TD': array([0.
79800940438871437, 16.07000000000005, 16.190000000000055, ..., 2.54499
99989941716, 0.1309463722449184, 2.219254385957723]), 'W_RD': array
([2.6811650992685414, 24.400000000000091, 42.666666666666742, ..., 13.
5, 5.7159305985793702, 8.3492357790161176]), 'W_FA': array([81.0315203
76175536, 60.5, 65.5, ..., 11.875, 92.17041009462406, 77.1915709728306
75])}
```

## Compute something with the condensed graph

In [22]:

```
# Compute something with the condensed graph...
```

## Use join-on-eq-with-dt to look at adjacent trips

This is one of the most complex operations available in Arkouda and it's not well documented ;-)

- `ak.join_on_eq_with_dt(a1, a2, t1, t1, dt, pred, result_limit=1000)`
- this function finds places where the first two integer arrays are equal value then uses the predicate to test the second two arrays, if all is true then append indices to result list
- so, `if ((a1[i] == a2[j]) and predicate(t1[i],t2[j])) then add (i,j) to the result list`
- the `pred` can be `'true_dt'` , `'pos_dt'` , `'abs_dt'`
- the `result_limit` makes sure you can handle the memory footprint of the result, this isn't implemented well right now in the multi-locale context

you can think of this as emitting 2-long chains from the line-graph of edges(trips) where the predicate is true

so trip1.DOLocationID is the same as trip2.PULocationID and the second trip happens within time related to the first trip

In [23]:
```
# ak.join_on_eq_with_dt(a1, a2, t1, t1, dt, pred, result_limit=1000)
#
# you can think of this as emitting 2-long chains from the line-graph
 of trips where the predicate is true
# so trip1.DOLocationID is the same as trip2.PULocationID
# and the second trip happens within time related to the first trip
#
(I,J) = ak.join_on_eq_with_dt(akdict['DOLocationID'], akdict['PULocati
onID'],
                              akdict['tpep_dropoff_datetime'], akdict[
'tpep_pickup_datetime'],
                              min_to_ns(1), 'pos_dt', result_limit=100
_000)
print(I.size,I,J.size,J)
```

```
100000 [0 0 0 ... 3210897 3210897 3210897] 100000 [3168 3590 4603 ...
3202453 3207378 3211660]
```

- use join-with-dt
- other things
  - model number of taxis at a given time
  - model taxis as specific entities (Kalman filter?)
- use time and location ids
- probability of paths of taxis
- ...
  - other things?

In [ ]:
```
```

## disconnect from the server or shutdown the server

In [24]:
```
# disconnect or shutdown the server
#ak.disconnect()
#ak.shutdown()
```

In [ ]:
```
```