The Journal of Open Source Software

ParquetDB: A Lightweight Python Parquet-Based Database

Logan L. Lang[1], **Eduardo Hernandez**[2], **Kamal Choudhary**[3], and **Aldo H. Romero**[1]

1 Department of Physics, West Virginia University, Morgantown, WV 26506, United States **2** Instituto de Ciencia de Materiales de Madrid, Campus de Cantoblanco, C. Sor Juana Inés de la Cruz, 3, Fuencarral-El Pardo, Madrid 28049, Spain **3** National Institute of Standards and Technology, 100 Bureau Dr, Gaithersburg, MD 20899, United States

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Summary

ParquetDB is a Python library designed to bridge the gap between traditional file storage and fully fledged databases, all while wrapping the PyArrow library to streamline data input and output. By leveraging the Parquet file format, ParquetDB provides the portability and simplicity of file-based data storage alongside advanced querying features typically found in database systems. Because ParquetDB is built on top of PyArrow, it seamlessly handles data types that need special processing to be compatible with the Parquet format. This reduces manual conversion and boilerplate code, allowing developers to focus on higher-level data operations. In addition, the Parquet format's columnar storage and rich metadata make it possible to efficiently perform predicate and column pushdown, leading to faster queries by reading only the subsets of data you truly need.

Statement of need

In an era where data is the driving force behind innovation, the demand for highly efficient, scalable, and adaptable storage solutions has never been greater. Traditional file-based storage formats (e.g., CSV, JSON, TXT) and database systems (e.g., SQLite(*About SQLite*, n.d.), MongoDB(*MongoDB*, n.d.)) have historically powered data handling in numerous applications (Habyarimana & Michailidou, 2021) (Jain et al., 2013) (*Well-Known Users Of SQLite*, n.d.) ("Customer Case Studies," n.d.). However, both approaches exhibit unique limitations that can impede rapid experimentation, large-scale research, and data-intensive development.

File-based solutions are popular for their simplicity and portability, often relying on straightforward ASCII/UTF encoding. This design choice, while human-readable, becomes highly inefficient for numerical data. For example, encoding an integer like 127 in ASCII demands three separate bytes (00110001 00110010 00110111), leading to significant overhead as data volumes expand. Such inflation in file size translates into slower input/output (I/O) operations and increased storage requirements, ultimately restricting scalability. Additionally, file-based formats typically lack built-in querying capabilities and indexing features, forcing developers to manage complex data relationships manually. These constraints limit the agility of workflows, especially as projects grow in complexity or require quick iteration cycles.

Conversely, traditional database management systems offer robust encoding, indexing, and querying capabilities out of the box. Relational databases, for instance, enforce structured schemas that ensure data integrity but introduce complexities when the data model evolves over time. Non-relational databases, such as document-oriented or key-value stores, are more flexible but risk data inconsistency and can become cumbersome to optimize for performance(Singh, 2024). Many of these solutions require dedicated servers or intricate configurations, increasing overhead for lightweight experimentation. Moreover, the underlying architectures—whether row-based or reliant on specialized storage engines—can exhibit performance bottlenecks when handling unstructured or semi-structured data at scale.

ParquetDB is intended to be a "medium ware" solution that sits between these two paradigms. Built in Python and leveraging the Parquet columnar format, ParquetDB combines the efficiency of column-based data storage with the accessibility of file-based approaches. Columnar storage significantly improves compression and read performance by grouping similar data together, reducing the cost of both serialization and deserialization. Parquet also preserves rich metadata at the table and column levels, enabling advanced features such as schema enforcement and automated indexing without the overhead typically associated with full-fledged database engines. By integrating these capabilities into a lightweight, serverless architecture, ParquetDB mitigates the complexities of evolving data models, which can pose challenges in rigid relational systems or loosely structured NoSQL stores.

Features

Features and Benefits	Description
Simple Interface	Easy-to-use methods for creating, reading, updating, deleting, and transforming data.
High Performance	Utilizes Apache Parquet and PyArrow for efficient data storage and retrieval.
Complex Data Types	Handles nested and complex data types (Ex. ndarrays, lists, dictionaries, python functions and classes etc.).
Portability	File-based storage, allows for easy transfer. (Any framework that can read a directory of parquet files can read ParquetDB databases)
Schema	Contains a schema that describes the data, ensuring consistency.
Schema Evolution	Supports adding new fields and updating schemas.
Predicate Pushdown	Optimizes queries by reading only relevant data blocks.
Column Pushdown	Selects columns to read into memory.
Efficient Encoding	Choice of field-level encoding.
Efficient Compression	Choice of field-level compression.
Metadata Support	Table and field-level metadata support.
Batching Support	Files are grouped to facilitate batching.

Benchmarks

In this section, we show two benchmark experiments used to evaluate the performance of ParquetDB in comparison to SQLite and MongoDB. In these experiments synthetic datasets consisting of 100 integer columns with varying record counts were used to simulate different load levels. Integers were chosen as they are a fundamental data type, allowing us to establish baseline performance metrics with minimal additional computational complexity. Since integers are more lightweight compared to other data types, such as strings or nested structures, they provide an initial estimation of database performance without the variability introduced by more complex data representations.

In the first experiment, we compare the write and read performance of ParquetDB, SQLite, and MongoDB over different load levels as shown in Figure . For smaller datasets, ParquetDB exhibits create times that are comparable to those of SQLite and MongoDB. However, as dataset size grows, ParquetDB demonstrates the second-best performance, following SQLite. In terms of read performance, ParquetDB initially lags behind both SQLite and MongoDB for small datasets but shows considerable improvement as the dataset size increases, ultimately outperforming both competitors beyond a threshold of several hundred to a thousand rows. This improved performance can be largely attributed to the efficiency of Parquet's row-columnar storage format, which becomes increasingly advantageous as dataset size grows.

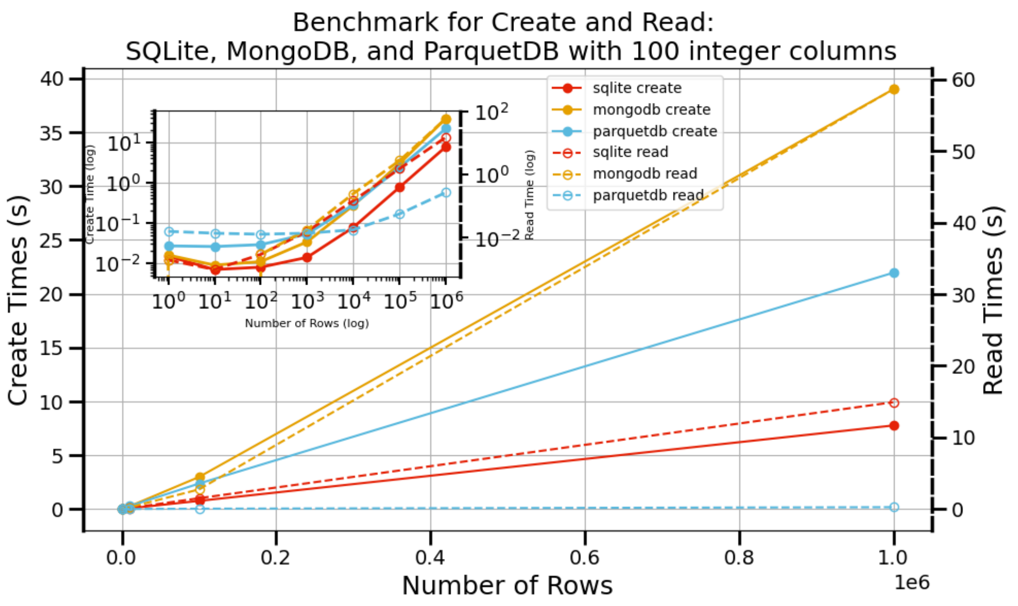

Figure 1: Benchmark Create and Read Times for Different Databases. Create time is plotted on the left y-axis, read time on the right y-axis, and the number of rows on the x-axis. A log plot is shown in the inset.

In the second experiment, we perform a needle-in-a-haystack benchmark to evaluate the performance of ParquetDB, SQLite, and MongoDB to query a particular `id` from the dataset. The results are shown in Figure . For smaller datasets, ParquetDB exhibits significantly worse performance, lagging behind indexed SQLite, non-indexed SQLite, indexed MongoDB, and non-indexed MongoDB by approximately an order of magnitude. The relatively high query times in ParquetDB can be attributed to its lack of explicit indexing, resulting in a more exhaustive data scan compared to databases that leverage efficient indexing structures.

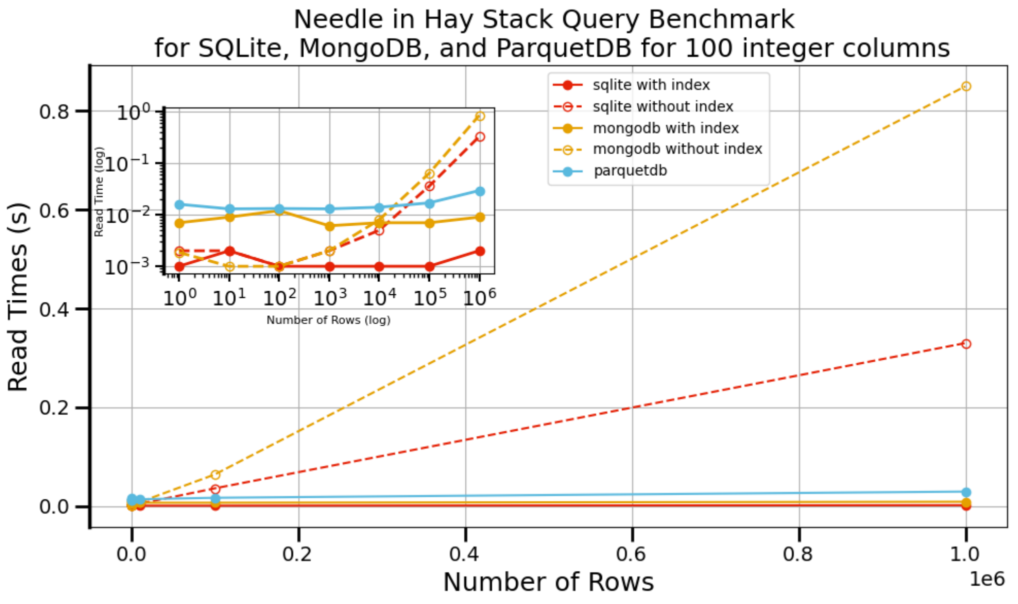

Figure 2: Needle-in-a-Haystack Benchmark Results. Time is on the y-axis, number of rows on the x-axis. The log plot is shown in the inset. SQLite and MongoDB are compared with and without indexing.

As dataset size increases, ParquetDB's performance improves significantly, ultimately becoming the third most efficient system, with a relatively constant query time as the number of rows grows. This trend of constant query time is also observed for indexed SQLite and indexed MongoDB, which can be attributed to the use of B-tree indexing. B-tree indexing is an efficient data structure that allows databases to maintain sorted data and perform searches, insertions, and deletions in logarithmic time, drastically reducing query time for indexed columns as dataset size scales. In contrast, the non-indexed SQLite and non-indexed MongoDB systems exhibit increasingly poor performance as dataset size grows, primarily due to their reliance on full table scans to locate the desired value. The absence of indexes forces these databases to perform linear scans, which results in longer query times with larger datasets.

Interestingly, ParquetDB is able to achieve query performance comparable to the indexed versions of SQLite and MongoDB, despite not utilizing traditional indexing mechanisms. This efficiency can be largely attributed to predicate pushdown filtering. Parquet files store field-level statistics in their schema, which allows for efficient filtering and retrieval of data without requiring a complete scan of the entire dataset. By leveraging these statistics, ParquetDB can effectively narrow down the search space, resulting in query times that are comparable to those of indexed systems.

Installation

For installation, please use pip:

```
pip install parquetdb
```

For more details, including advanced features and contributions, please visit the GitHub repository. The repository contains additional examples, API documentation, and guidelines for contributing to the project.

Acknowledgements

We thank the Pittsburgh Supercomputer Center (Bridges2) and San Diego Supercomputer Center (Expanse) through allocation DMR140031 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296. We gratefully acknowledge the computational resources provided by the WVU Research Computing Dolly Sods HPC cluster, partially funded by NSF OAC-2117575. Additionally, we recognize the support from the West Virginia Higher Education Policy Commission through the Research Challenge Grant Program 2022 (Award RCG 23-007), as well as NASA EPSCoR (Award 80NSSC22M0173), for their contributions to this work. The work of E.R.H. is supported by MCIN/AEI/ 10.13039/501100011033/FEDER, UE through projects PID2022-139776NB-C66. K.C. thanks funding from the CHIPS Metrology Program, part of CHIPS for America, National Institute of Standards and Technology, U.S. Department of Commerce. Certain commercial equipment, instruments, software, or materials are identified in this paper in order to specify the experimental procedure adequately. Such identifications are not intended to imply recommendation or endorsement by NIST, nor are they intended to imply that the materials or equipment identified are necessarily the best available for the purpose.

References

About SQLite. (n.d.). https://www.sqlite.org/about.html.

Customer Case Studies. (n.d.). In *MongoDB*. https://www.mongodb.com/solutions/customer-case-studies.

Habyarimana, E., & Michailidou, S. (2021). Genomics Data. In C. Sodergard, T. Mildorf, E. Habyarimana, A. J. Berre, J. A. Fernandes, & C. Zinke-Wehlmann (Eds.), *Big Data in Bioeconomy: Results from the European DataBio Project* (pp. 69–76). Springer International Publishing. ISBN: 978-3-030-71069-9

Jain, A., Ong, S. P., Hautier, G., Chen, W., Richards, W. D., Dacek, S., Cholia, S., Gunter, D., Skinner, D., Ceder, G., & Persson, K. A. (2013). Commentary: The Materials Project: A materials genome approach to accelerating materials innovation. *APL Materials*, *1*(1), 011002. https://doi.org/10.1063/1.4812323

MongoDB: The Developer Data Platform | MongoDB. (n.d.). https://www.mongodb.com/.

Singh, A. P. (2024). *15 Types of Databases and When to Use Them*. https://blog.algomaster.io/p/15-types-of-databases.

Well-Known Users Of SQLite. (n.d.). https://www.sqlite.org/famous.html.