Big Data Project Report

Martin Jurkovič mj5835@student.uni-lj.si
Faculty of Computer and Information Science
63180015

Valter Hudovernik vh0153@student.uni-lj.si
Faculty of Computer and Information Science
63160134

PROJECT REPOSITORY: *https://github.com/martinjurkovic/BIG-DATA-PROJECT*

Introduction

This report presents a comparison between three different file formats - *Parquet, DuckDB, and HDF5* and the computing framework *Dask* on the task of processing a large dataset. In the context of big data analysis, it is essential to understand the performance characteristics of different data formats, when integrated with distributed computing frameworks like Dask. We also perform our analysis in a streaming manner using the popular Kafka framework and compare it with the distributed computing approaches.

Our approach involves systematic testing across four distinct tasks, such as data conversion, augmentation, exploratory data analysis (EDA), and machine learning (ML), with three different configurations of worker nodes and memory limits. Our experimental setup focuses on systematically logging memory usage and processing times across all tasks. The goal is to provide a comprehensive comparison of these data formats, offering insights into their suitability for big data applications. Additionally, we comment on the implications of these findings and cross-reference our experiences with those of the community on the practical use of Dask in real-world scenarios, particularly in terms of data format selection and resource allocation.

Experimental Setup

We implement a programmatic way of logging system memory usage and time spent on tasks by following and logging the usage of process ids connected to corresponding python tasks. We log time spent for different tasks (and subtasks if applicable) memory usage logged every second and then report the average and maximum memory usage. We test each task (where applicable) with multiple worker numbers and memory limits per worker. The maximum amount of memory allowed cumulatively for all workers is 32GB, since otherwise the whole dataset can be stored in RAM, with the exception of HDF5 where 128Gb of RAM was

required[1]. Still we need RAM not only to store but to process the data and train the models as well, so the data size is not the only consideration when choosing the amount of RAM.

Data Types and DASK

Dask natively supports reading Parquet and HDF5 files, which means it is optimized for reading the files in a distributed way in parallel. This however does not hold for DuckDB files, where we first have to compute the pandas df and then use dask on that df. This means that loading times will differ depending on the file type, since DuckDB will have to compute the df immediately, while Parquet and HDF5 will be computed on the fly when needed. We also note that we had to compute the df for HDF5 in some of the tasks since otherwise we got a dask internal error which we were not able to fix. We note however that we optimized the DuckDB query to read only necessary columns and read the data in batches.

Python Package

We create a python package called *bigdata* where we define all of the utility methods for reading and storing data, so the function is the same for each task, we just pass a different argument to specify the data type. We also add utility functions for machine learning and data augmentation pipelines. In the package we separately define HDF5 utility functions since we had the most problems with HDF5 and had to write some custom loading code. This way, we are able to keep our project and code clean. The code can be found in the project repository (https://github.com/martinjurkovic/BIG-DATA-PROJECT).

T1 - Data Conversion

We convert the CSV files into *Parquet* and *HDF5* format by reading the data into a pandas dataframe and then converting it to a specified format. For Parquet files the process was simple. We had to specify column names since they did not match over all of the files. For HDF5 we had to convert the pandas object data type columns (which are usually strings) into a fixed length varchar format supported by HDF5, since HDF5 does not support variable length strings. We also create a DuckDB database by reading Parquet files into a single table in the database. As discussed in section *T2 - Data Augmentation*, we insert additional datasets for data augmentation as new tables in the database. The results of conversion times and memory consumption can be seen in Table 1.

We first note that for DuckDB we ingest all files at the same time with one command, so we can only report the total amount of time divided by the number of files as a mean. We see that Parquet and DuckDB used approcimately the same amount of average memory, however Parquet used the whole 32GB of memory available while DuckDB only used 17Gb. HDF5 used the maximum 64Gb RAM available, with average memory consumption of 20Gb. In the time comparison we see that DuckDB is highly optimized for ingesting files

1. We had to subsample the HDF5 data to 50% of the dataset length for the tasks to finish before the Arnes Hyper-Computer Cluster limit. Even with this limit not all of the tasks were able to finish before the time limit.

DATA TYPE	Memory [Mb]		Time [s]											
	Mean	Max	2014	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024	Mean
PARQUET	11236 ± 6535	34132	180	106	246	92	96	351	95	270	121	169	97	173 ± 85
DUCKDB	12809 ± 4321	17077	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	7 ± 00
HDF5	20087 ± 10882	63789	196	311	243	242	251	234	268	321	334	510	260	288 ± 80

Table 1: **Results of task T1** for different data types. For duckdb we only report the mean conversion time as the database is stored together using an SQL-like connection.

and was able to create an embedded database in only 7 seconds per file, whereas Parquet needed 173 seconds on average and HDF5 288 seconds, making it the slowest. Here we can clearly see the advantage of DuckDB.

The file size comparison can be seen in Table 2. We see that parquet is much smaller than CSV, however HDF5 is much much larger than CSV. We note that we do not use any compression and that HDF5 is stored in a *table format*, which makes it larger than other formats, however is needed for use with Dask.

File Name	CSV Size (GB)	Parquet Size (GB)	HDF5 Size (GB)
2014	1.79	0.34	5.86
2015	2.51	0.53	8.31
2016	2.07	0.36	6.97
2017	2.09	0.53	6.96
2018	2.17	0.41	7.55
2019	2.00	0.38	7.39
2020	2.32	0.40	8.05
2021	2.75	0.46	9.63
2022	2.90	0.47	9.94
2023	4.03	0.64	13.89
2024_april	2.25	0.35	7.73
TOTAL	**26.89**	**4.87**	**92.28**

Table 2: **T1 File Sizes.** We do not include DuckDB in the comparison since a) it only contains all of the files and b) it uses high compressions so the comparison is not completely fair.

T2 - Data Augmentation

We enhance the dataset by integrating supplementary information from the *New York City open data repository* (NYC Open Data, 2024) as well as other sources such as *Open Meteo* (Open-Meteo, 2024) for weather data and *photon* (komoot, 2024) for location information. The augmented data includes the locations of schools which are visualized in Figure 1 and opening times of schools [2]. We also incorporate data on business working hours, landmark locations and density, details about events (such as special events and construction activities), and comprehensive holiday information (covering national, NYC-specific, gen-

2. We obtain school hours manually from `https://nces.ed.gov/programs/statereform/tab5_14.asp`.

eral, religious, and school holidays). Additionally, we incorporate weather information to provide a more detailed and enriched dataset.

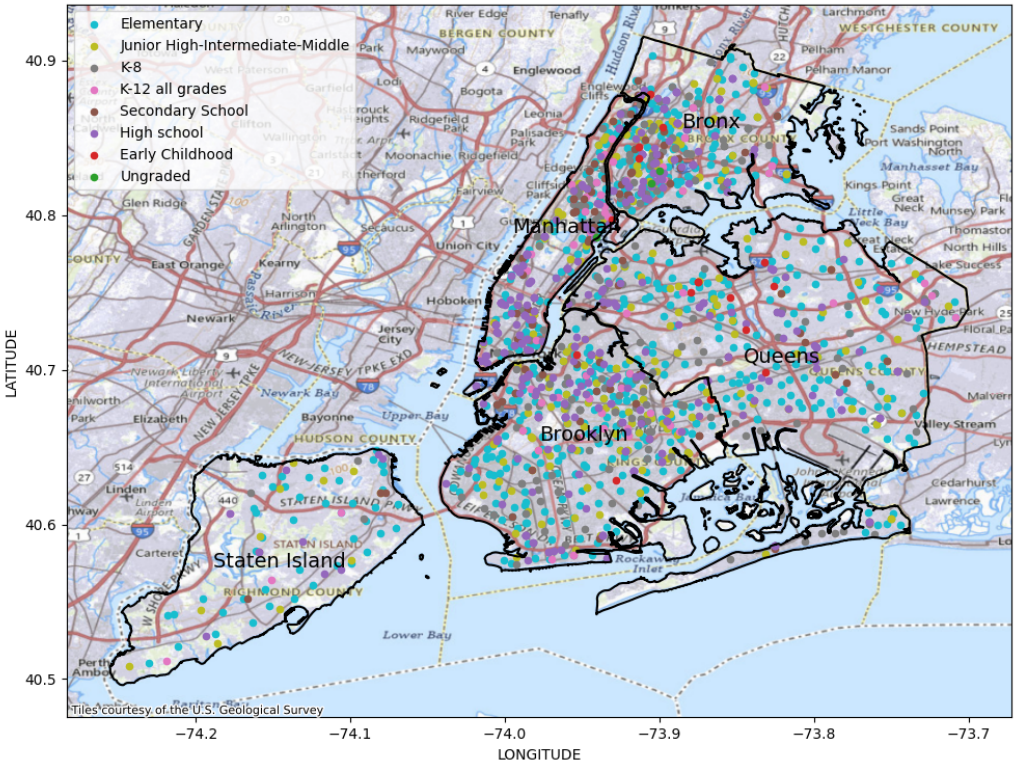


Figure 1: **Location of Schools by the type of school** in our data augmentation we then count the number of schools which are open on an hourly basis for each borough. While the original school data contains precise locations we are unable to join them to the data as it would be to costly (as explained in Section T2 - Data Augmentation).

The granularity of location for all of our features is at the county level (Brooklyn, Bronx, Manhattan, Staten Island and Queens). Geocoding each of the addresses would be infeasible due to API limits and costs (e.g. Google's geocoding api costs 500$ per 100.000 requests [3], which is orders of magnitude less than our data size). The detailed description of all of the columns is available in Appendix A.

The results of the experiments can be seen in Table 3. We see that the amount of memory used is so small that one worker can handle most of the processing, so the processing times and memory used do not change drastically when comparing the same data type. We can also see that the processing time is almost the same for Parquet, as well as DuckDB. This means that most of the time is actually spent processing the augmentation data and only a negligeable amount is spent saving it to a specific format. This is not true for HDF5 howeverm wgere we can see that it took much longer to process the files. The memory used however is similar to Parquet, whereas DuckDB uses the most amount of memory.

3. `https://mapsplatform.google.com/pricing/`

DATA TYPE	N WORKERS	Memory [Mb]		Time [s]						
		Mean	Max	Schools	Holidays	Businesses	Events	Landmarks	Weather	Mean
PARQUET	1	517 ± 73	656	6.67	41.25	311.54	28.78	0.30	0.42	64.83 ± 111.37
	2	515 ± 70	649	6.07	41.35	311.44	28.57	0.41	0.40	64.71 ± 111.38
	3	506 ± 64	624	6.08	41.68	312.29	29.37	0.62	0.41	65.08 ± 111.62
DUCKDB	1	1245 ± 328	1730	6.25	42.45	315.40	28.98	0.37	1.03	65.75 ± 112.71
	2	1171 ± 297	1663	6.35	43.25	324.40	30.29	0.60	1.09	67.67 ± 115.90
	3	1157 ± 291	1638	6.40	43.01	327.16	29.84	0.65	1.04	68.02 ± 116.95
HDF5	1	535 ± 127	787	43.03	154.60	526.67	63.22	14.87	7.91	135.06 ± 181
	2	586 ± 119	781	27.69	86.56	535.89	62.85	7.07	4.15	120.71 ± 187.99
	3	599 ± 123	795	25.38	88.98	536.55	61.89	7.46	3.97	120.71 ± 188

Table 3: **Results of task T2** for different data types and DASK worker specifications. Memory consumption for DuckDB is significantly higher than for the other two file formats due to expensive write operations. Average processing times for DucDB and Parquet are roughly similar with hdf5 processing times being rougly twice as long. Due to a diverse selection of visualizations the variability of processing times is high for all three file formats.

T3 - Exploratory Data Analysis

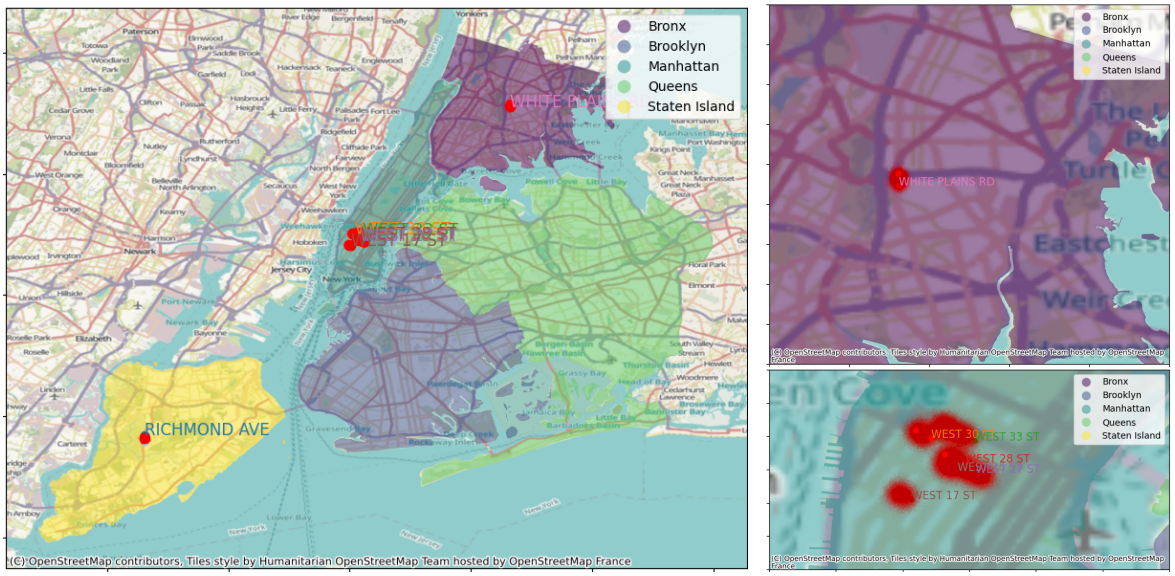
We implement an exploratory data analysis (EDA) pipeline in completely in dask, where we run the same EDA pipeline on all three data types (Parquet, DuckDB and HDF5) with different worker specifications to test scalability. We implement the EDA pipeline with memory efficiency in mind by only loading the necessary columns into memory for a specific visualisations. That means that we can time each plot and rigorously compare the speed and memory efficiency of the EDA pipeline between different data formats.

We create a pipeline for displaying 10 plots which differ in plot type as well as data type (numerical vs categorical). The plots created are: distribution of registration states, distribution of vehicle body type, distribution of vehicle year, correlation matrix of numerical features, top 10 most frequent violations with count, violations over time, violations by precinct, top 10 vehicles with most violations, violations by issuing agency and heatmap of violations counts by location and precinct. We omit these plots in the report to not make the report contain too many figures, so we include them in the git repo and only include the plot using geographical visualizations (as required by tast T7), which can be seen in Figures 2 and 1. The results of the programatic EDA can be seen in Table 4.

DATA TYPE	N WORKERS	Memory [Mb]		Time per Plot [s]										
		Mean	Max	1	2	3	4	5	6	7	8	9	10	Mean
PARQUET	1	6030 ± 3202	10087	27.74	10.83	3.62	0.74	10.30	15.65	9.62	11.45	7.94	18.22	10.39 ± 6.93
	2	5847 ± 2735	8758	24.27	10.90	3.65	1.15	9.93	15.80	9.65	11.36	9.41	17.70	10.21 ± 6.17
	3	6020 ± 2916	9449	24.43	11.23	3.69	1.10	9.82	15.68	9.62	11.27	8.12	17.76	10.12 ± 6.23
DUCKDB	1	6425 ± 2169	9871	2.69	0.79	0.35	0.15	0.73	0.20	0.64	0.80	0.64	2.90	0.89 ± 0.84
	2	6299 ± 2040	9704	2.47	0.84	0.36	0.16	0.76	0.23	0.67	0.83	0.67	3.05	0.90 ± 0.83
	3	6304 ± 2049	9730	2.45	0.81	0.34	0.14	0.75	0.21	0.64	0.79	0.64	2.93	0.87 ± 0.81
HDF5	1	5275 ± 1503	11951	79.32	4.55	5.13	2.78	3.74	3.74	3.72	4.27	3.65	19.92	13.08 ± 22.59
	2	5223 ± 1801	11569	62.29	4.09	4.19	2.40	3.25	3.03	3.01	3.82	3.07	19.45	10.86 ± 17.81
	3	5454 ± 1678	12356	48.78	4.38	3.58	2.31	2.72	2.77	2.87	3.58	2.89	19.27	9.32 ± 14.02

Table 4: **Results of task T3** for different data types and DASK worker specifications. The mean memory consumption for all file formats is rougly similar, however DuckDB has a rougly 10% lower maximum memory consumption, while also producing the plots the fastest on average.

We see that all data types used a similar amount of memory, where DuckDB used the lowest amount. For speed we see DuckDB being for an order of magnitude faster than the other two, which are similar to each other.



(a) (b)

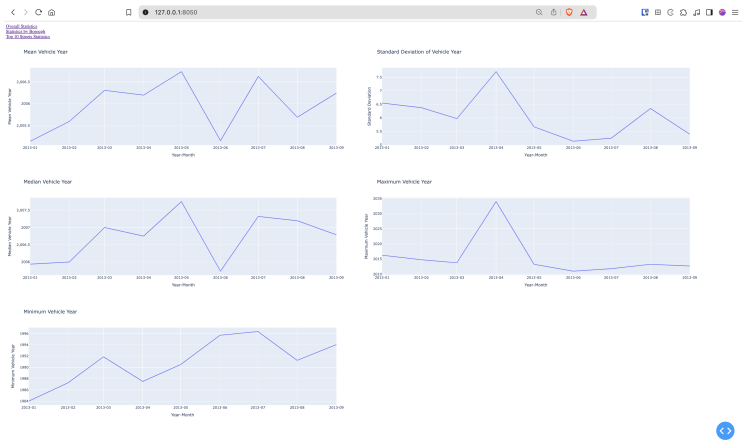
Figure 2: **Highest ticket count streets.** The busies streets for ticket violations are located in Manhattan, the red areas represent the locations with most daily tickets. In (a) we can see that most of the busiest streets are located in Manhattan, with Staten Island and Bronx each being home to one. A zoomed in version of the busiest streets in Manhattan and Bronx is shown in (b). We do not show the zoomed in version of Richmond Avenue as most of its violations are concentrated at a single location 2655 Richmond Avenue or "The Staten Island Mall".

T4 - Kafka Streaming Analysis

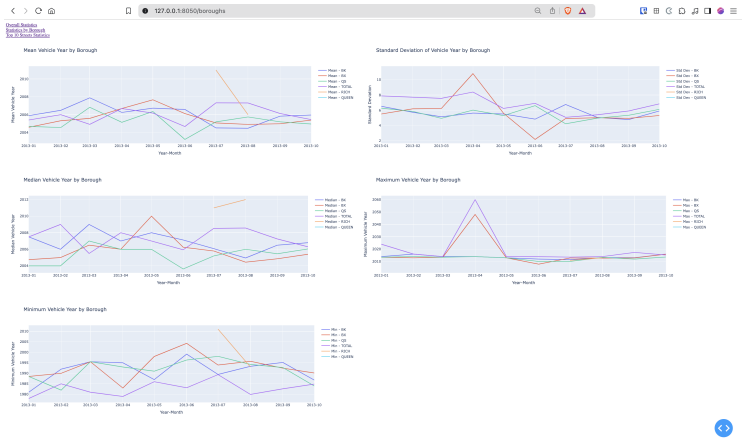
We implement a complete Kafka streaming pipeline. First we implement Kafka and Zookeeper in Docker as two services running in parallel. Then we implement a producer, which produces data to a single topic in Kafka.

We define and implement two consumers. The first one is for statistical data analysis with moving averages whereas the second one is for clustering the data and using the clusters for predicting whether the violator is coming from inside or out of the state of New York.

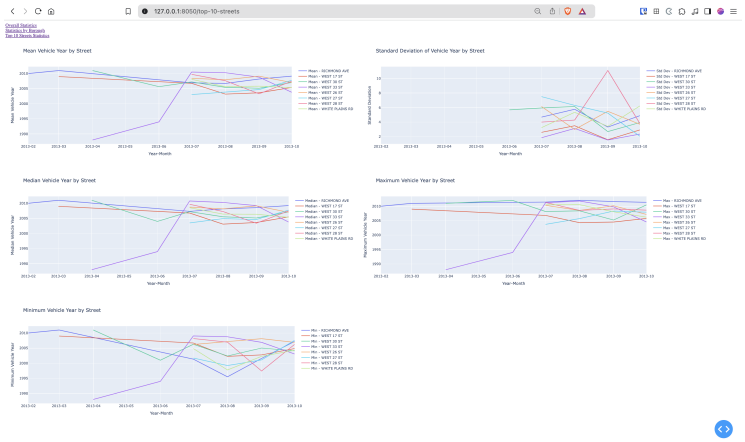
For the data analysis consumer we implement a live dashboard which runs in parallel to the consumer. On it we can see live graphs of the moving average mean, standard deviation, median, maximum and minimum value of the column we decide to analyse. We also implement another two pages in the dashboard, where the same graphs are shown however the analysis is split by boroughs and another for the top 10 streets with the highest number of tickets (See Fig. 2).



(a) Overall



(b) Boroughs



(c) Top 10 Streets

Figure 3: **Streaming analyssis of the whole dataset, boroughs, and top 10 streets.** We conduct the analysis by writing a web app in python with the Dash python library and Flask framework on top of Kafka consumers and producers.

For the clustering consumer we implement a dashboard in a similar way. First, we preprocess the data live as it is consumed from kafka. Then, we cluster it with a streaming clustering algorithm *MiniBatchKMeans*, which implements a partial fit method. With the clustering algorithm we decide to predict whether the violator is coming from the state of New York or from another state. We can track the accuracy of our clustering algorithm live in the dashboard. The accuracy is reported for every N violations, which in our case is 10000.

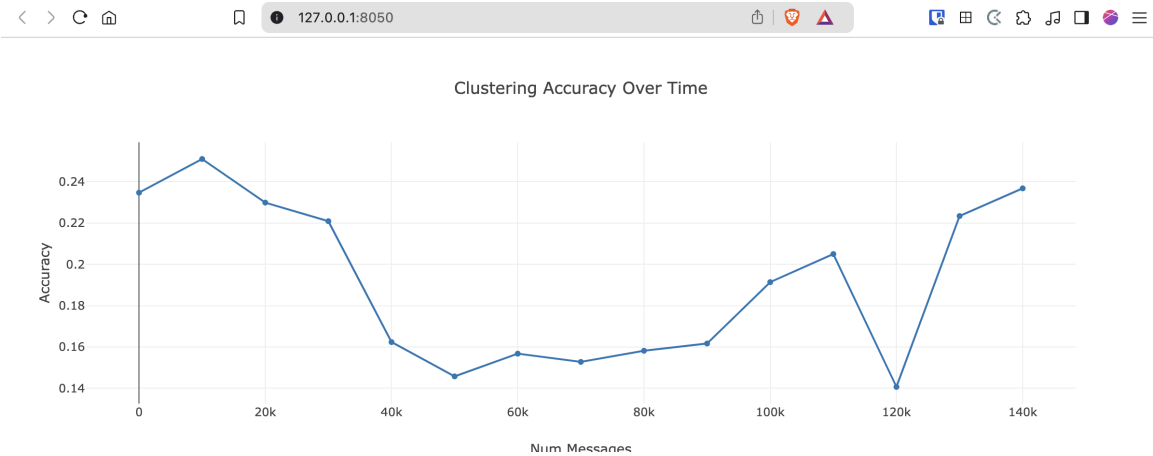


Figure 4: **Clustering Accuracy** for Out of State vehicle prediction using the mini-batch K-means clustering algorithm.

T5 - Machine Learning

We define two task to test how suitable our data is for machine learning. The first task for prediction is the number of daily tickets and the second is predicting if an offending vehicle is registered out of state.

Number of Daily Tickets

For this task we are unable to use any of the features of the original dataset, for this reason we only aggregate the daily ticket counts and combine them with the augmented data from Section T2 - Data Augmentation, which we aggregate to daily frequency. We evaluate the results in terms of the root mean squared error (RMSE). The results are available in Table 7.

Out Of State Prediction

For our second task we decide to answer the question wheter we can reliably predict wheter a violation was commited by a visitor or a NY local. We define the task our task a binary classification of the *Registration State* attribute (Registration State == "NY"). For this task we use all of the augmented features at an hourly frequency.

We use the *Issue Date* and *Violation Time* columns to round the time of the violation to the nearest hour and combine it with the augmented features. From the available features

DATA TYPE	N WORKERS	Memory [Mb]		Times [s]				
		Mean	Max	Read Data	Data Augmentation	XGB	Linear Regregssion	SGD
PARQUET	1	328 ± 73	515	0.01	55.09675	0.67	9.30	0.34
	2	319 ± 78	471	0.01	34.99	0.66	9.23	0.35
	3	349 ± 88	512	0.01	27.40	0.65	9.40	0.34
DUCKDB	1	3032 ± 1112	3698	4.87	4.92	0.52	0.67	0.30
	2	3116 ± 1145	3937	5.05	4.98	0.52	0.70	0.32
	3	3053 ± 1120	3696	4.90	4.99	0.51	0.65	0.32
HDF5	1	4596 ± 3052	11204	12.50	77.18	2.83	1.39	0.60
	2	4921 ± 3137	11196	12.27	66.27	2.13	0.85	0.42
	3	4249 ± 3108	12364	12.41	53.07	2.13	0.87	0.42

Table 5: **T5b: Daily number of ticket prediction resource consumption.** We observe that our parquet implementation consumes by far the least memory, however this is traded off for the increased processing times as oposed to DuckDB.

we use the following subset: *Violation County*,*Violation Code*, *Vehicle Body Type*, V*ehicle Make*, *Issuing Agency*, *Violation Legal Code*, *From Hours In Effect*, *To Hours In Effect*, *Vehicle Year*, and *Feet From Curb*. We aggregate the categorical attributes of String data types based on our exploratory data analysis from Section T3 - Exploratory Data Analysis. We select the most common categories and convert the others to other, however we also make sure to first aggregate categories which have multiple different entries (e.g. "SDN" and "SEDN" both denote *Vehicle Body Type* sedan). Afterwards we one-hot encode the categorical variables with now reduced cardinalities. We encode the time data with seasonality cosine and sine frequencies at a yearly, monthly and daily level.

DATA TYPE	N WORKERS	Memory [Mb]		Times [s]				
		Mean	Max	Read Data	Data Augmentation	XGB	Logistic Regression	SGD
PARQUET	1	2691 ± 585	3670	0.0168130	33.360436916	15.8198621	290.69854	4.438780
	2	2426 ± 600	3368	0.0183370	29.691886	15.54641890	268.734862	3.889915
	3	2536 ± 573	3682	0.0177221	28.0054380	15.5464298	260.32085	3.7217259
DUCKDB	1	3535 ± 578	3724	4.926777	9.09383	5.88472	120.52267	1.91218
	2	3300 ± 511	3448	4.3164401	9.36235	5.827890	132.36339497	1.63496
	3	/	/	/	/	/	/	/
HDF5*	1	/	/	/	/	/	/	/
	2	/	/	/	/	/	/	/
	3	/	/	/	/	/	/	/

Table 6: **T5a: Out of State prediction resource consumption.** * When running the task with the hdf5 file format, the job timed out after 24 hours of runtime. As the code worked for other file formats and due to time constraints of we did not further debug the hdf5 code and leave take this as a limitation (likely due to our implementation) of the file format.

We evaluate the models in terms of accuracy and visualize the confusion matrix for the best performing predictive model is Figure 5. The results are available in Table 7.

T6 - Data Format & Scalability Comparison

We conduct the quantitative comparison throughout the report for each task and outline our qualitative findings in the conclusion.

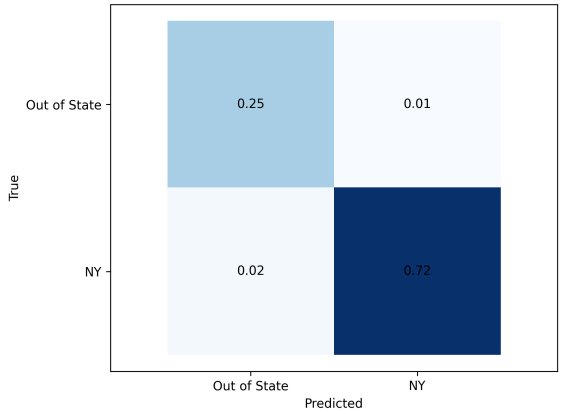


Figure 5: **Confusion matrix for the best performing model on "out of state" prediction** indicates that the model is able to predict whether a violation was produced by a visitor or a New Yorker reliably.

Parameter	XGBoost	Linear Model	SGD
T5a	**0.974**	0.969	0.968
T5b	5321	5307	**4970**

Table 7: **T5 Machine learning results.** Interestingly the best performing model on the classification task T5a is XGBoost and the worst SGD, while on the regression task T5b, the order is reversed. We draw this up to the XGBoost model overfiting to the training data on the T5b task, for which there is significantly less data than in the T5a task and the diferences between the train (to 2023) and test (2023 onwards) set are more pronounced.

T7 - Geographical Visualizations

We visualize the locations with most tickets in Figure 3c and the locations of schools by borough in Figure 1.

Conclusion

After all of the experiments we can with great confidence compare the utility and performance of different file types. We see that HDF5 has the most problems in usage with python and the Dask framework, with bugs in code. We believe this is because HDF5 is not really used that much anymore in modern data science and for that reason does not have that big of a community which would optimize it for modern data science pipelines. At the same time HDF5 is the most limited in terms of flexibility of data, which makes it unsuitable for unstructured data, which is very common in the modern data pipelines.

We were able to show the potential of Dask with Parquet files, where dask had no problem reading them efficiently in a distributed way and then processing them for either EDA or ML pipelines. We believe Parquet should be a de-facto standard for data science

with it's great file size and performance, however CSV is still a leading data type because of the ease of editability (parquet files are not editable with a text editor).

We also note the performance of DuckDB, however since it is not directly supported by Dask it is not appropriate for big data tasks with Dask. We believe DuckDB would be even more efficient with partial fit methods by scikit learn and with DuckDB query language used for reading the data in batches. Maybe this can be left for the next year's project.

A strong point of both DuckDB and Dask are their documentations (DuckDB Development Team, 2024; Dask Development Team, 2024). We would also like to point out the use of Dask for big data. Dask is great for it's distributed computing support, however it is very tedious to use with lots of bugs. We admit that we are not Dask experts and would need a lot more experience to fully use the potential of Dask, however we believe that using Hadoop/Spark might a better way to conquer big data. After reading the discussions online Dask is better than Hadoop/Spark for "not so big data" (Reddit: u/tuomas979, 2017; Reddit: u/mrocklin, 2023; Makait et al., 2024).

References

Dask Development Team. Dask documentation, 2024. URL https://docs.dask.org/en/stable/. https://docs.dask.org/en/stable/.

DuckDB Development Team. Duckdb documentation, 2024. URL https://duckdb.org/docs/. https://duckdb.org/docs/.

komoot. photon, 2024. URL https://photon.komoot.io/.

Hendrik Makait, Sarah Johnson, and Matthew Rocklin. Dataframes at scale comparison: Tpc-h. https://docs.coiled.io/blog/tpch.html, May 2024.

NYC Open Data. Nyc open data, 2024. URL https://opendata.cityofnewyork.us/.

Open-Meteo. Open-meteo, 2024. URL https://open-meteo.com/.

Reddit: u/mrocklin. Spark, dask, duckdb, polars: Tpc-h benchmarks at scale, 2023. URL https://www.reddit.com/r/Python/comments/17pwxfn/spark_dask_duckdb_polars_tpch_benchmarks_at_scale/.

Reddit: u/tuomas979. Why not just use dask instead of spark/hadoop?, 2017. URL https://www.reddit.com/r/datascience/comments/6hi5zc/why_not_just_use_dask_instead_of_sparkhadoop/.

Appendix A. Additional Information

Table 8 contains textual descriptions for each feature that we use to augment our data.

Column Name	Description
open_businesses	The number of open businesses in the borough.
num_events	The number of events occurring in the borough.
num_constructions	The number of construction activities in the borough.
num_street_closures	The number of street closures in the borough.
landmark_density	The density of landmarks in the borough.
total_landmark_length	The total circumference of landmarks in the borough.
total_landmarks	The total number of landmarks in the borough.
open_schools	The number of open schools in the borough.
temperature_2m (°C)	The air temperature at 2 meters above ground level.
relative_humidity_2m (%)	The relative humidity at 2 meters above ground level.
dew_point_2m (°C)	The dew point temperature at 2 meters above ground level.
apparent_temperature (°C)	The apparent temperature in degrees Celsius, accounting for wind chill or heat index.
precipitation (mm)	The amount of precipitation in millimeters.
rain (mm)	The amount of rainfall in millimeters.
snowfall (cm)	The amount of snowfall in centimeters.
snow_depth (m)	The depth of snow on the ground in meters.
weather_code (wmo code)	The WMO code representing the type of weather.
pressure_msl (hPa)	The atmospheric pressure at mean sea level in hectopascals.
surface_pressure (hPa)	The atmospheric pressure at the surface level in hectopascals.
cloud_cover (%)	The total cloud cover as a percentage.
cloud_cover_low (%)	The low-level cloud cover as a percentage.
cloud_cover_mid (%)	The mid-level cloud cover as a percentage.
cloud_cover_high (%)	The high-level cloud cover as a percentage.
et0_fao_evapotranspiration (mm)	The reference evapotranspiration in millimeters according to the FAO standard.
vapour_pressure_deficit (kPa)	The vapor pressure deficit in kilopascals.
wind_speed_10m (km/h)	The wind speed at 10 meters above ground level in kilometers per hour.
wind_speed_100m (km/h)	The wind speed at 100 meters above ground level in kilometers per hour.
federal_holiday	Indicator for whether the date is a federal holiday.
national_holiday	Indicator for whether the date is a national holiday.
religious_holiday	Indicator for whether the date is a religious holiday.
special_day	Indicator for whether the date is a special day (e.g., NYC-specific observances).
school_holiday	Indicator for whether the date is a school holiday.

Table 8: Descriptions of the additional columns incorporated into the dataset.