Analysis of Geospatial Data Loading

Aske Wachs
IT University of Copenhagen
Denmark
askw@itu.dk

Eleni Tzirita Zacharatou
IT University of Copenhagen
Denmark
elza@itu.dk

ABSTRACT

The rate at which applications gather geospatial data today has turned data loading into a critical component of data analysis pipelines. However, users are confronted with multiple file formats for storing geospatial data and an array of systems for processing it. To shed light on how the choice of file format and system affects performance, this paper explores the performance of loading geospatial data stored in diverse file formats using different libraries. It aims to study the impact of different file formats, compare loading throughput across spatial libraries, and examine the micro-architectural behavior of geospatial data loading. Our findings show that GeoParquet files provide the highest loading throughput across all benchmarked libraries. Furthermore, we note that the more spatial features per byte a file format can store, the higher the data loading throughput. Our micro-architectural analysis reveals high instructions per cycle (IPC) during spatial data loading for most libraries and formats. Additionally, our experiments show that instruction misses dominate L1 cache misses, except for GeoParquet files, where data misses take over.

CCS CONCEPTS

• **Information systems** → **Database performance evaluation**; **Geographic information systems**.

KEYWORDS

spatial libraries, benchmarking, micro-architectural analysis

ACM Reference Format:
Aske Wachs and Eleni Tzirita Zacharatou. 2024. Analysis of Geospatial Data Loading. In *International Workshop on Testing Database Systems (DBTest '24), June 9, 2024, Santiago, AA, Chile*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3662165.3662761

1 INTRODUCTION

Applications in both science and industry accumulate geospatial data at an unprecedented rate from a plethora of sources such as GPS-enabled devices (e.g., cell phones, cars, and sensors), scientific simulations [31], consumer-based applications (e.g., Uber), and social media platforms (e.g., location-tagged posts on Facebook, X, and Instagram). To meet increasing application demands, there is a considerable amount of research on improving spatial data processing [1, 19, 27, 29, 30, 32].

DBTest '24, June 9, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0669-1/24/06
https://doi.org/10.1145/3662165.3662761

Geospatial data is captured in various file formats, such as Shapefile and GeoParquet [10]. Remarkably, OGR [8], one of the most widely used spatial data libraries, supports 83 spatial vector file formats. Extracting value from large amounts of data effectively requires loading it into a data processing or GIS system. However, the landscape of spatial systems and libraries is very diverse, with each tool supporting only a subset of existing spatial file formats and exhibiting varying data loading efficiency. This plethora of different tools and file formats poses challenges in choosing the best tool for a specific format. However, this choice is crucial as loading large amounts of data can easily become an analysis bottleneck.

Geospatial data analytics have been extensively benchmarked both in literature and anecdotally [6, 11, 14, 17, 18, 21, 35]. However, no benchmarks focus on data loading performance and the impact of the file format. Existing spatial benchmarks focus on the performance of specific operations, for example, spatial joins [6, 17, 18], topological relations, such as intersections, overlaps, or crossing features [14, 21], and exploratory analytics workloads [35]. Furthermore, a recent anecdotal blogpost presents a performance exploration of writing vector data with different file formats and libraries [11]. A prior data loading performance study focuses on relational databases [4] and does not consider spatial data and systems. Finally, existing micro-architectural studies do not consider spatial workloads either [22, 24, 25, 28]. They range from investigating the behavior of online transaction processing (OLTP) [25, 28], to online analytical processing (OLAP) [24] and graph [22] workloads.

This paper presents a comprehensive analysis of geospatial data loading with the following goals: 1) analyze how different file formats impact data loading, 2) compare the loading throughput of different spatial libraries, 3) understand the stress that geospatial data loading imposes on processor and caches across different file formats and libraries, and 4) provide insights to data scientists for optimizing their pipeline and avoiding loading bottlenecks.

The results of our analysis reveal the following:

- GeoParquet is the fastest file format for reading across all libraries that we benchmarked. Furthermore, DuckDB's [20] native GeoParquet reader achieves the highest loading throughput and can exploit parallelism.
- Data density, measured by the number of features stored per byte, correlates with loading throughput: denser files are loaded faster. However, this correlation no longer holds when loading in parallel with DuckDB.
- Examining the micro-architectural behavior of spatial data loading, we observe that it exhibits high instructions per cycle (IPC). Furthermore, instruction misses constitute the majority of misses in the L1 cache, except for GeoParquet files where data misses dominate.

The rest of the paper is organized as follows. Section 2 describes our experimental setup and methodology. Section 3 presents an

experimental comparison of different spatial file formats and loading approaches. Section 4 presents a micro-architectural analysis of spatial data loading in DuckDB. Sections 5 and 6, respectively, present a micro-architectural analysis of loading CSV and GeoParquet files using four different loading approaches. Finally, Section 7 concludes the paper and discusses future work.

2 SETUP AND METHODOLOGY

The goal of our experimental study is to investigate the impact of GIS library and file format on spatial data loading. We executed the experiments presented in this paper on real hardware and measured the performance using `perf` event counters. The rest of this section details the setup and methodology for our study.

Hardware. We used a desktop system for our experiments. Table 1 lists the system characteristics.

CPU name	Ryzen 5 5600G
CPU clock speed	3.9-4.65[0] GHz
Cores (threads/core)	6 (2)
CPU architecture	AMD *Zen* 3
Sockets	1
Cache line size	64 bytes
L1 data cache	32 KB per core
L1 instruction cache	32 KB per core
L2 unified cache size	512 KB per core
L3 unified cache size	16 MB shared
L1 data TLB entries	64 per core
L1 instruction TLB entries	64 per core
L2 data TLB entries	2048 per core
L2 instruction TLB entries	512 per core
RAM capacity	32 GB
Page Size	4 KB
Operating System	Ubuntu Server 22.04.3 LTS

Table 1: System Properties.

Analyzed File Formats. We analyze four widely-supported spatial file formats: Shapefile [5], GeoJSON [2], CSV [23], and GeoParquet [3]. Shapefile is a multi-file format including a main file storing geometric data, as well as index, attribute, and projection files. GeoJSON is a text-based row-major format that represents geographic features and their associated properties using JSON syntax. CSV files, despite their simplicity, are often used to store spatial data. Lastly, GeoParquet is a columnar binary format that enhances Parquet by specifying how to encode geometries in the geometry column and incorporating spatial metadata such as Coordinate Reference System (CRS) information.

Analyzed Libraries. We analyze three different GIS libraries: DuckDB [20], GeoPandas [12], and OGR [8]. DuckDB is a state-of-the-art analytical database that recently launched a spatial extension [7]. GeoPandas is an extension of the popular Pandas library [15]. Both GeoPandas and DuckDB use OGR under the hood to load some of the analyzed file formats. Therefore, by including OGR in our evaluation, we aim to investigate the impact of hooking into it for data loading. DuckDB was built by cloning the DuckDB Spatial repository at commit `afd6452` on the `main` branch. For its OGR-backed data loading table function (`st_read`), DuckDB uses GDAL 3.6.3 in this commit. As of the latest DuckDB release (v0.10.0), the GDAL version has been upgraded to 3.8.0. DuckDB uses `st_read` to load Shapefiles and GeoJSON. However, for CSV and GeoParquet files, we use DuckDB's native readers and then convert geometric feature data into DuckDB's geometry types using a scalar function provided by DuckDB's spatial extension. GeoPandas is version 0.14.0, with Pandas [15] 1.4.0, Pyproj [34] 3.3.0, Shapely [9] 1.8.0 and Pyogrio [33] 0.7.2. OGR was built from the source with GeoParquet support by cloning the GDAL repository at commit `2cfeb1a` on the `master` branch. This commit has been integrated into releases since version 3.8.0.

Concurrency. OGR and GeoPandas do not inherently support parallel loading, except when loading GeoParquet files. In contrast, DuckDB offers parallel loading for all file formats. Therefore, we evaluate two versions of DuckDB: a single-threaded version and a multi-threaded version utilizing all 12 hardware threads available. For GeoParquet, OGR and GeoPandas default to using a maximum of 4 and 6 (i.e., the number of available cores) threads, respectively.

Datasets. We generated synthetic spatial vector datasets using SpiderWeb [13]. Specifically, we generated four polygon datasets of increasing size (containing 1, 2, 4, and 8 million polygons) using a Gaussian distribution for both x and y coordinates. We fixed the seed for random number generation to 5 and the maximum number of line segments per polygon to 10. Finally, we use the default affine matrix, generating coordinates where $0 \leq x, y \leq 1$. SpiderWeb saves the generated data in a GeoJSON file, which we subsequently converted to Shapefile, CSV, and GeoParquet formats using the `ogr2ogr` tool [8]. GeoParquet files are compressed using the default `Snappy` compression provided by the `ogr2ogr` tool. Table 2 presents an overview of the synthetic datasets used in our study. We note that GeoParquet files, being compressed, have the smallest size among the formats. In contrast, the verbosity and redundancy of GeoJSON syntax contribute to its larger file size. CSV files are over $2x$ larger than GeoParquet, while Shapefiles are slightly less than $2x$ the size of GeoParquet.

Furthermore, we use a real dataset from OpenStreetMap (OSM) [16] that corresponds to the map of Denmark and contains 6,799,943 features of different geometries.

Metrics. The metrics used in this study are data loading throughput (# of features per time unit), instructions per cycle (IPC), and data and instruction misses. To ease the understanding of our results, here we provide some background information on out-of-order processors.

Out-of-order processors have two main building blocks: frontend and backend. The frontend is responsible for handling the initial stages of instruction processing. Its key functions include fetching, decoding, and issuing instructions. The instructions are fetched from the memory hierarchy. Ideally, the next instructions to be executed should be found in the L1 instruction cache, which is the closest to the core. If a miss occurs, instructions need to be fetched from higher levels in the memory hierarchy, incurring higher latency and stalling the instruction pipeline. Preventing stalls in the frontend is crucial, since they unavoidably lead to underutilization of the backend too, decreasing the IPC.

[0] AMD's specified clock speed for the CPU is 4.4 GHz but the CPU is overclocked to 4.65 GHz on this system.

# of polygons	Total # of points	Avg. # of points/polygon	Size (GeoJSON)	Size (Shapefile)	Size (CSV)	Size (GeoParquet)
1,000,000	8,004,343	8.004	410MB	202MB	295MB	122MB
2,000,000	16,002,087	8.001	820MB	404MB	591MB	244MB
4,000,000	32,004,894	8.001	1.7GB	808MB	1.2GB	488MB
8,000,000	64,004,157	8.001	3.3GB	1.6GB	2.4GB	975MB

Table 2: Synthetic Data Statistics

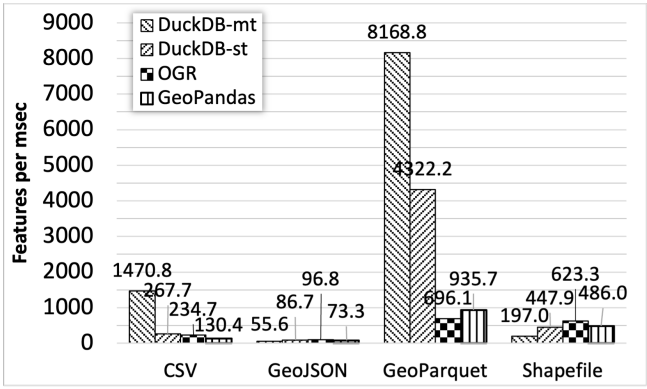


Figure 1: Throughput (synthetic dataset, 8M polygons)

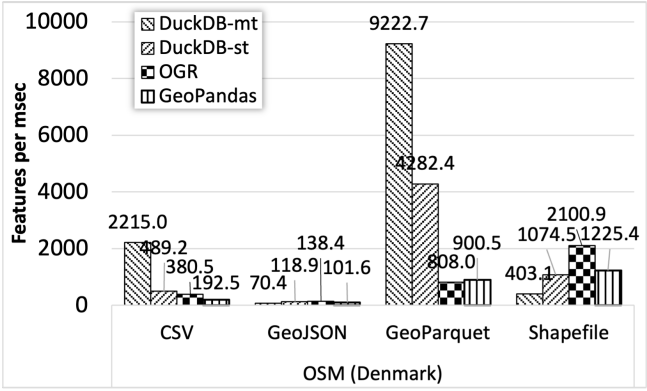


Figure 2: Throughput (real OSM dataset)

The backend executes the micro-operations (µOps) issued by the frontend. Micro-operations are registered in reservation stations, which track the operands and dependencies for each µOp. The backend includes various execution units that perform the actual operation specified by the µOps. These units can operate in parallel, allowing multiple instructions to be executed simultaneously, unless there are dependencies between them. Furthermore, the backend buffers outstanding data load and store requests, minimizing stall time caused by memory misses and improving overall efficiency. Once all operands are available, the µOp is executed and retired.

We measure the instructions per cycle (IPC) during data loading. AMD Zen 3 processors can retire four instructions per cycle, i.e., the maximum achievable IPC is four. Since instruction and data cache misses are primary sources of stalls, effectively reducing the IPC, we also measure the number of misses per 1K instructions (MPKI) in the L1 cache and the L1 TLB.

Measurements. We use the `perf stat` tool to collect performance counter statistics. We first copy the files from disk to main memory using `/dev/shm`. This prevents disk I/O bottlenecks from impacting the results. For each GIS library, we developed a Python script that takes a file path as an argument and loads the specified in-memory data file using the respective library. Additionally, we created a benchmark harness that invokes these Python scripts using the `subprocess` module and profiles them end-to-end with perf. Running this harness profiles all libraries for all file formats. The reported results are an average of 10 executions.

3 COMPARATIVE ANALYSIS

Figure 1 plots the *throughput* measured as the number of features (polygons) loaded per millisecond for the largest synthetic dataset (8 million polygons). DuckDB-mt excels at loading GeoParquet, outperforming OGR by 12x and GeoPandas by 9x. Even when restricting DuckDB to one thread (DuckDB-st), it achieves a 6x and 5x higher throughput than OGR and GeoPandas, respectively. GeoPandas and OGR exhibit their highest throughput for GeoParquet compared to other formats at 935.7 and 696.1 features/msec, respectively. Overall, GeoParquet stands out as the most efficient file format in terms of reading throughput. This can be attributed to storing data more densely in the form of Well-Known Binary (WKB) [26] and having a smaller file size thanks to the applied compression, as shown in Table 2. Furthermore, all libraries support some amount of parallelism when loading GeoParquet files, as explained in Section 2.

In the multi-threaded configuration, DuckDB's native CSV reader outperforms OGR and GeoPandas by factors of 6x and 11x, respectively, while in the single-threaded setup, it achieves a slightly higher throughput than OGR and is around 2x faster than GeoPandas. When loading GeoJSON, all libraries demonstrate a low throughput, below 100 features/msec. This is mainly attributed to the substantial file size of GeoJSON (cf. Table 2) and the intensive string parsing involved in the loading process.

OGR takes the lead for GeoJSON and Shapefile. This is expected, given that all other libraries use OGR to load these file formats, as explained in Section 2. Moreover, we observe that when employing DuckDB's `st_read` function for Shapefiles and GeoJSON, there is a decrease in throughput with multi-threading. This might happen because `st_read` does not process data fast enough to feed all threads, causing threads to be suspended until new data arrives and resulting in frequent context switches.

Figure 2 plots the throughput for the real OSM dataset, demonstrating that the results align with the trends observed in Figure 1.

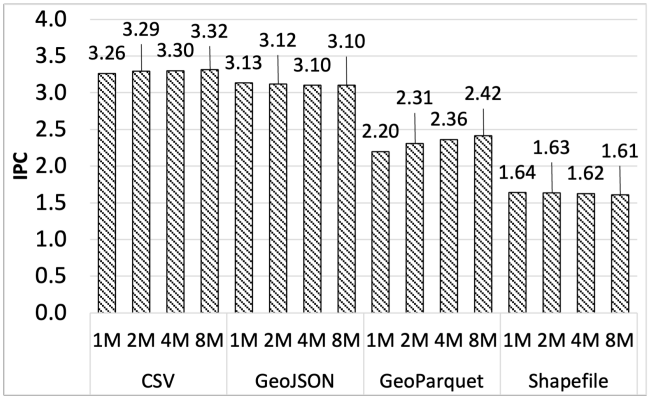


Figure 3: Instructions committed per cycle when loading different file formats and file sizes with DuckDB-st.

Overall, our experiments highlight that the choice of both library and file format can significantly impact the loading performance. Specifically, the throughput difference between the fastest (GeoParquet) and the slowest (GeoJSON) format in DuckDB exceeds two orders of magnitude. Additionally, loading GeoParquet in parallel in DuckDB is over one order of magnitude faster than in OGR.

4 DUCKDB ANALYSIS

We conduct a micro-architectural analysis of DuckDB using synthetic data to gain a deeper insight into its loading performance across different file formats. To avoid thread interference, here we focus on the single-threaded version of DuckDB. The next sections extend our analysis to include multi-threaded DuckDB.

4.1 Instruction-level Parallelism

We analyze the number of instructions per cycle (IPC) in Figure 3. We observe that for both CSV and GeoJSON, the IPC is consistently high, above 3, which indicates that there are not many dependencies between µOps. GeoParquet exhibits a slightly lower IPC, ranging from 2.2 for 1M polygons to 2.42 for 8M polygons. Finally, Shapefile loading incurs the lowest IPC, albeit remaining above 1. This lower IPC could be a result of dependencies between instructions when loading the Shapefile that the out-of-order execution cannot efficiently resolve. These dependencies could be caused by the more complex, multi-file structure of Shapefiles, as explained in Section 2.

4.2 Data and Instruction Misses

Figure 4 shows the misses per k-instructions (MPKI) in the L1 cache on the left-hand side and in the L1 TLB on the right-hand side. We categorize misses as L1 instruction cache misses (L1i), L1 data cache misses (L1d), L1 instruction TLB misses (L1 iTLB), and L1 data TLB misses (L1 dTLB). We observe that L1 instruction cache misses dominate the total number of misses regardless of the file size for all file formats except GeoParquet. This is not surprising given that we are loading the entire file from start to end, so all the parts of the cache lines brought from a file page are accessed, leading to lower data MPKI. CSV exhibits the lowest values of MPKI compared to other formats (around 6), followed by GeoJSON with around 11 MPKI, and GeoParquet with 14-17 MPKI. In contrast, Shapefile has the highest MPKI with around 56 MPKI. This may be attributed to the complex multi-file structure of Shapefile, leading to a larger instruction footprint.

In the case of GeoParquet, L1 data cache misses dominate the total misses. The reduced number of instruction misses can be attributed to the vectorized data loading of GeoParquet files, which increases instruction-level parallelism. Additionally, we observe a slight decrease in the number of instruction misses with larger file sizes. This is because as the data loading instructions are iteratively executed over more data, the impact of instructions outside the loading function decreases. In other words, the overhead of startup and shutdown instructions gets amortized. This observation aligns with prior micro-architectural studies [25]. Finally, the higher number of L1 data cache misses might result from the dictionary used for Snappy decompression exceeding the L1 cache capacity.

In the L1 TLB, data misses dominate the total number of misses for GeoJSON and GeoParquet, while instruction misses dominate for CSV. Shapefile experiences a roughly equal distribution of misses between data and instruction. Similarly to the L1 cache misses, CSV incurs the lowest number of MPKI, while Shapefile exhibits the highest, with GeoJSON now incurring more misses than GeoParquet. The file size seems to have a small impact on GeoParquet and Shapefile, leading to a decrease in data misses for the former and an increase for the latter.

5 CSV LOADING ANALYSIS

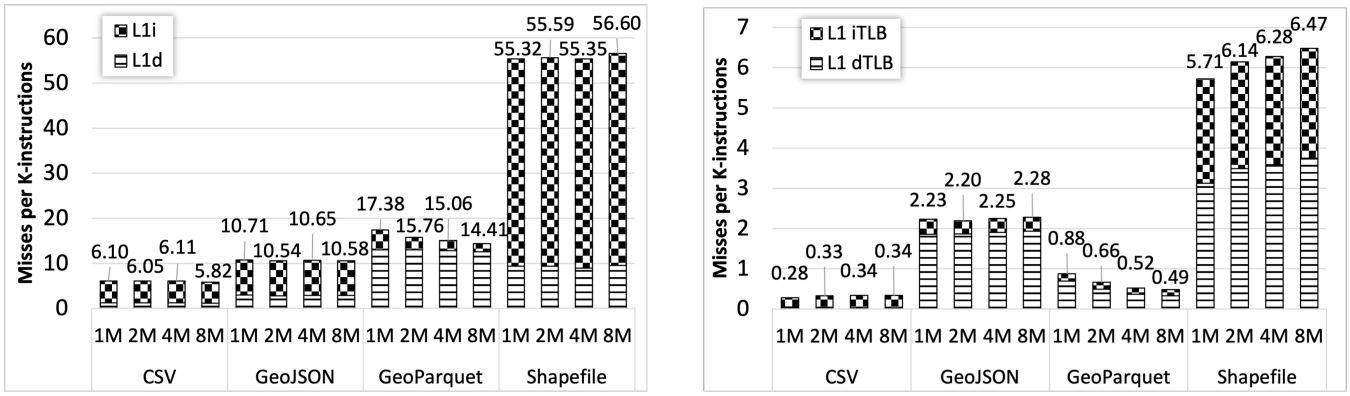
Our next round of experiments focuses on the CSV loading performance across different libraries using synthetic data.

5.1 Instruction-level Parallelism

We analyze the number of instructions per cycle (IPC) in Figure 5a. DuckDB-st consistently maintains the highest IPC for all file sizes. This highlights the resource efficiency of DuckDB's native CSV reader, which is also reflected in the achieved throughput, as shown in Figure 1. We also note that the IPC decreases in multi-threaded DuckDB. This can be attributed to Simultaneous Multi-Threading (SMT) used by AMD Zen 3 processors, where two threads share the resources of a single core. Consequently, the concurrent execution of multiple threads may lead to resource contention, causing more stalls and ultimately reducing the IPC. Finally, all libraries except DuckDB-mt show an increase in the IPC as the file size increases. This implies that larger file sizes provide more opportunities for instruction-level parallelism.

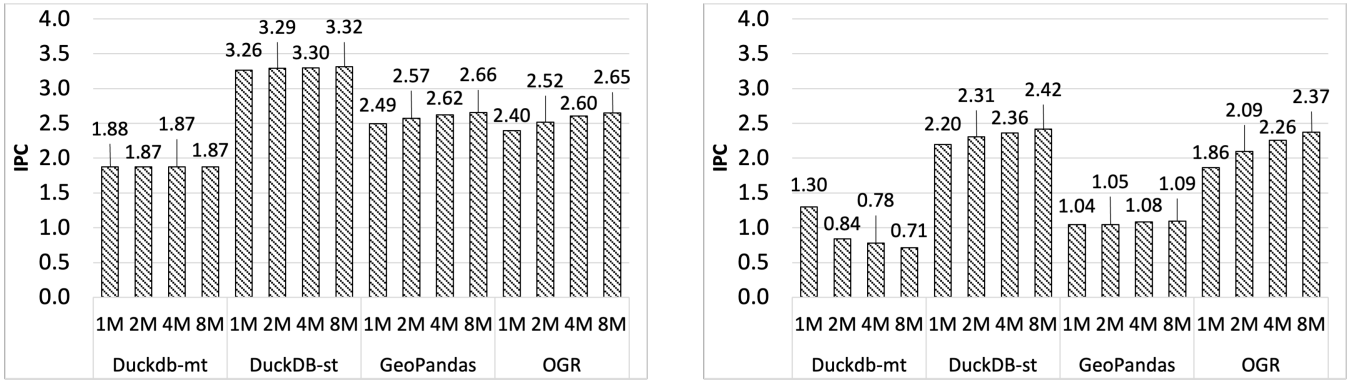
5.2 Data and Instruction Misses

Figure 6 shows the misses per k-instructions in the L1 cache (left-hand side) and L1 TLB (right-hand side), categorized into instruction and data misses. We observe that instruction misses consistently dominate data misses in the L1 cache across all libraries and file sizes. The low data MPKI values can be attributed to the sequential loading of entire files, ensuring access to all parts of the cache lines fetched from a file page. While DuckDB (both single- and multi-threaded) maintains low MPKI values, GeoPandas and OGR suffer from a higher number of misses when loading CSV files.



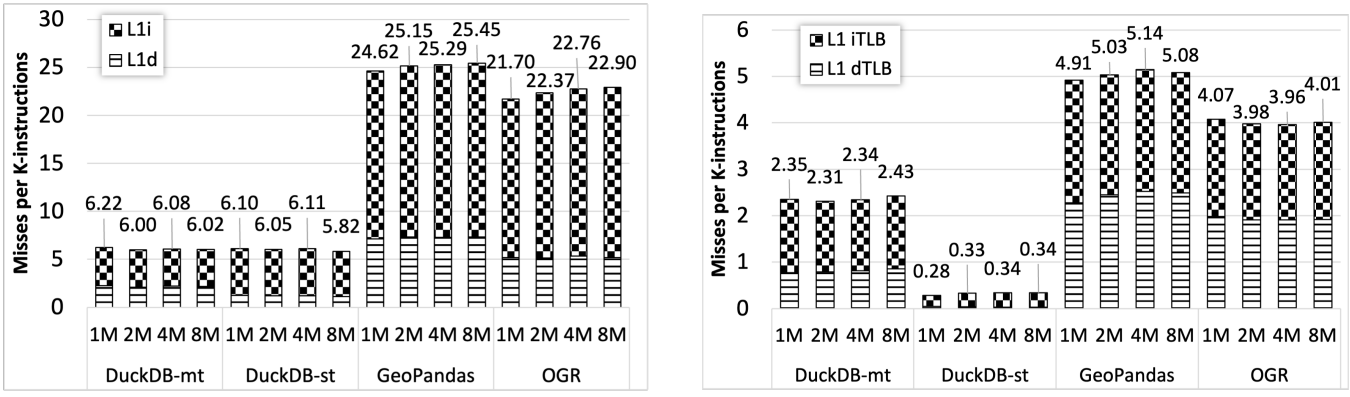
(a) L1 cache (b) L1 TLB

Figure 4: Effect of file format and file size on MPKI when loading data with DuckDB-st.



(a) CSV Loading (b) GeoParquet Loading

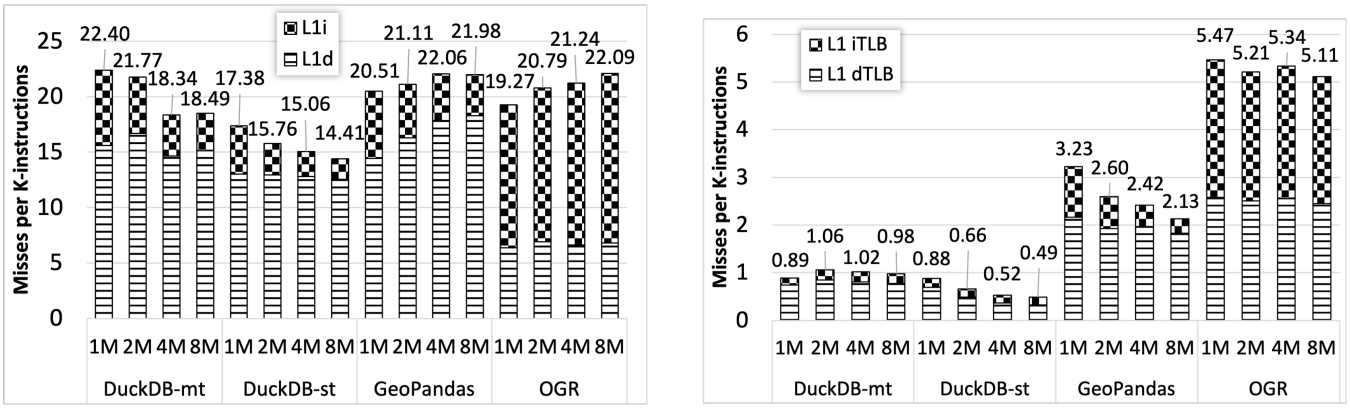
Figure 5: Instructions committed per cycle when loading files of varying sizes with different libraries.



(a) L1 cache (b) L1 TLB

Figure 6: Effect of library and file size on MPKI for CSV loading.

In the TLB, DuckDB once again experiences more instruction than data misses and has the lowest MPKI values. Additionally, DuckDB-mt exhibits a higher miss rate than DuckDB-st, which can be attributed to potential TLB pollution by multiple threads. Finally, in GeoPandas and OGR, TLB misses are nearly evenly divided between instruction and data, with GeoPandas having a higher



(a) L1 cache **(b) L1 TLB**

Figure 7: Effect of library and file size on MPKI for GeoParquet loading.

number of instruction misses. The higher instruction miss rate in GeoPandas can be explained by the fact that it executes a larger number of instructions than other libraries.

6 GEOPARQUET LOADING ANALYSIS

The last round of experiments focuses on the performance of loading GeoParquet files with different libraries using synthetic data.

6.1 Instruction-level Parallelism

We analyze the number of instructions per cycle (IPC) in Figure 5b. We observe that all libraries exhibit lower IPC values for GeoParquet loading compared to CSV loading. While DuckDB-st has the highest IPC for all file sizes, DuckDB-mt experiences a significant drop in the IPC. As discussed before, this drop could be attributed to the resource contention caused by SMT. This low IPC could explain why parallelizing parsing and loading with 12 hardware threads fails to achieve more than a $1.9x$ increase in throughput compared to DuckDB-st, as shown in Figure 1. Finally, as in the case of CSV, all libraries except DuckDB-mt show an increase in the IPC with increasing file size. This is because the same data loading instructions can be reused in a loop for an increasingly large amount of data, while the impact of executed instructions outside the loading function decreases.

6.2 Data and Instruction Misses

Figure 7 shows the MPKI in the L1 cache (left-hand side) and the L1 TLB (right-hand side). As before, we break down the misses into instruction and data. Unlike CSV loading where instruction misses dominated, GeoParquet loading is dominated by data misses for all libraries except OGR. While all libraries exhibit similar MPKI values, DuckDB experiences a slight decrease in misses, while the other two libraries observe an increase with larger file sizes.

In the TLB, data misses dominate except in OGR, where they are roughly evenly divided between data and instruction. DuckDB has a low miss rate, mostly below 1 MPKI. GeoPandas follows, with misses decreasing from 3.23 to 2.13 MPKI as the file size increases. OGR comes last with around 5 MPKI.

Finally, looking at the MPKI for the largest file of 8 million polygons, we notice that OGR experiences the highest rate of misses in both the L1 cache and the L1 TLB. This correlates with the fact that it exhibits the lowest throughput, as shown in Figure 1.

7 CONCLUSION

Given the amount of geospatial data gathered by applications today, reducing the overhead of data loading is crucial to prevent it from becoming a bottleneck in data analysis pipelines. This paper studies geospatial data loading across three spatial libraries and four popular spatial file formats in terms of loading throughput and micro-architectural behavior.

Our analysis shows that GeoParquet files provide the highest data loading throughput overall. Given that GeoParquet is a binary format, it stores geospatial data more densely than the text-based CSV and GeoJSON formats. Furthermore, its column-oriented layout enables fast vectorized execution. DuckDB outperforms all other libraries in loading GeoParquet files, while it achieves a further $2x$ increase in loading throughput using parallelization. Overall, data density in terms of the number of features per byte a file format can store correlates with loading speed. The only exception is multi-threaded DuckDB that can load CSVs faster than Shapefiles, despite Shapefiles being more dense. This is because Shapefiles are loaded into DuckDB using an OGR-backed data loading table function (`st_read`). In contrast, CSVs are loaded using DuckDB's native CSV reader, which is more efficient. Finally, we observe that while instruction misses dominate the total number of misses in the L1 cache, this is not the case for GeoParquet files.

This study focuses on loading entire files. However, some file formats, such as GeoParquet, support data loading optimizations like data skipping, while Shapefiles include an index file allowing fast seeking. In future work, we plan to investigate the efficiency of such data loading optimizations.

REFERENCES

[1] Ahmet Kerem Aksoy, Pavel Dushev, Eleni Tzirita Zacharatou, Holmer Hemsen, Marcela Charfuelan, Jorge-Arnulfo Quiané-Ruiz, Begüm Demir, and Volker Markl. 2022. Satellite image search in AgoraEO. *PVLDB* 15, 12 (2022), 3646–3649.

[2] H. Butler, M. Daly, A. Doyle, Sean Gillies, T. Schaub, and Stefan Hagen. 2016. *The GeoJSON Format*. Technical Report 7946. https://doi.org/10.17487/RFC7946

[3] GeoParquet Community. 2023. GeoParquet. https://geoparquet.org/. Accessed on March 5, 2024.

[4] Adam Dziedzic, Manos Karpathiotakis, Ioannis Alagiannis, Raja Appuswamy, and Anastasia Ailamaki. 2016. DBMS Data Loading: An Analysis on Modern Hardware. In *ADMS (Lecture Notes in Computer Science, Vol. 10195)*. Springer, 95–117.

[5] Esri 2023. *ShapeFiles - ArcGIS Online Reference Manual*. Esri. https://doc.arcgis.com/en/arcgis-online/reference/shapefiles.htm Accessed on March 5, 2024.

[6] Martin Fleischmann. 2022. Dask-GeoPandas vs PostGIS vs GPU: Performance and Spatial Joins. https://martinfleischmann.net/dask-geopandas-vs-postgis-vs-gpu-performance-and-spatial-joins/. Accessed on March 5, 2024.

[7] Max Gabrielsson. 2023. *PostGEESE? Introducing The DuckDB Spatial Extension*. DuckDB Foundation. https://duckdb.org/2023/04/28/spatial.html Accessed on March 5, 2024.

[8] GDAL/OGR contributors. 2023. *GDAL/OGR Geospatial Data Abstraction software Library*. Open Source Geospatial Foundation. https://doi.org/10.5281/zenodo.5884351

[9] Sean Gillies, Casper van der Wel, Joris Van den Bossche, Mike W. Taves, Joshua Arnott, Brendan C. Ward, et al. 2024. *Shapely*. https://doi.org/10.5281/zenodo.10671398

[10] GISGeography. 2023. *The Ultimate List of GIS Formats and Geospatial File Extensions*. https://gisgeography.com/gis-formats/ Last updated: March 9, 2024, Accessed: March 14, 2024.

[11] Chris Holmes. 2023. *Performance Explorations of GeoParquet (and DuckDB)*. Cloud-Native Geospatial Foundation. https://cloudnativegeo.org/blog/2023/08/performance-explorations-of-geoparquet-and-duckdb/ Accessed on March 5, 2024.

[12] Kelsey Jordahl, Joris Van den Bossche, Martin Fleischmann, Jacob Wasserman, James McBride, Jeffrey Gerard, Jeff Tratner, Matthew Perry, Adrian Garcia Badaracco, Carson Farmer, Geir Arne Hjelle, Alan D. Snow, Micah Cochran, Sean Gillies, Lucas Culbertson, Matt Bartos, Nick Eubank, maxalbert, Aleksey Bilogur, Sergio Rey, Christopher Ren, Dani Arribas-Bel, Leah Wasser, Levi John Wolf, Martin Journois, Joshua Wilson, Adam Greenhall, Chris Holdgraf, Filipe, and François Leblanc. 2020. *geopandas/geopandas: v0.8.1*.

[13] Puloma Katiyar, Tin Vu, Ahmed Eldawy, Sara Migliorini, and Alberto Belussi. 2020. SpiderWeb: A Spatial Data Generator on the Web. In *SIGSPATIAL*. ACM, New York, NY, USA, 465–468.

[14] Suneuy Kim and Yuvraj Singh Kanwar. 2019. GeoYCSB: A Benchmark Framework for the Performance and Scalability Evaluation of NoSQL Databases for Geospatial Workloads. In *Big Data*. IEEE, 3666–3675.

[15] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Python in Science Conference*, Stéfan van der Walt and Jarrod Millman (Eds.). 51 – 56.

[16] OpenStreetMap. 2024. https://www.openstreetmap.org.

[17] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. 2018. How good are modern spatial analytics systems? *PVLDB* 11, 11 (2018), 1661–1673.

[18] Varun Pandey, Alexander van Renen, Andreas Kipf, and Alfons Kemper. 2021. How Good Are Modern Spatial Libraries? *Data Sci. Eng.* 6, 2 (2021), 192–208.

[19] Varun Pandey, Alexander van Renen, Eleni Tzirita Zacharatou, Andreas Kipf, Ibrahim Sabek, Jialin Ding, Volker Markl, and Alfons Kemper. 2023. Enhancing In-Memory Spatial Indexing with Learned Search. arXiv:2309.06354 [cs.DB]

[20] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *SIGMOD*. ACM, New York, NY, USA, 1981–1984.

[21] Suprio Ray, Bogdan Simion, and Angela Demke Brown. 2011. Jackpine: A benchmark to evaluate spatial database performance. In *ICDE*. IEEE, 1139–1150.

[22] Rathijit Sen and Yuanyuan Tian. 2023. Microarchitectural Analysis of Graph BI Queries on RDBMS. In *DAMON*. ACM, New York, NY, USA, 102–106.

[23] Yakov Shafranovich. 2005. Common Format and MIME Type for Comma-Separated Values (CSV) Files. RFC 4180. https://doi.org/10.17487/RFC4180

[24] Utku Sirin and Anastasia Ailamaki. 2020. Micro-architectural Analysis of OLAP: Limitations and Opportunities. *PVLDB* 13, 6 (2020), 840–853.

[25] Utku Sirin, Pinar Tözün, Danica Porobic, and Anastasia Ailamaki. 2016. Micro-architectural Analysis of In-memory OLTP. In *SIGMOD*. ACM, 387–402.

[26] Knut Stolze. 2003. SQL/MM spatial: The standard to manage spatial data in a relational database system. In *Datenbanksysteme für Business, Technologie und Web (BTW)*.

[27] Ruby Y. Tahboub and Tiark Rompf. 2020. Architecting a Query Compiler for Spatial Workloads. In *SIGMOD*. 2103–2118.

[28] Pinar Tözün, Ippokratis Pandis, Cansu Kaynak, Djordje Jevdjic, and Anastasia Ailamaki. 2013. From A to E: analyzing TPC's OLTP benchmarks: the obsolete, the ubiquitous, the unexplored. In *EDBT*. ACM, 17–28.

[29] Eleni Tzirita Zacharatou, Harish Doraiswamy, Anastasia Ailamaki, Cláudio T. Silva, and Juliana Freire. 2017. GPU Rasterization for Real-Time Spatial Aggregation over Arbitrary Polygons. *Proc. VLDB Endow.* 11, 3 (2017), 352–365.

[30] Eleni Tzirita Zacharatou, Andreas Kipf, Ibrahim Sabek, Varun Pandey, Harish Doraiswamy, and Volker Markl. 2021. The Case for Distance-Bounded Spatial Approximations. In *Conference on Innovative Data Systems Research, CIDR, Virtual Event, Online Proceedings*. http://cidrdb.org/cidr2021/papers/cidr2021_paper19.pdf

[31] Eleni Tzirita Zacharatou, Darius Sidlauskas, Farhan Tauheed, Thomas Heinis, and Anastasia Ailamaki. 2019. Efficient Bundled Spatial Range Queries. In *ACM SIGSPATIAL*. 139–148. https://doi.org/10.1145/3347146.3359077

[32] Tin Vu, Ahmed Eldawy, Vagelis Hristidis, and Vassilis J. Tsotras. 2021. Incremental Partitioning for Efficient Spatial Data Analytics. *PVLDB* 15, 3 (2021), 713–726.

[33] Brendan C. Ward. 2024. https://pypi.org/project/pyogrio/

[34] Jeff Whitaker. 2024. https://pypi.org/project/pyproj/

[35] Yaming Zhang and Ahmed Eldawy. 2020. Evaluating computational geometry libraries for big spatial data exploration. In *GeoRich*. ACM, New York, NY, USA, Article 3.