CS594 Provenance & Explanations Project Report

Amy Byrnes

I. Introduction

The concept of provenance-based data skipping is introduced in the paper 2021 paper "Provenance-based data skipping". The technique may be used to speed up certain classes of queries, such as top-k queries. Pandas is a Python library that offers powerful and flexible data analysis tools. However, Python can be slow, and is not optimized for dealing with large amounts of data in the way that many database management systems are. It may be possible to improve the performance of operations in Pandas by applying techniques from databases, such as provenance-based data skipping.

II. Provenance-based Data Skipping

Provenance-based data skipping is a technique to that aims to improve query computation speed by using provenance to filter out tuples that are not necessary for producing the query result. The technique relies on the idea that the provenance of a query forms a sufficient set for answering that query; if query Q is performed on database D and produces result r, then running query Q on the provenance of result r should also produce r.

There are many elements to be considered to implement a propre provenance-based data skipping system. Provenance has to be collected and stored. Some mechanism must be in place to decide if it is appropriate to use that stored provenance to speed up a new query.

III. Pandas

A. Why Pandas?

Pandas is an extremely popular Python library for performing data analysis. It is built on top of the NumPy library to perform fast computations and has built in support for the visualization library matplotlib. Many Python libraries have in turn been built on top of Pandas, such as the visualization library seaborn.

Python is increasingly the preferred language of programmers of all kinds, from computer scientists to commercial developers to domain scientists. There is strong motivation to remain in the Python ecosystem when working with datasets. Python has tools to create and visualize data, and so using Python data analysis tools allows for performing an entire experimental pipeline using one preferred language.

B. Pandas is not a database system

There is certainly overlap between Pandas and dedicated database systems like Postgres. However, there are some crucial differences that call into question the viability of implementing techniques from databases for Pandas. The most critical of these differences is the ability of most database systems to manage data both in memory and on disk. A Postgres database is not just its data; there are a number of metadata structures that Postgres uses to perform queries which are maintained separately from the data.

Pandas, on the other hand, is an object-oriented library. Data and metadata exist together within Pandas objects, and so are only available as long as that object exists in memory. Pandas does have utilities for reading from and writing to files, including dumping and loading the binary for Pandas objects, but this must be managed entirely manually. In short, Pandas provides little, if any, native support for working with datasets that do not fit into memory. The only method presented by the Pandas user guide is to read and process data in chunks, which again must be done manually and is not suitable for all data operations. If this solution does not work for a given use case, users are encouraged to look to other libraries or tool; managing very large datasets is just not in the scope of Pandas.

This does not mean that database techniques absolutely cannot be used to enhance Pandas performance. It may be the case that certain queries still take a (relatively) long time even for (relatively) small datasets - Pandas is built on Python, after all, and while Pandas offers the ability to apply "custom" functions to Series and DataFrames, it is not recommended, as straying away from the core NumPy functions Pandas is built on leads to drastically worsened performance. Additionally, there are libraries built on Pandas that *are* built for working with large datasets; Dask, for example. More discussion on Dask in [[[[REPLACE]]]].

IV. Annotating DataFrames

Provenance-based data skipping using annotations both for generating provenance and filtering using provenance sketches. As described in [1], annotations are based on splitting the domain of an DataFrame column into intervals, and labeling each row based on which interval the attribute is in for that row. This is accomplished using the Pandas method 'pd.qcut', which takes a Series and splits it into a given number of quantiles. This produces an even division of the values in the Series. For example, calling 'pd.qcut' on a Series with ten items and passing the argument 'q=2' will split the Series into two intervals such that each interval contains five items from the Series.

This even partitioning of the Series is returned in the form of a Series with the data type 'pd.Categorical'. This is the

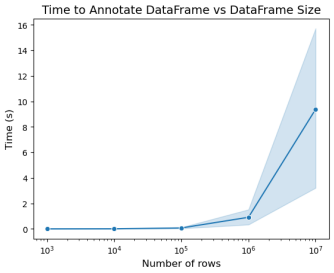

Fig. 1. Time to annotate with ‘pd.qcut‘, averaged over number of bins.

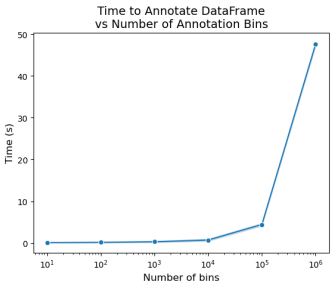

Fig. 2. Time to annotate with ‘pd.qcut‘, averaged over number of rows.

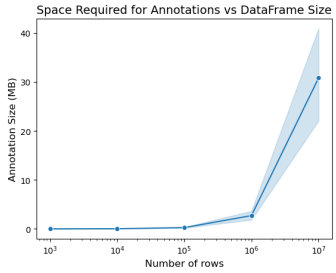

Fig. 3. Additional space taken by annotation data, averaged over number of bins.

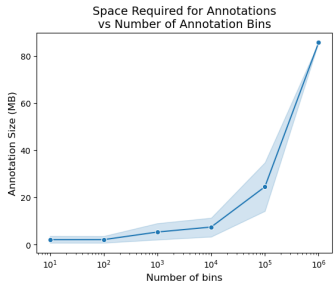

Fig. 4. Additonal space taken by annotation data, averaged over number of rows.

Pandas datatype for categorical data. ‘pd.Categorical‘ data has categories, which can be of any data type, and integer codes corresponding to those categories. An ordering can also be imposed on the categories. ‘pd.qcut‘ creates a Series of categorical data where the categories are ‘pd.Interval‘ objects, and the actual values of the Series are integers corresponding to those intervals. Returning to the example of a Series with ten items, the categories would be two intervals, and the codes would be 0 and 1.

‘pd.qcut‘ cannot be used directly on non-numeric data; however, non-numeric data can be transformed into numeric data by turning it into categorical data. Then ‘pd.qcut‘ can be applied to the codes of the categorical data to split it into even intervals.

A. Annotation Time

For the case of filtering, annotation can be written off as a one-time cost. If many queries are to be performed that are filtered with a sketch on the same attribute, then that attribute just needs to be annotated a single time before the queries can be filtered. However, as described in [1], annotation is also a part of provenance capture. In that case, annotation is part of the query for which provenance is captured, and so we should care about how long annotations take in that case.

1 and 2 show that the time required to annotate a column with ‘pd.qcut‘ is exponential with respect to the size of the DataFrame (and so the size of the column) and the number of bins that the column is split into. However, this increase is far more significant with respect to the number of bins; annotating a 10,000,000 row DataFrame takes on average 10 seconds. On the other hand, annotating a DataFrame with 100,000 bins takes up to a minute depending on the number of rows. It is desirable to have a large number of bins; granular annotations mean that each integer included in a sketch selects fewer rows. However, there is obviously a limit to which this granularity is helpful. The last ”reasonable” time for annotation appears be when there are 10,000 bins - corroborating the conclusion about optimal partitioning from [1]

B. Annotation Memory Consumption

We should also consider the size taken up by annotations. The representation of the annotations as a categorical variable could save space if there are many repeated values; however, as previously stated, granularity is desirable, and as the number of bins increases, the number of repeated values decreases, increasing the size of the categorical data.

?? and 4 show how the size of the annotation column increases exponentially with the size of the DataFrame and the number of bins. Once again, it is the number of bins that has the more dramatic affect; a 100,000 bin annotation requires 90 MB of space, while an annotation column on a 10,000,000 row DataFrame may only take about 35 MB.

V. Filtering DataFrames

It is important to use efficient techniques for filtering datasets, as the purpose of provenance-based data skipping is to speed up queries. This can only happen if the time to filter the data plus the time to perform the query on the filtered data is less than the time to perform the query on the unfiltered data.

As implementing provenance-tracking is outside of scope of this project, dummy values are used for provenance sketches. In this context a provenance sketch is a list of integers corresponding to the integer annotations applied to each row

of the DataFrame. It is assumed that however provenance sketches are generated and stored, they can ultimately be decomposed into such an integer list.

A. Boolean Filtering

Pandas inherits the concept of boolean masking from NumPy. A boolean array can be used as an indexer to a Series or DataFrame (given that it has the same dimensions), and the result will be all the elements in the Series or DataFrame where that array is true. This is the basis of almost all ”filtering” in Pandas.

From this point, filtering strategies were approached from two main angles. First, the operation to produce the Boolean mask. Second, how the Boolean mask should be applied.

In the first case, the techniques used are as follows:

1) Equality comparison between each item in the sketch and the annotation. If the annotation is equal to any of the integers in the sketch, then that row should be included in the query computation.
2) Membership of the annotation in the sketch set. If the annotation is one of the values in the set of integers represented by the sketch, then that row should be included in the query computation.

In the second case, the techniques used are as follows:

1) Using the ‘[]‘ accessor to apply the mask to the DataFrame.
2) Using the ‘.loc[]‘ accessor to apply the mask to the DataFrame.
3) Using NumPy functions to apply the mask to the NumPy array underlying the DataFrame.

B. Equality Comparison

5, 6, and 7 summarize the time taken to filter a DataFrame as the DataFrame size, number of annotation bins, and sketch size increase.

Unsurprisingly, filtering time is exponential in DataFrame size and linear in sketch size. It is also clear that while there is some variation between the ‘[]‘ and ‘.loc[]‘ accessors to the DataFrame, the difference between them is negligible.

Interestingly, using NumPy functions on the underlying values appears to be consistently faster, and the diffrerence between the other two methods and using NumPy appears to increase as the DataFrame gets larger. This is promising if the technique can be expanded to larger datasets.

A final observation is that there is obviously an ideal number of annotation bins. Both too few and too many bins causes the filter time to increase. This gives further evidence that the optimal number of bins is somewhere between 1000 and 10,000

C. Membership Check

The results when the Boolean mask is generated by checking membership in a set are roughly the same. The major difference is that filter time becomes logarithmic with respect to sketch size rather than linear; this is expected behavior for checking whether a value is a member of a set. The results

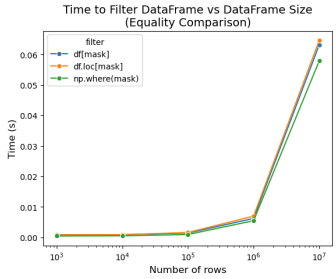

Fig. 5. Time to filter DataFrame as DataFrame size increases, averaged over number of bins and sketch size.

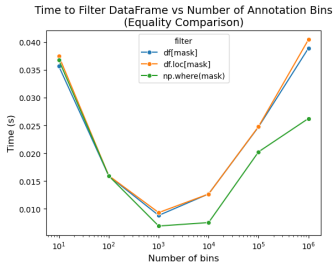

Fig. 6. Time to filter DataFrame as number of bins increases, averaged over number of rows and sketch size.

indicate that equality comparison should be used for small or singleton sketches, and membership checking should be used for sketches containing multiple fragments.

D. Equality vs Membership

The best performing method of applying a Boolean mask to filter a DataFrame for both equality comparisons and membership checking is to use NumPy functions on the underlying NumPy array of the DataFrame. Comparing these methods (9, 10, 11) reveals that equality comparison outdoes membership checking in every case tested here. Membership checking seems to be trending to surpass equality comparisons when the sketch size gets large - however, we should consider that once we are at the point of a sketch containing 10+ fragments we may be losing the benefit of filtering entirely, as more fragments means more rows in the final computation.

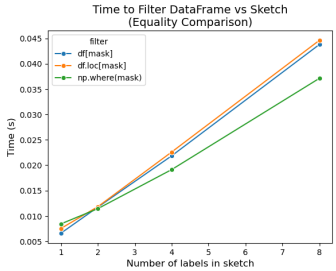

Fig. 7. Time to filter DataFrame as sketch size increases, averaged over number of rows and number of bins.

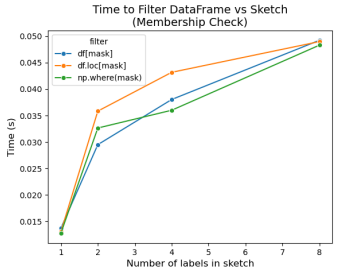

Fig. 8. Time to filter DataFrame as sketch size increases, averaged over number of rows and number of bins.

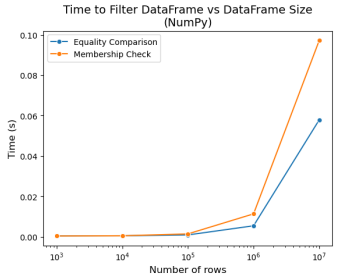

Fig. 9. Time to filter DataFrame as number of rows increases, averaged over number of bins and sketch size.

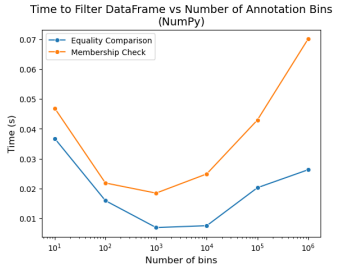

Fig. 10. Time to filter DataFrame as number of bins increases, averaged over number of rows and sketch size.

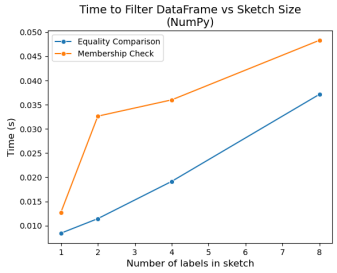

Fig. 11. Time to filter DataFrame as sketch size increases, averaged over number of rows and number of bins.

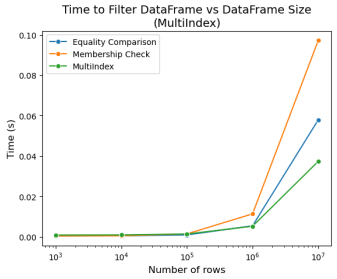

Fig. 12. Time to filter DataFrame as number of rows increases, averaged over number of bins and sketch size.

E. The MultiIndex

A Pandas Series or DataFrame has a special Index column, which is used to access elements of the Series or rows of the DataFrame. There are many flavors of Index in Pandas; one is the MultiIndex. One important utility of a MultiIndex is that it allows for multiple DataFrame rows to be mapped to the same index value. Sections of a DataFrame with a MultiIndex can be quickly accessed using the 'DataFrame.xs' or DataFrame cross-section function.

F. Results

As shown by 12, reindexing the DataFrame on the annotations with a MultiIndex produces some intriguing results. Concatenating all the cross-sections of the MultiIndexed DataFrame turns out to be faster than applying a Boolean mask for any method. However, this isn't a magic solution. Time to filter still appears to be increasing exponentially with DataFrame size, albeit much slower than with the other methods. More problematic is that the reindexing introduces *another* step that takes potentially quite a lot of time. This isn't so bad if the DataFrame can simply be reindexing once and then queries can take advantage of the MultiIndex many times; the cost of reindexing is then amortized. But reindexing also produces a *copy* of the original DataFrame - one that is *larger* than the original. If one is trying to save time because one is working with a large dataset, this can be a very problematic solution - it more than doubles the size of data in memory.

G. Conclusion

Fast filtering is possible using Pandas - or, more exactly, using NumPy. It remains to be seen whether there can be true speedups to Pandas operations using provenance-based methods. The operations that Pandas offers out of the box are quite optimized, and Pandas does not seem to be equipped to handle the truly large datasets that would most benefit from these methods. Dask is a promising way towards working with large datasets using the tools offered by Pandas. Dask is a library built for performing distributed operations in Python, and it includes a wrappers for Pandas objects so that they can be loaded and manipulated in pieces. Unfortunately, it is still quite easy to overload system memory when performing computations on DataFrames loaded with Dask and was therefore not used in this work.

Of course there is a great deal more work to be done to actually implement true provenance-based data skipping in Pandas! Provenance-capture is the most obvious missing piece, and then a mechanism for deciding when and how to use the captured provenance, as well as for determining sketch safety. This is really a very, very small piece of the puzzle.

VI. Personal Note

Because it's a project, not a conference paper!

I have such mixed feelings about this project. I feel like I've done a ridiculous amount of work just to demonstrate that there are some really tiny differences in timing if you filter a DataFrame one way vs the other. On a personal level I don't really mind that, because I feel like I've gotten a lot out of the experience. I certainly have a much better appreciation for Pandas, and all the tiny little details that go into implementing, well, anything where performance is a concern. But it also does not feel good to be submitting things that are insubstantial. I can't shake the feeling that if I had just been better at organizing my thoughts two months ago I could've gotten a lot further with this. On the other hand, I think struggling through this project and this semester has made me better at it for next time. So that's something.

References

[1] Xing Niu, Ziyu Liu, Pengyuan Li, and Boris Glavic. Provenance-based data skipping (techreport), 2021.