Koln Traffic Regulator with Parallel Computing

Management and Analysis of Physics Dataset, mod. B

Tommaso Amico
Andrea Lazzari
Paolo Zinesi
Nicola Zomer

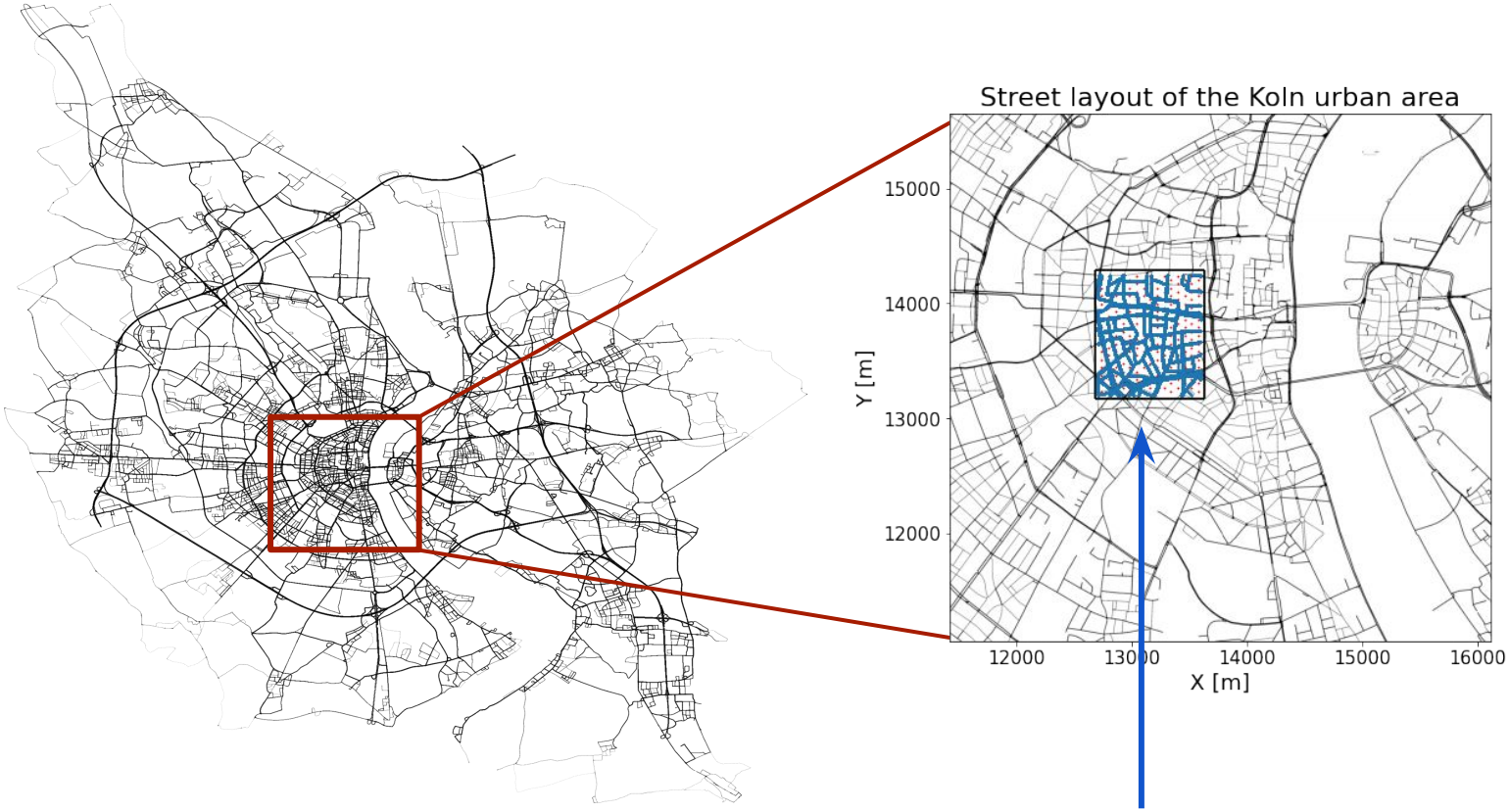
Introduction

- Starting from an IBM project, we used parallel computing to regulate the traffic of the German city of Koln.

- The mobility is emulated with SUMO, an open-source traffic simulation suite that allows generating the movement around a predefined city road map, extracting the data for the analysis from past scenarios

time [s]	ID	x	y	Velocity [m/s]
Time-stamp of each record	Vehicle identifier	Coordinate x of the car	Coordinate y of the car	Speed of the car

Introduction



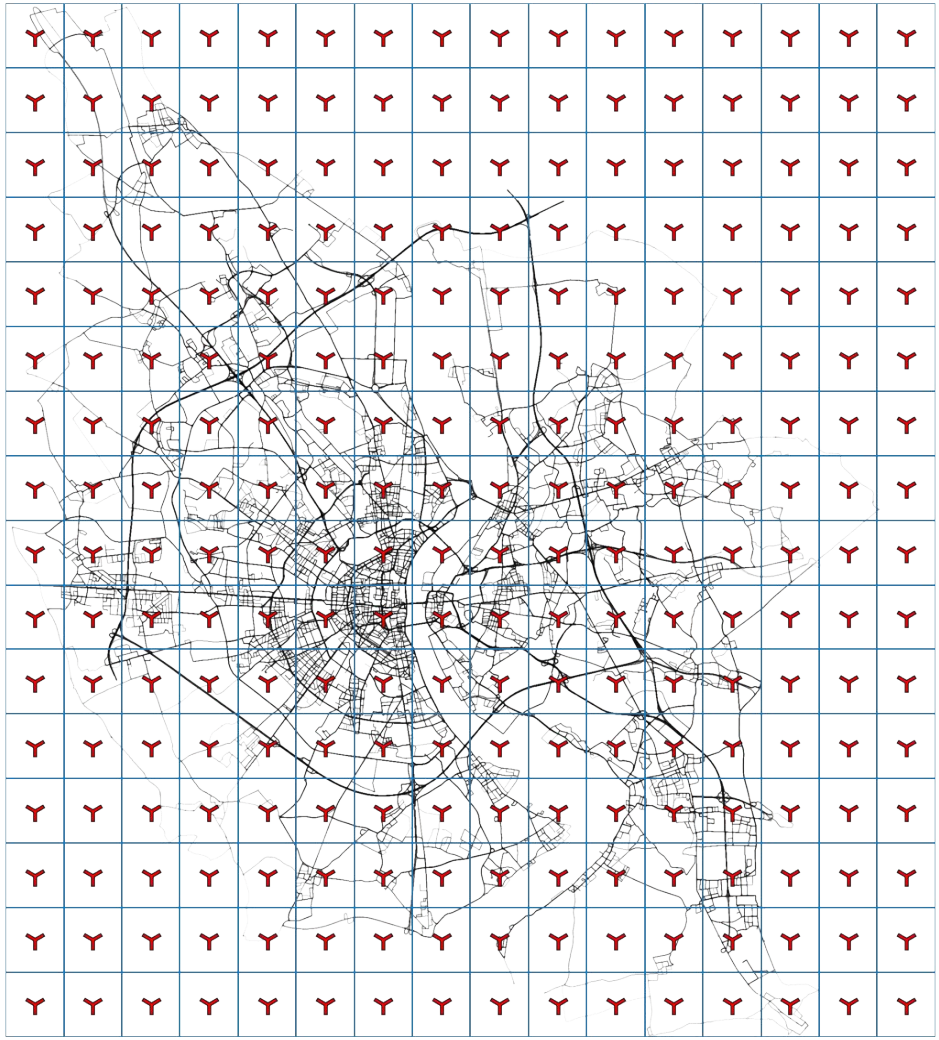
3.300.000 connections
394 millions records
20 GB

77.000 connections
6.9 millions records
150 MB

The Grid

- Create a grid of rectangles that spans the whole map and then fill it with Base Stations (BS) at the center of each rectangle.

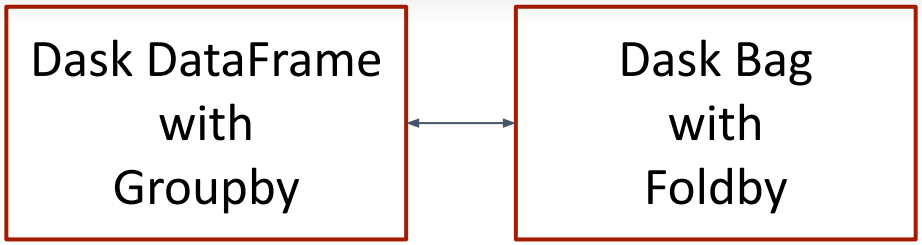
- Unique matching between cars and Base Stations which are in the same rectangle of the grid



The Cluster

- Dask cluster structure:
  - 5 machines - 4 cores and 8 GB RAM each
    - 1 scheduler + worker
    - 4 workers
  - Mounted volume NFS for the data

```
from dask.distributed import Client, SSHCluster
cluster = SSHCluster(
     ["10.67.22.41", "10.67.22.41", "10.67.22.145", "10.67.22.146", '10.67.22.160', '10.67.22.253'],
     worker_options={"nthreads": 1, 'n_workers':4},
     scheduler_options={"dashboard_address": ":8789"}
)
client = Client(cluster)
```



First Benchmarks

- Benchmark on the optimal number of files in the import of the data
- **5** iterations for each choice of the number of files
- We fix **{n_workers = 4, nthreads = 1, blocksize = 9.455MB}**

					compute_time [s]	
					mean	std
nthreads	n_workers	blocksize_MB	nfiles	npartitions		
1	4	9.455	1	2147	241.461	9.809
			2	2146	239.505	5.839
			4	2144	237.882	11.660
			8	2144	240.448	9.574
			16	2144	239.607	6.377

We do not notice a definite difference between the import times:
we therefore choose to keep the starting configuration, reading the data from a single file.

Data reading & cleansing

- Data imported into a Dask Bag

- Same data cleansing operation for both of our approaches

```
db_travel = db.read_text('/dataNFS/koln.txt', blocksize='9.455MB')

def clean_round_row(row):

    pattern = '[0-9]+\.?[0-9]*'
    findings = re.findall(pattern, row[:-1])

    if(len(findings) == len(row[:-1].split(' '))):
        round_findings = [round(float(x),4) for x in findings]
        return round_findings
    else:
        return []

db_cleaned = db_travel.map(clean_round_row).filter(lambda x: len(x)>0)
```

Car - BS matching

Dask Dataframe

```
dask_df = db_cleaned.to_dataframe(meta = {'time':int,
'ID':int, 'x':float, 'y':float, 'velocity':float})
```

```
dask_df['horizontal'] = (dask_df['x'] - left)//delta_x
dask_df['vertical'] = (dask_df['y'] - lower)//delta_y

def square_match(row):

    base_targets = df_BS[df_BS['horizontal'] == row.horizontal]
    base_targets = base_targets[base_targets['vertical'] == row.vertical]
    base_targets = base_targets['base_ID'].values

    if len(base_targets) == 1:
        return base_targets[0]

    else: return -1

dask_df['base_ID'] = dask_df.apply(square_match, axis=1, meta=int)
dask_df_filt = dask_df[dask_df['base_ID'] >= 0]
```

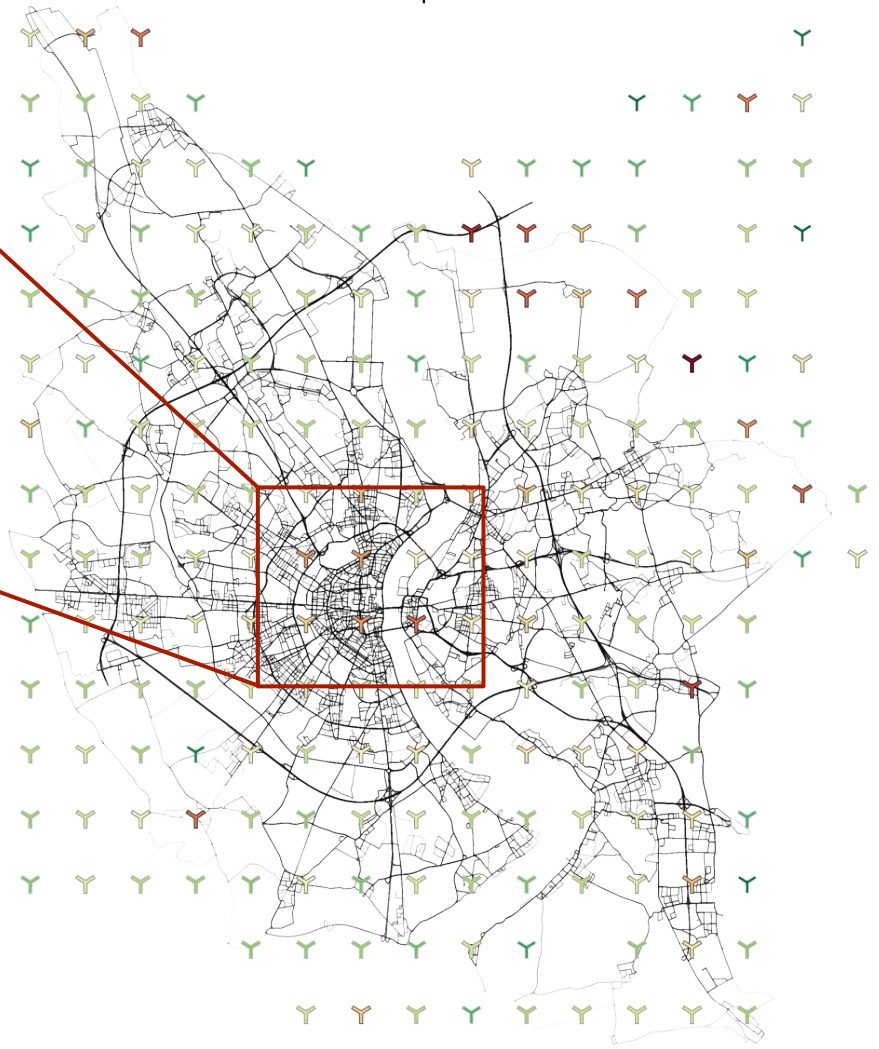
Dask Bag

```
db_cleaned = db_travel.map(clean_round_row).filter(lambda x: len(x)>0)\
                      .map(lambda x: {'time':int(x[0]), 'ID':int(x[1]),
                                      'x':float(x[2]), 'y':float(x[3]),
                                      'velocity':float(x[4])})
```

```
def hor_ver_func(row):

    row['horizontal'] = int((row['x'] - left)//delta_x)
    row['vertical'] = int((row['y'] - lower)//delta_y)

    return row

def square_match_db(row):

    base_targets = df_BS[df_BS['horizontal'] == row['horizontal']]
    base_targets = base_targets[base_targets['vertical'] == row['vertical']]
    base_targets['base_ID'] = base_targets.index

    if len(base_targets) == 1:
        row['base_ID'] = base_targets['base_ID'].values[0]

    else:
        row['base_ID'] = -1

    return row

db_HV_matched = db_HV.map(square_match_db).filter(lambda x: x['base_ID'] >= 0)
```

Metrics - Mean Time

Mean time spent in each BS




```
groupby(['base_ID', 'ID'])\
    .time.agg(['max','min'])
```

↓

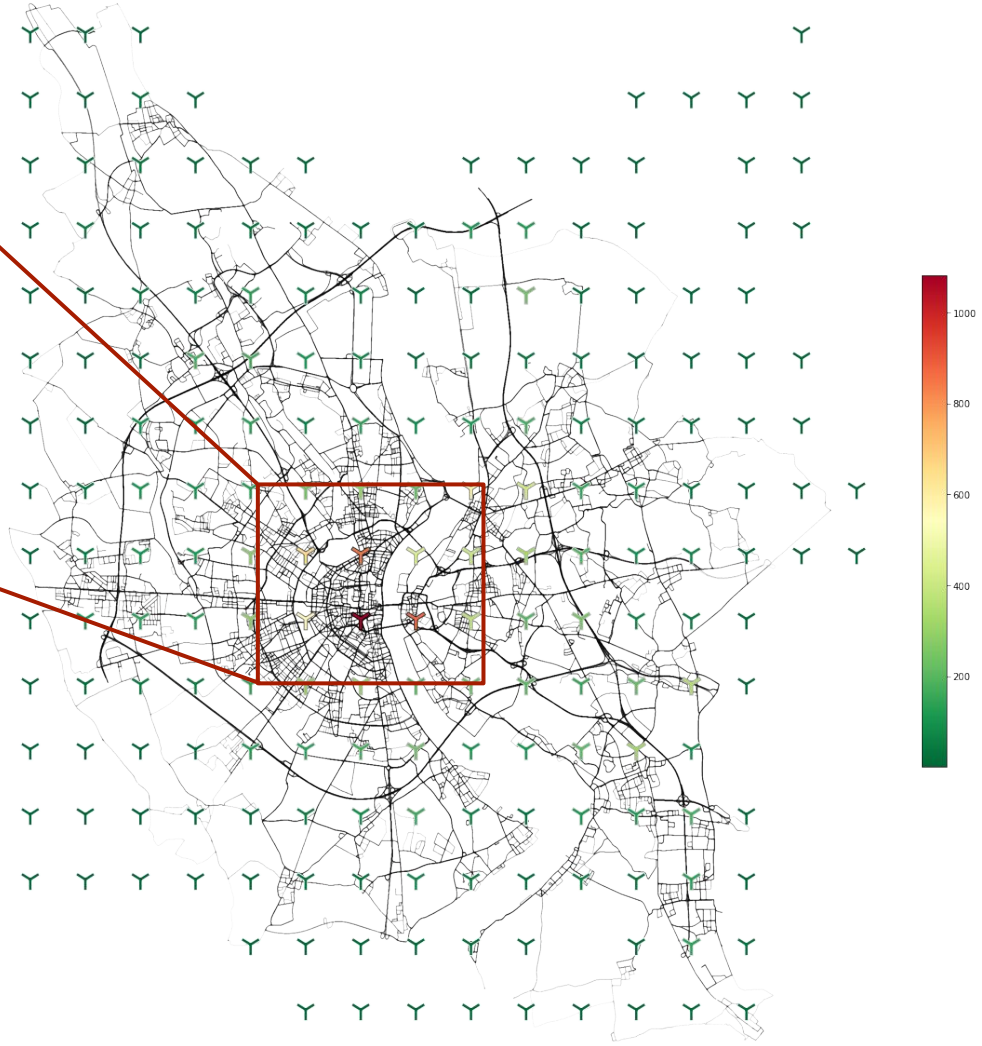
```
diff_time = max - min
```

↓

```
groupby('base_ID').mean()
```

Metrics - Max cars at once

Max number of cars connected to each BS



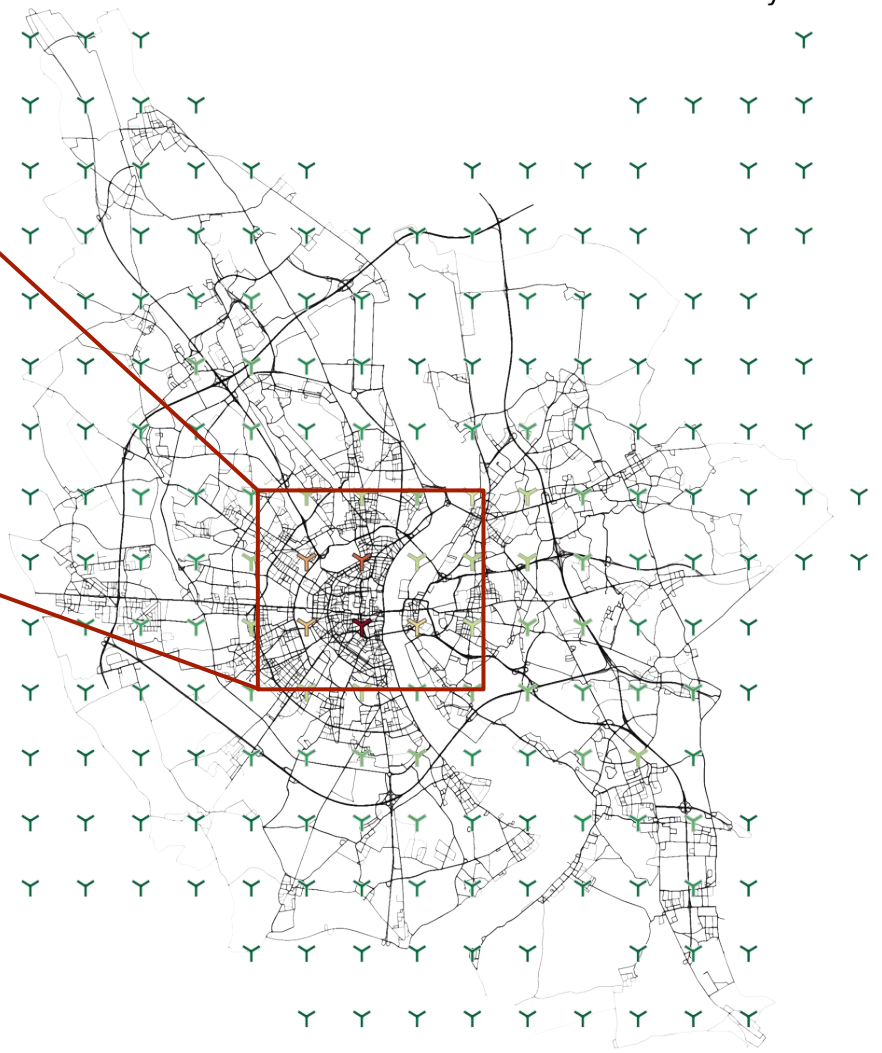

```
groupby(['time', 'base_ID'])\
    .ID.count()
```

↓

```
groupby('base_ID').max()
```

Metrics - Daily connections

Number of connections to a BS in the whole day



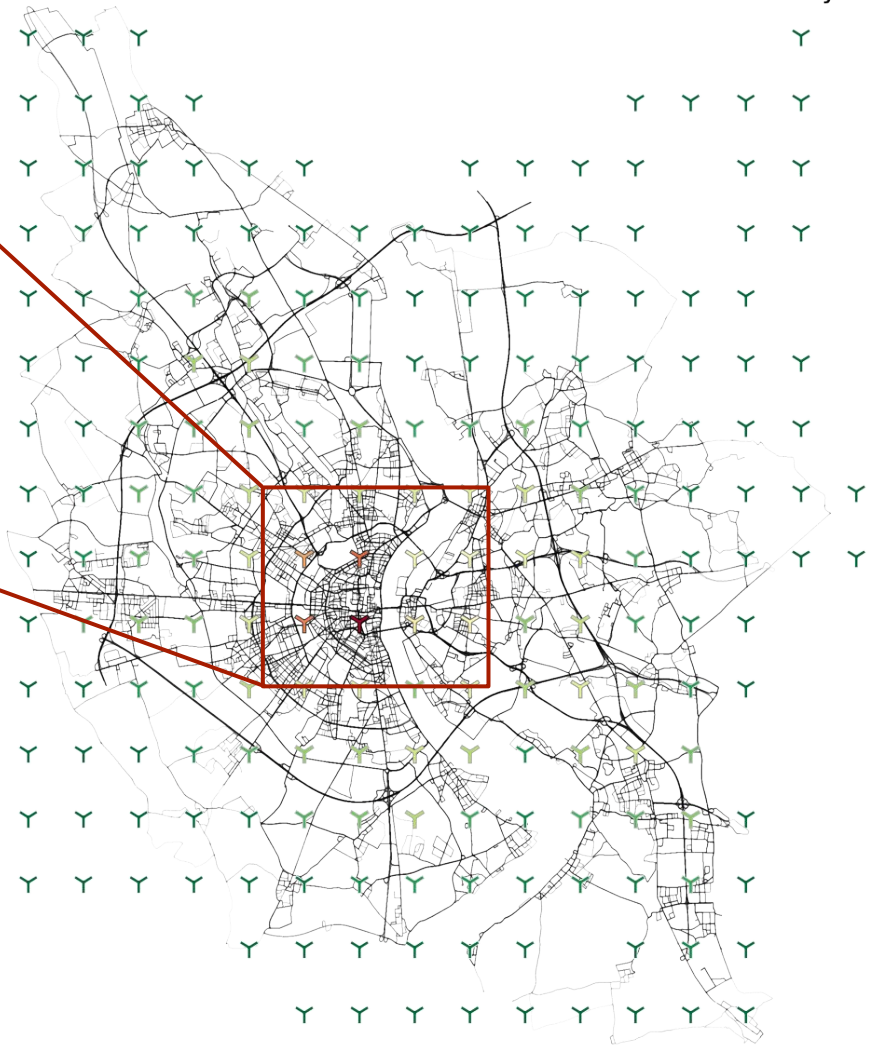

```
groupby(['time', 'base_ID'])\
    .ID.count()
```

(Computed already)

```
groupby('base_ID').sum()
```

Metrics - Total cars per day

Total number of cars connected to a BS in the whole day




```
groupby(['base ID', 'ID'])\
    .time.agg(['max','min'])
```

(Computed already)

```
groupby('base_ID').count()
```

Groupby - Foldby comparison

UNIVERSITÀ DEGLI STUDI DI PADOVA

Groupby

```
grouped_df = dask_df_filt.groupby(['base_ID', 'ID']).time.agg(['max','min'], split_out=4)
```

Foldby

```
result_fold_db_minmax = db_HV_matched.foldby( key = ['base_ID', 'ID'],
                                              binop = lambda tot, x: {'min':min(tot['min'],x['time']),
                                                                      'max':max(tot['max'],x['time'])},
                                              initial = {'min':np.inf, 'max':-np.inf},
                                              combine = lambda tot1, tot2: {'min':min(tot1['min'],tot2['min']),
                                                                            'max':max(tot1['max'],tot2['max'])},
                                              combine_initial = {'min':np.inf, 'max':-np.inf})
```

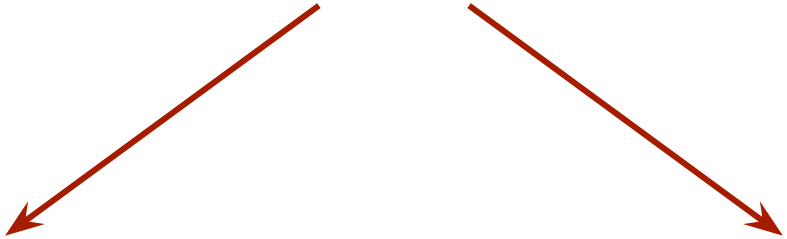
Groupby - Foldby benchmark

Benchmark results:
plots and tables…

Groupby - Foldby comparison

UNIVERSITÀ DEGLI STUDI DI PADOVA

Similar computing times, but…



Groupby

- Easy to implement
- Use the same API of Pandas Dataframe
- Possibility to split the final results into multiple partitions

Foldby

- Difficult to implement (a lot of binary operations have to be defined)
- Must use Dask bags even if the dataset is structured
- Results stored in a single partition

Dashboard

Plot the **number of connections to each BS** in a selected time window.

SLIDERS

- Time of Day
- Window Size
- Update Plot

Python libraries: `Panel`, `HoloViews`

```
# convert time column to datetime format
origin_time = '2022-07-13'

dd_bs_connect["time"] = dd_bs_connect["time"].apply(
    lambda x: pd.to_datetime(x, unit='s', origin=origin_time),
    meta=('time', 'datetime64[ns]')
)

dd_bs_connect = dd_bs_connect.persist()
```

"connection" = car-BS pair at a given time

—> the same car-BS pair at different times correspond to different connections

Update Function


```python
# update plot function
def load_time(time_picker, window_size, update):
    global joined_car_per_base
    global title

    if update=='ON':
        # convert input time interval to datetime format
        if time_picker[6:]=='PM':
            seconds = (int(time_picker[0:2])+12)*3600+int(time_picker[3:5])*60
        else:
            seconds = int(time_picker[0:2])*3600+int(time_picker[3:5])*60

        time_start = pd.to_datetime(seconds, unit='s', origin=origin_time)
        time_end   = pd.to_datetime(seconds+window_size*3600, unit='s', origin=origin_time)

        # update title
        title= 'Number of connections to each BS from '+time_start.strftime("%I:%M %p")+' to '+time_end.strftime("%I:%M %p")

        # update plot dataframe
        df_sel = dd_bs_connect[dd_bs_connect['time'].between(time_start, time_end)].groupby('base_ID').sum().compute()
        joined_car_per_base = df_BS[['x', 'y', 'base_ID']].merge(
            df_sel, left_on='base_ID', right_on='base_ID',
            right_index=False, how='left'
        )

        joined_car_per_base.fillna(0, inplace=True)

    return hv.Scatter(joined_car_per_base, kdims=['x'], vdims=['y', 'n_cars']).opts(...)
```

Streaming

Generate a stream of data from the dataset containing the number of car connected to each BS at each time during the day (**persisted Dask Dataframe**)

Stream generation:

- Library `streamz`
- Library `tornado`
- Information related to a specific time instant one next to the other

```
time_i, time_f = dask.compute(dd_bs_connect['time'].min(),
dd_bs_connect['time'].max())
time_i += datetime.timedelta(seconds=10000)
time = time_i

# generate one record as pandas dataframe, according to time
def emit_rec():
    global time

    record = dd_bs_connect[dd_bs_connect['time']==time].compute()

    time += datetime.timedelta(seconds=1)
    return record

# stream with streamz
source = Stream()
source.scatter()
```

Streaming

- Streaming input data are stored in a `streamz` streaming dataframe
- A time window is set to update the plot

Processing:

```
sdf_occ = sdf.set_index('time').window(value='5s')[['base_ID', 'n_cars']].groupby('base_ID').mean()

sdf_occ['x'] = df_BS.x
sdf_occ['y'] = df_BS.y
```

Dynamic plots:

Table of data in the selected window	Average number of connections in 5s	Average energy amount in 5s