# Grizzlies: Auto-Indexing for Pandas Dataframes

Kiran Bodipati
bodipati@umich.edu
University of Michigan
Ann Arbor, MI, United States

Shruti Jain
shrujain@umich.edu
University of Michigan
Ann Arbor, MI, United States

Shashank Kambhammettu
skambh@umich.edu
University of Michigan
Ann Arbor, MI, United States

Jai Narayanan
jainara@umich.edu
University of Michigan
Ann Arbor, MI, United States


**Figure 1.** Grizzlies Logo

## Abstract

Pandas is a Python library originally built for exploratory data analysis. Its simple interface has made it popular well beyond this use case, resulting in the library being commonly used for more complex tasks where performance becomes a consideration. Grizzlies is a library built on top of Pandas that implements auto-indexing, similar to many popular DBMS systems, allowing users to experience the same simple Pandas interface but with significantly improved performance. Experiments with relatively simple index create & drop heuristics show that we get highly improved runtime compared to Pandas at the cost of slight memory usage increases.

***Keywords:*** Pandas, Dataframe, Index

## 1 Introduction

Pandas is a Python library primarily built for exploratory data analysis. It provides users with a framework for working with datasets known as DataFrames. Most users working with small or medium sized datasets don't notice many issues, but anyone using Pandas for more than surface-level tasks (like displaying summary statistics or performing quick filters), will notice significant user and performance issues. Firstly, Pandas uses the term "indexing" to refer to slicing, i.e. selecting a subset of rows from the DataFrame or selecting a subset of values from a Series (which is a list of values in the form of a numpy array, usually representing a single column or a single row). Whenever such a slice is returned, it's referred to in the Pandas documentation as an index. This presents much confusion for users because Pandas also uses the term "index" to refer to a data structure which associates a set of values to each row in the dataframe. Whenever a Pandas dataframe is created, users may notice a that it contains a column which is actually the default index: a list of integers starting at 0 and increasing by 1 for each row (essentially a row ID). However, the default index is essentially useless as users rarely need to lookup a specific n-th row. Instead, lookups are typically performed by value in a certain column, which requires a linear scan of the entire dataframe.

Many DBMSs provide the ability to create data structures known as indexes, which are most commonly implemented using hash tables or B-trees, to optimize lookups or range-based queries, but Pandas has no such feature and instead uses the term index to refer to a column which essentially acts as row IDs. This design choice – to rely exclusively on linear search – may be attributed to the fact that dataframes are stored entirely in memory, so the space taken up by an additional data structure would limit the number/size of dataframes that users may want to work with. However, this introduces a significant performance limitation for users performing many range-based queries, as these now require O(n) time to search every tuple rather than the potential O(log(n)) time improvement introduced by b-tree indices or O(1) time improvement introduced by hash maps. Naturally, this problem becomes exponentially worse as the size of the in-memory datasets increase.

In recent years, the number of Pandas users (especially those

*CSE 584, University of Michigan,*
WN 2025.

without as much computer science background) is increasing, along with the number of uses for Pandas as a temporary, in-memory database, introducing many performance limitations for a variety of new users. Therefore, this project aims to address these limitations by implementing some optimizations frequently seen in DBMS systems. Namely: (a) implementing hash based and sorted indices so that users can see increased performance for lookups and range-based queries, and (b) creating an automatic Index Management feature so that Pandas users can have optimized performance without sacrificing too much memory or having to learn the underlying complexities of Pandas indices. Another interesting question to study would be to analyze how the above goals would differ in data science workloads as compared to DBMS workloads in terms of implementation and adaptation challenges. This project also aims to find and adapt representative DBMS workloads, similar to TPC-H benchmarks, to validate the performance increases gained by adding these new features to the Pandas library.

## 2 Related Work

Given the popularity of Python, and the Pandas dataframe structure, many research and industry efforts have attempted to optimize Pandas or create alternative frameworks with improved performance. Most of these works focus on addressing scalability and computational throughput, especially for large datasets and distributed systems.

### 2.1 Bodo

Bodo [2] is a high-performance compute engine for Python that uses a Just-In-Time (JIT) compiler to automatically parallelize Pandas and NumPy workloads. By transforming Python programs into optimized, distributed binaries using MPI (Message Passing Interface), Bodo can achieve performance improvements on the order of 20x to 240x compared to native Pandas. It integrates seamlessly with standard Python APIs and is designed for large-scale workloads in data science and machine learning. Bodo's optimization strategy is holistic, relying on ahead-of-time compilation and static analysis to parallelize operations. However, it does not modify the core semantics of Pandas indexing or optimize for access patterns in the traditional database sense. In contrast, our work addresses performance on a single machine, since this is more relevant to most users, by automatically constructing and managing indices on Pandas columns to speed up common operations like joins and lookups, without requiring distributed execution.

### 2.2 Dask

Dask [4] DataFrame provides a similar interface for users working with Pandas but partitions large DataFrames across a cluster and coordinates parallel execution under the hood. It targets datasets that exceed memory limits or computations that are slow to execute serially. Dask's execution model is similar to Pandas' as well but requires explicit computation triggers (.compute(), .persist()) and can incur overhead in scheduling and data shuffling. Dask is ideal for large-scale ETL, ML preprocessing, and data wrangling, but, similarly to Bodo, it does not address performance bottlenecks in small-to-medium datasets or provide internal indexing optimizations. Our work can also benefit Dask when used in conjunction with Pandas-style syntax.

### 2.3 Pandarallel

Pandarallel [5] is a lightweight library that parallelizes Pandas operations across available CPU cores with minimal code changes. Unlike Dask and Bodo, it does not support out-of-core computation or distributed execution; instead, it provides a drop-in replacement for commonly used Pandas functions (e.g., apply) and internally uses multiprocessing. Although effective for accelerating parallel tasks, Pandarallel does not alter the underlying execution model or data structures of Pandas. In particular, operations like filtering or joins still rely on linear scans of memory and suffer if data is not appropriately indexed. Our work addresses this exact limitation by introducing automatic indexing mechanisms to avoid such scans and provide faster access patterns.

### 2.4 Cylon

Cylon [3] is a distributed memory data processing framework built on Apache Arrow and OpenMPI. It provides high-performance dataframe operators (e.g., joins, filters) and is designed to interoperate with Machine Learning and Deep Learning frameworks. Cylon's strength lies in its integration with distributed computing environments and efficient communication protocols. It achieves speedups through lower-level optimizations and alternative memory layouts (columnar Apache Arrow format). However, like Bodo and Dask, it operates primarily in a distributed setting, making it a nonviable solution for most users, and does not change the way index structures are used for performance. Our project remains in the Pandas ecosystem and enhances performance by learning from traditional database indexing to build access structures automatically.

### 2.5 Polars

Polars [6] is a fast, Rust-based DataFrame library that uses a columnar memory layout and SIMD/vectorized execution. It introduces "lazy evaluation" and expression trees to enable query planning and optimization, providing up to 30x performance improvement over Pandas for some benchmarks. One downside of Polars is that it is not API-compatible with Pandas and requires users to learn a new syntax. While its architectural changes provide high performance, it does not focus on access pattern optimization in the way traditional databases do. In contrast, our approach stays within Pandas

and introduces lightweight, intelligent indexing mechanisms that speed up common operations without requiring users to adopt a new framework or syntax. [6]

### 2.6 Auto-Indexing SQL Databases in Microsoft Azure

Traditional database systems have long recognized the importance of indexing to improve query performance. Microsoft's SQL Server provides two relevant tools: the Missing Index (MI) tool and the Database Tuning Advisor (DTA) [1], which automatically suggest index recommendations based on observed query workloads. These systems track query plans and identify columns frequently used in filters or joins, recommending indices to speed up those operations. Although well-established in the database world, such ideas have not yet been broadly applied to in-memory dataframe systems like Pandas. Our work aims to bring these concepts to Pandas by tracking access patterns and dynamically building and maintaining indices on frequently used columns, bridging a gap between data science tooling and classical database optimization techniques. That being said, the main limitation for this is that Pandas dataframes are ephemeral, making it difficult to collect metrics, develop a representative workload, and track query performance. This limitation means that even though our solution is inspired by these kinds of systems, it will never be as sophisticated as them, but a consequence is that our solution is therefore forced to be simple and lightweight, which works well with the Python/Pandas philosophy and serves to limit consumption of memory and computational resources.

## 3 Implementation Overview

In order to speed up Pandas using indexing we built a wrapper class around Pandas called Grizzlies. The Grizzlies class extends a Pandas dataframe and allows us to intercept queries to different columns and note which columns are accessed to understand the workload in the session. Then, when a certain number of accesses are reached for a given column, we create an index on that column.

In order to keep our solution relatively lightweight, we also need to make sure that the total amount of space taken by our indices is not so high that it slows down performance. To counter this problem, we have a dropping policy to keep the number of indices relatively limited.

### 3.1 Overall Structure

A Grizzlies DataFrame is created in the same way as a Pandas DataFrame. However, since we also wanted to experiment with different hyperparameters for our creation and drop heuristics, we also allow the user to specify different auto-indexing parameters at the initialization of a Grizzlies object.

Specifically, the user can set the `create_scheme`, `windowsize`, `threshold`, `xval`, `drop_scheme`, and `index_type`. We now describe what each of these hyperparameters mean.

### 3.2 create_scheme

The `create_scheme` parameter decides what method we use to track the column accesses. The two options for this are "basic" and "sliding".

The "basic" create scheme uses a dictionary as a counter to simply keep track of the number of accesses to a column. Any access to a column that is not row-based (ex. by doing `df[col]` or `df[df[col] = x]` increments the counter in the basic scheme.

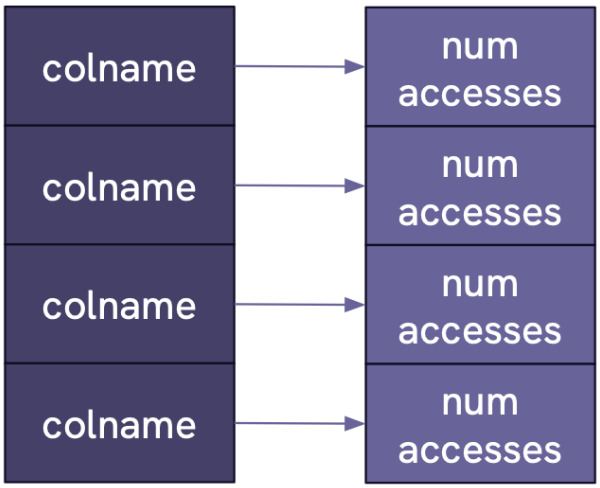


**Figure 3.** Basic Scheme

However, this scheme comes with certain issues. In longer Python programs with changing workloads, this tracking scheme can cause index creation on irrelevant columns. Consider a situation where we want a maximum of only 2 indices. If our creation threshold is 7, and column A is accessed 6 times in the beginning, and then columns B and C are the only ones accessed for all subsequent queries but we have a single column A access, since the overall amount of column A accesses are now 7, (though not recent), Grizzlies might then drop the index on column B or C and create one on column A even though it was only accessed once recently. We have tried to mitigate this issue by resetting the counter every time an index on a column is created so that the statistics do not infinitely increase, but the above example problem still holds.

To remedy this issue, the second create scheme, "sliding", uses a sliding window to keep track of only the last $n$ accesses. It does so by instead keeping a deque with a max length of size $n$.

We intend our solution to be used with the sliding `create_scheme`, but we have kept the basic scheme for testing and because it provides the benefit of being slightly more lightweight.

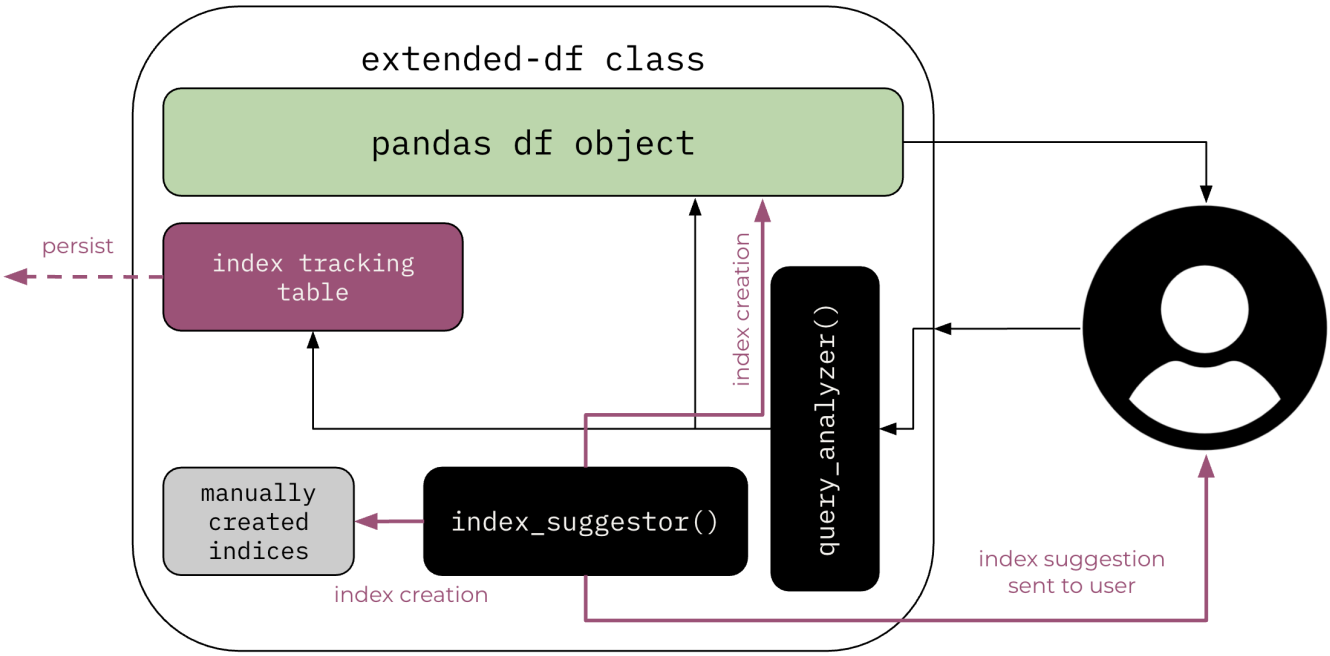


**Figure 2.** Architecture Overview

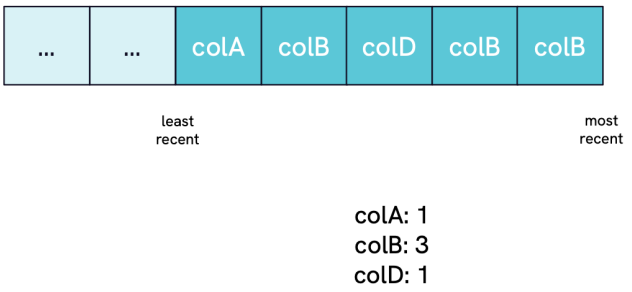


**Figure 4.** Sliding Window Scheme

### 3.3 windowsize

The `windowsize` specifies the value *n*. We set a default windowsize of 16 because that gave us the best results on the datasets we used for the results reported in the paper. In practice, a user should tune this value based on the variability in their workload.

### 3.4 threshold

The `threshold` parameter specifies the creation threshold such that when a column has been accessed more than or equal to the threshold, we create an index on the column. Again, we have provided a default value for this to be 5 so that we do not wait too long before seeing the benefits of an index, but we don't create an index on an infrequently accessed column either.

### 3.5 xval

The `xval` is the frequency with which we check for index creation (and consequently, index dropping). We realized that if we checked whether the threshold was reached on every access, we would slow down all accesses, so we instead opted to have a default `xval = 10` so that only on every 10th access would be a check to see if we wanted to create an index on a specific column. A user can technically set `xval = 1` to check every access for creating/dropping indices, but we recommend setting `xval` greater than or equal to the threshold as we cannot create an index till the threshold is reached.

Note that the access counting for a column is done on every access independently of the `xval`.

### 3.6 drop_scheme

Since Pandas is entirely in memory, we have a limit on how many indices we want to create in order to conserve memory. We set the `_max_indices` to be `windowsize/threshold` (as it logically follows that that is the max number of columns that can be meeting a threshold). The user can specify the `drop_scheme` to be `lru`, `min`, or `threshold`.

In the LRU (`lru`) scheme, a separate dictionary tracks when a column was last accessed. The overall LRU counter is incremented whenever an access is made to the index on a column. When dropping an index, the column that was least recently accessed is dropped (this is the column with the lowest LRU). The LRU is tracked only for columns that

currently have an index since we do not use LRU for creating indexes.

In the minimum (`min`) scheme, a separate dictionary tracks how many times a column was last accessed. When dropping an index, the column that was least accessed is dropped. Again, we only track this for columns that currently have an index.

In the threshold (`threshold`) scheme, all columns which do not have enough accesses in the sliding window to meet the threshold are dropped. This scheme only makes sense to use with the `sliding` creation scheme and not with the `basic` scheme because the value of the `_max_indices` is set to be `windowsize/threshold`, so there can only be at most `_max_indices` number of columns that meet the threshold.

We have found the `lru` scheme to make the most logical sense, so we have set our default drop type to be `lru`. There is not much performance difference between the different drop schemes.

### 3.7 index_type

The `index_type` can be either `hash` or `sorted`. The default is `hash`, which is designed to be O(1) for equality based queries as it uses a dictionary. The `sorted` one uses a `SortedDict`, which is O(log(n)), but designed to be better for range based queries. In practice both are relatively fast for equality queries and not very fast for range based queries.

### 3.8 Using The Created Index

Although our original aim was to overload queries like `df[df[col] < x]`, the way that Pandas evaluates these queries is by computing a boolean mask in the form of a series object. Trying to intercept this to use our index proved to be complicated, and we recognized that the goal of our project was not to create an identical, optimized version of Pandas but rather to experiment what performance benefits we could ideally have. Therefore, we instead created an `evalfunc()` that takes in a column, an operator ($<$, $\leq$, $>$, $\geq$, ), and a value. It then checks if there is a hash or sorted index on the accessed column and uses it for faster results. We use this function for testing purposes.

### 3.9 Incorporating All Different Hyperparameters

Since we have many different hyperparameters, we do not want to check for each in an `if` statement every time we have a function call. To address this issue, we set our functions for incrementing the access count, dropping and creating indices to be the appropriate version of the function in the constructor itself. This allows us to reduce the number of checks and have the lowest overhead possible while offering flexibility.

### 3.10 Persisting Tracking Across Sessions

We noted that since Pandas runs in memory during a single Python session, in instances where the same Python file is run multiple times, all data in Pandas is lost. However, this means that there is major predictability in terms of which columns are accessed in each session. To take advantage of this, we provide a `save()` function that allows users to save their existing session stats (not indices, just access counts or sliding window) in the form of a pkl file. This pkl file is named based on the columns, shape, and create scheme and automatically checked for when a new Grizzlies object is initialized to see if we can reload existing stats. We acknowledge that these stats may not be entirely accurate across sessions, but we hope that the sliding window scheme mitigates any negative impact caused by inaccurate stats caused by changing queries across sessions.

### 3.11 Updating The Index

When an update occurs, to ensure correctness, we drop and then rebuild the index. This causes our performance to be slow in workloads with many updates. We also considered simply dropping the index instead of rebuilding one, but in general we believe that our solution is better suited for workloads that are read heavy.

### 3.12 Using Middle Layer Over Pandas Indexing

One of the solutions we attempted was to use Pandas' native `set_index()` function instead of implementing our own hash-based and tree-based indexes. This was motivated by our review of Pandas' documentation, which states that the `set_index()` method uses a hash-based algorithm. With initial testing, where we tested access times with and without a Pandas index, we observed some speedup, so we implemented a version of our library which intercepts access calls and creates/drops indexes the same way our manual implementation does, but with the native Pandas index instead. While implementing `merge()`, `groupby()`, and `sort()`, however, we did not observe the kind of speedups we expected. Further reading revealed that the Pandas `set_index()` function does not actually create a hash-based index structure, as the term index is typically used in a DBMS context, but in fact just creates a mapping between the column values and the tuple, essentially allowing users to replace the default row ID (an auto-incrementing integer column), with some other column. This was a disappointing revelation for the team since we had dedicated a lot of effort (now wasted) to building a solution with support for native indexing, only to find that Pandas was using the term index unconventionally (closer to the definition of indexing into an array than creating an index structure for a data table) with no explanation.

Even though we did notice some performance increase with this method, we don't have a reasonable explanation for why this occurred. Additionally, it doesn't logically make sense to set a column as the de-facto row ID just for performance increases if it doesn't make sense semantically. More importantly, this affects user interface because it changes the way users interact with the DataFrame (using `loc`, `iloc`, `[]`,

etc.), so having changing indices and `MultiIndexes` would cause more compilation errors and frustrations for a user, costing significantly more time in development and debugging than is saved in runtime.

## 4 Experimental Setup

We conducted several experiments on real-world and benchmark datasets to evaluate the performance of our proposed auto-indexing methods, both in terms of effectiveness and scalability, in comparison to the Pandas library. For the results mentioned in the paper, the experiments were run on a machine with an Intel Core i7-1355U 1.70 GHz CPU, with 16GB RAM and a Windows 11 24H2 operating system.

The implementation and experiments were run on Python 3.12.4. All baselines were compared against Pandas 2.2.3, and the Grizzlies object extended the same version of Pandas. The memory consumption of each function call was tracked using the `memory_profiler` package and the execution time was clocked using the built-in `time` package. To reduce system variability and errors, each experiment was conducted 10 times and the results reported are the average performance across these multiple runs, unless otherwise mentioned.

Some of the datasets we used to benchmark our results include:

- **TPC-H Benchmark** - We used the TPC-H [7] benchmark at scale factor 1. This benchmark consists of standard analytical workloads, including filtering, joins, aggregates, etc.
- **Kaggle Datasets** - We selected several open-source Kaggle data sets, including Movie Reviews, Yelp dataset, Financial datasets etc., to reflect diverse and representative data characteristics and distributions. As such, we ensured that the datasets we selected covered different datatypes, such as numerical, categorical, string etc. We also ensured different levels of sparsity and uniqueness, especially in the categorical data types.

To collect column access statistics, we run the TPC-H queries several times and allow the system to collect stats across multiple runs, so that the index creation algorithm can create the appropriate set of indices, before we measure the performance. For Kaggle datasets, we synthetically simulate column accesses to collect the required statistics for index creation, before measuring performance.

For testing performance at scale, we scaled our datasets by applying a scaling factor. We examined the effect of independently scaling rows while keeping the columns fixed, and scaling columns while keeping the rows fixed, as well as the combined effects of scaling columns and rows together. Due to hardware constraints, we are only able to report performance at scales upto 16GB, but we provide analytical trends based on the results at experimentally verified scales.

The correctness of results are verified between Pandas and our implementation by several tests including `pytests` and Python assert functions.

## 5 Results

### 5.1 Performance Comparison with Pandas Baseline

We begin by comparing the raw performance of our auto-indexing methods against baseline Pandas. We specifically measure the row filtering tasks, since creation of indices most benefits these lookups. Figure 5 shows the comparison of time taken for row accesses between the baseline Pandas and the two Grizzlies index creation strategies. We observe that both the index creation strategies outperform the baseline Pandas. Specifically, we see that our methods are several orders of magnitude better than Pandas.

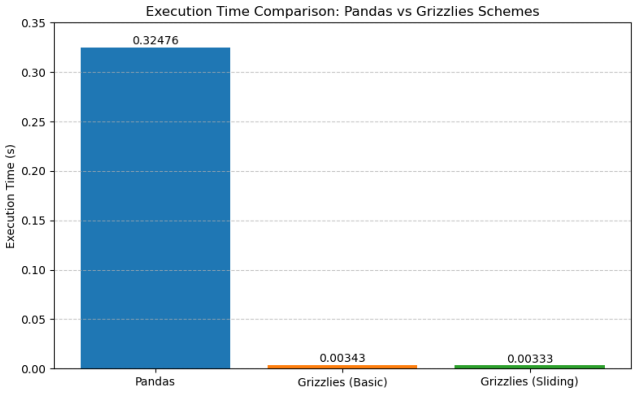


**Figure 5.** Pandas vs Grizzlies: Execution Time

Table 1 compares the performance of the various index creation, index drop and index type strategies, both in terms of time and memory usage. We notice that on average, the our indexing techniques offers an **80x - 100x** improvement over Pandas for the row lookup task. The specific configurations not have such a significant impact on performance, and we observed that these parameters are often workload dependent, and would have to be selected by tuning the parameters. Other hyperparameters such as `xval`, `threshold` and `windowsize` are also workload dependent, and need to be tuned for different workloads. The average memory usage is also relatively consistent across all parameter and hyperparameter settings.

### 5.2 Scalability Analysis

We performed extensive tests on scalability by taking datasets and applying a scaling factor to both rows and columns. For the results reported, we used a Yelp 1M row 15 column dataset, and we scaled the rows by factors 2, 4, 8 and 16. We similarly scaled the columns by factors 1, 2, 3 and 4. We measured the execution times and memory usage of the functions for increasing number of rows and columns.

**Table 1.** Grizzlies Performance and Memory Usage Across Indexing Schemes

| create_scheme | drop_scheme | index_type | time (s) | memory (MB) |
|---|---|---|---|---|
| Pandas | | | 0.311635 | 2553.35 |
| sliding | lru | ordered | 0.003855 | 2401.74 |
| sliding | lru | hash | 0.003326 | 3007.83 |
| sliding | min | ordered | 0.003518 | 2402.59 |
| sliding | min | hash | 0.003280 | 3624.08 |
| sliding | treshhold | ordered | 0.003955 | 3172.35 |
| sliding | treshhold | hash | 0.003718 | 2392.77 |
| basic | lru | ordered | 0.003610 | 2402.24 |
| basic | lru | hash | 0.003684 | 2547.94 |
| basic | min | ordered | 0.003434 | 2401.96 |
| basic | min | hash | 0.003481 | 2547.25 |

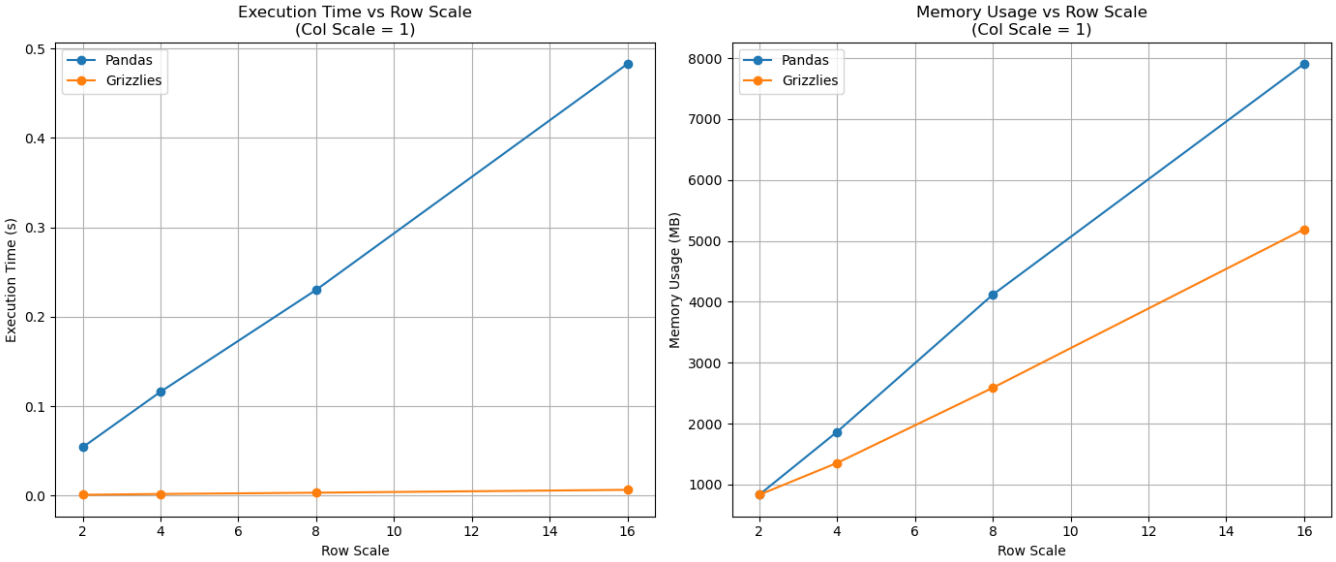


**Figure 6.** Pandas vs Grizzlies: Scaling of Rows

**Table 2.** Performance and Memory Usage of Pandas vs. Grizzlies Across Varying Row Scales

| Data Size (MB) | Row Scale Factor | Pandas Time (s) | Pandas Memory (MB) | Grizzlies time (s) | Grizzlies Memory (MB) |
|---|---|---|---|---|---|
| 984.54 | 2 | 0.053991 | 831.28 | 0.000851 | 830.27 |
| 1969.09 | 4 | 0.116027 | 1863.44 | 0.001747 | 1353.02 |
| 3938.18 | 8 | 0.230032 | 4115.27 | 0.003195 | 2586.85 |
| 7876.35 | 16 | 0.482795 | 7905.70 | 0.006409 | 5189.55 |

Figure 6 shows lineplots comparing the execution and memory usage of Pandas and Grizzlies when the rows are scaled. We see that while both Pandas and Grizzlies grow linearly, Pandas grows at a significantly higher rate as compared to Pandas. We notice that Grizzlies is roughly 50x-100x better than Pandas at around 2 million rows, but when we work with 16 million rows(8GB) of data, Grizzlies is roughly 1000x better than Pandas. We also observe an improving trend of memory usage for Grizzlies over Pandas when the rows are scaled. As such we conclude that our proposed solution is extremely scalable with respect to the number of rows. The detailed metrics are found in Table 2

However, we cannot say the same about the scalability of columns. We observe that when the number of columns

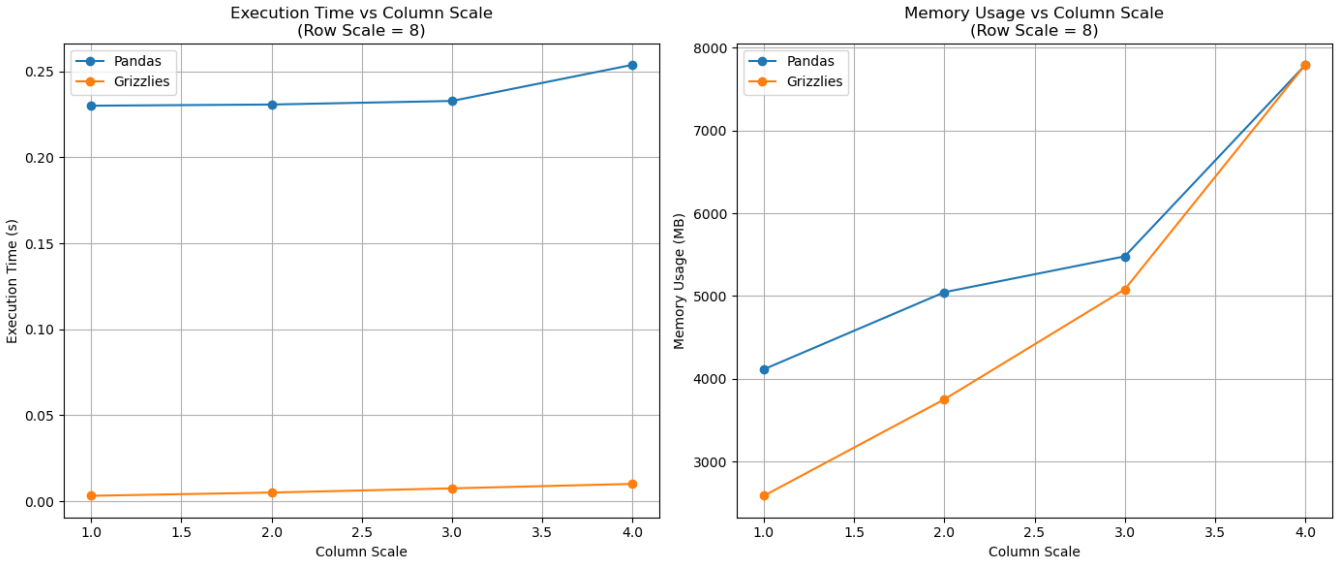


**Figure 7.** Pandas vs Grizzlies: Scaling of Columns

**Table 3.** Performance and Memory Comparison between Pandas and Grizzlies across varying column scales

| Data Size (MB) | Column Scale Factor | Pandas Time (s) | Pandas Memory (MB) | Grizzlies time (s) | Grizzlies Memory (MB) |
|---|---|---|---|---|---|
| 3938.18 | 1 | 0.230032 | 4115.27 | 0.003195 | 2586.85 |
| 7876.35 | 2 | 0.230718 | 5046.43 | 0.005085 | 3752.26 |
| 11814.53 | 3 | 0.232797 | 5479.66 | 0.007529 | 5080.67 |
| 15752.71 | 4 | 0.253802 | 7790.33 | 0.010108 | 7793.64 |

are increased, the time and memory consumptions scale at a complexity greater than the linear rate, as indicated in Table 3. However, from the table and Figure 7 we do note that while scaling of columns is poor, we still observe that Grizzlies is still several orders of magnitude better than Pandas at data sizes of approximately 16GB. For all practical intents and purposes, we still observe that Grizzlies holds the edge, in terms of scalability. Further discussions on the practical implications of scalability can be found in a later section.

**Table 4.** Performance across Data Types: Execution time

| Datatype | Pandas (s) | Grizzlies (s) |
|---|---|---|
| Long Strings, infrequent | 0.3174 | **0.0037** |
| Strings, categorical | 0.3381 | **0.0177** |
| Integer, unique | 0.0088 | **0.0004** |
| Integer, categorical | **0.1745** | 0.2124 |
| Float, Short Range | 0.0970 | **0.0068** |
| Float, Long Range | 0.0079 | **0.0004** |

### 5.3 Indexing performance across data types

We evaluated the performance of the index across several datasets with several data types. We include numerically heavy data (eg. stock prices), categorical heavy data (Eg. States), high cardinality data (eg. Star Rating), low cardinality data (eg. reviews, phone numbers) etc.

We summarize this data in Table 4. The index based filtering methods in Grizzlies heavily improve row lookup tasks in numerical and low cardinality data, where there are lots of unique values. Pandas, on the other hand uses linear probing, which is slower than or methods. However, we observe a slight degradation in performance when we have to filter datasets that have categorical columns and high cardinality data, where the linear probing methods in Pandas is better than the index based methods in Grizzlies.

### 5.4 TPC-H Query Performance

We evaluated TPC-H queries Q1-Q6 against Pandas and Grizzlies. These queries were chosen for their diverse tasks, including filtering, sorting, joins, aggregates etc.

As shown in Table 5, we notice performance improvements in queries Q2, Q3, Q4 because they primarily focus on filtering tasks. Given that our improvements primarily focus on row accesses, the improvement for these tasks is as expected. In contrast we observe a slight degradation in performance in Q1 and Q5. These queries have aggressive

multi table joins, and we notice significant implementation overheads leading to degradation of performance.

**Table 5.** Performance Comparison: Pandas vs Grizzlies on TPC-H Queries 1–7

| TPC-H Query | Pandas (s) | Grizzlies (s) |
|---|---|---|
| Query 1 | **1.0580** | 1.0754 |
| Query 2 | 0.0963 | **0.0904** |
| Query 3 | 0.2827 | **0.2715** |
| Query 4 | 0.4907 | **0.4076** |
| Query 5 | **0.5148** | 0.5629 |
| Query 6 | 0.0764 | **0.0763** |

A commonly asked question is the observed performance improvements in TPC-H as compared to our other benchmarking tests. This is attributed to the workload characteristics and the query designs. In the case of our custom benchmarking, we design functions to specifically measure only the row filtering tasks. However, the TPC-H queries do not isolate specific tasks but rather contain a combination of tasks. Given that our implementation currently focuses on improvements to row accesses, the other functions, such as sorting, groupbys, etc. will have performance similar to Pandas. Hence, we observe just minor improvements in the overall query.

# 6 Discussion

## 6.1 Implications in Data Science

Some of the most common applications for Pandas in data science include exploratory data analysis, preparing data for creating plots and writing dataloaders, and creating training and validation splits to prepare for downstream machine learning tasks. We believe that in all these cases, there are extensive row lookups, and our auto-indexing strategies significantly improve performance in these tasks.

As such, Grizzlies in extremely scalable with increasing number of rows. One key point of discussion is the increasing number of columns, and its implications, especially since several machine learning tasks generally have multiple features as columns in a dataframe. While the scaling of columns is poor, we still note that at 16GB of scale, it is several orders of magnitude better than pandas. Additionally, most machine learning tasks involve batch processing, and will take smaller number of rows per batch as input, thus mitigating the performance degradation caused by scaling of columns. We believe that in all practical cases, we are able to achieve an improvement over Pandas for row lookups.

## 6.2 Next Steps

**6.2.1 Memory Usage.** Our auto indexing implementation currently does not directly account for the memory usage of individual indices or the memory pressure on the running program. While our LRU and threshold based dropping schemes should mitigate excessive memory consumption by limiting the number of indices, they drop based on heuristics rather than based on actual memory statistics. This means that under workloads on large datasets where a single index could take up a substantial portion of memory, memory usage can still spike unexpectedly. An improvement would be integrating our index creation process with some sort of memory profiler, that would allow us to monitor memory usage and create the index based on those statistics.

**6.2.2 Column Cardinality.** Currently, we do not account for the cardinality of a column's data when deciding whether to create an index. Columns with high-cardinality (not many unique values), can lead to indices that are not very useful, particularly for equality lookups. An improvement would be to evaluate column cardinality on the fly through either sampling data or keeping track of the number of unique values, and factoring that into the decision to create an index, and index type selection.

**6.2.3 Auto Selecting Index Type.** With our current implementation, users must manually specify the type of index they want for all columns when initializing their Grizzlies dataframe through the index_type parameter, choosing either hash or sorted. However, certain access and query patterns benefit more from one type of index over the other. For example, range based queries would be better served by sorted indices, while equality checks would perform better with hash indices. An improvement would be to automatically infer the optimal index type by keeping track of the query patterns of the user on specific columns. This information would allow us to better adapt the index type at creation, rather than requiring users to opt for a single index type at initialization.

**6.2.4 Post Index Query Performance Monitoring.** Currently, Grizzlies makes an assumption that creating an index will guarantee performance improvement. However, we do not measure the impact of an index after it is created. This can lead to situations where an index is maintained for a column that does not actually benefit from it. Incorporating post-index creation performance monitoring could allow us to evaluate whether an index is truly effective. If the index does not yield performance improvements, it could be dropped so that space could be freed for other potential indices.

**6.2.5 Integrating Query Overloading with Native Syntax.** Our current design uses a custom `evalfunc` to leverage indices for faster query performance. Our original intention was to overlaod native Pandas-style queries, such as `df[df[col] == x]`. Due to how Pandas evaluates queries with this syntax, it was difficult to overload the operator to intercept and use our indices before query evaluation. As

a result, we opted for a separate evaluation function. An improvement would be to properly integrate our evaluation approach with Pandas syntax, such that our indices are still used, while preserving native Pandas syntax.

## 7 Conclusion

In our paper, we presented Grizzlies, an auto-indexing layer built on top of Pandas to improve data access performance. Our approach tracks column access patterns and uses heuristics to control index creation and eviction, without requiring input from the user. Our results show that Grizzlies performs several orders of magnitudes, on average 80x-100x, better than Pandas at row filtering.

## References

[1] Das, Sudipto, et al. "Automatically indexing millions of databases in microsoft azure sql database." Proceedings of the 2019 International Conference on Management of Data. 2019.
[2] Bodo. https://github.com/bodo-ai/Bodo
[3] Cylon. https://cylondata.org/
[4] Dask. https://docs.dask.org/en/stable/dataframe.html
[5] Pandarallel. https://github.com/nalepae/pandarallel
[6] Polars. https://pola.rs/
[7] TPC-H Benchmark. https://www.tpc.org/tpch/