Some Reminders for a Seamless Online Class…

- Please turn on your video
- Mute yourself (press and hold spacebar when you'd like to talk)
- Don't do anything you wouldn't do in an in-person class
- I will occasionally check the chat for messages if you'd like to share there instead
- Please say your name before you speak

Logistics

- New homework is up.
- New schedule is up.
  - Thank you for participating in the poll and for your feedback!
  - I'm glad to see that every topic is interesting to at least some people
  - I'm sorry that we'll be covering (or we have covered) some topics that some of you already know
    - e.g., spreadsheets may not be useful to people who know advanced spreadsheet features - but that's not everyone
  - I was also going a bit slower to account for the fact that many of you aren't in an ideal learning environment, but I have heard your feedback and will make sure to cover all the topics
  - Cleaning and integration has been dropped.
- We will now spill over into RRR week (since there aren't any exams for this class, we will use this for covering other material)

Recap

- Data-savviness is the future!
- "Classical" relational databases
    - Notion of a DBMS
    - The relational data model and algebra: bags and sets
    - SQL Queries, Modifications, DDL
    - Database Design
    - Views, constraints, triggers, and indexes
    - Query processing & optimization
    - Transactions
- Non-classical data systems
    - Semi-structured data and document stores
    - Unstructured data and search engines
    - Cell-structured data and spreadsheets
    - Next: Dataframes and dataframe systems

So far…

- After relational/structured data, we've studied
  - Semi-structured data, where the schema is nested, flexible, and non-atomic…
  - Cell-structured data, where there is no schema, and cells can be filled in with data or with computation
- We're now going to look at yet another way to relax requirements from a database

Classical Database Assumption III

- *Data systems should manage data in relations that are unordered, with a predefined and rarely-modified schema, with queries that operate on relations as a whole, and primarily leverage relational algebra.*

- We'll consider dataframe systems, where these assumptions are relaxed before returning to these assumptions.

Relational databases are painful for exploratory data science

- From a data model standpoint:
  - Data needs to be defined "schema-first": each column needs to have a pre-declared type

- From a querying standpoint:
  - The declarative nature of SQL makes it hard to incrementally construct and debug queries
    - Direct manipulation is one possible solution but for those conversant in programming may be overkill, especially on large data
  - Hard to mix-and-match programming (e.g. python) and dataframes
  - Hard to employ linear algebra (e.g., transpose, matrix multiplication)

What are dataframes?

- Dataframes are an abstraction (a data model and query language) used to
  - represent, structure, clean, and analyze data
  - during exploratory data analysis and data science
- Perform relational, cell-structured, and linear algebraic operations
- The convenience, flexibility, and power of dataframes make it a one-stop-shop for all stages of data science
- Dataframes are embedded in conventional programming languages like Python or R.
  - Pandas, Python's dataframe, short for "Python Data Analysis Library"
  - R comes bundled with an native dataframe library

Dataframes: Popularity

- Python's popularity (now exceeding Java and C++) has been attributed to that of Pandas
- Pandas has been
  - downloaded 200M times
  - a dependency for 150K+ packages on GitHub

- R's dataframe libraries are similarly popular
- All the more reason for us to study it!

Dataframes: History

More in our paper [Petersohn, …, P., 2020]

- S: the predecessor of R was developed in Bell Labs in 1976: a prog. language for statistics
- Dataframes were added by Chambers and Hastie to S in 1990
  - *"We have introduced into S a class of objects called data.frames, which can be used if convenient to organize all of the variables relevant to a particular analysis …"*
- They then wrote a book about it in 1992, where they state:
  - *"Data frames are more general than matrices in the sense that matrices in S assume all elements to be of the same mode—all numeric, all logical, all character string, etc"*
  - *"... data frames support matrix-like computation, with variables as columns and observations as rows, and, in addition, they allow computations in which the variables act as separate objects, referred to by name."*
- To them, dataframes were removing a key limitation of matrices, which is that all data is of the same type.
  - Of course, relations don't have this limitation, but they also do not support matrix-like computation.

Dataframes: History

More in our paper [Petersohn, …, P., 2020]

- Dataframes were added by Chambers and Hastie to S in 1990
  - To them, dataframes were removing a key limitation of matrices, which is that all data is of the same type.
    - Of course, relations don't have this limitation, but they also do not support matrix-like computation.
- R was released in 1995 (with a stable version in 2000) as an open-source implementation of S, gaining popularity in the statistics community
- Wes McKinney brought dataframe capabilities to Python in the form of Pandas in 2008.

Dataframe Basic Demo

- The demo will show the fact that dataframes mix features from databases, spreadsheets, and matrices, all in a convenient syntax in a conventional programming language

Dataframe Data Model

- An array (A, size m x n), where each entry could be of arbitrary type, with m labels, one for each row, and n labels, one for each column (i.e., attribute names)
- An array implicitly has order along both the rows and the columns
- Rows and columns are equivalent (except for one aspect…)
  - Many operations on dataframes can be applied on both rows and columns by simply setting the *axis*
  - Projections can become selections by changing the axis
- The labels are called "indexes" because they allow you to index into the array
  - Note potential confusion with database indexes
- Optionally, each column may have a predefined type, or it can be inferred
  - That is, it is a *lazy* schema

	Price	Rating	Wireless Charging	Display_5.8-inch	...
Price	29868.3	19.967	-16.8317	33.3333	...
Rating	19.967	0.0466667	-7.40149e-17	-0.0666667	...
Wireless Charging	-16.8317	-7.40149e-17	0.25	-0.166667	...
Display_5.8-inch	33.3333	-0.0666667	-0.166667	0.333333	...

Dataframe Query Language

- At least in Pandas, there are lots of operations (**200+**)
  - Rich functionality all emerging "bottom-up" based on open-source contributions to satisfy immediate needs
  - Combining relational (ordered), cell-structured, and linear algebra

- Q: what is the downside of having such a large # of operations?
  - Hard to remember
  - Hard to optimize
    - Cannot come up with rewrite rules for so many operations
  - Sadly, it is impossible to cover in class! Google is your friend
- Thus, dataframe systems perform very little optimization and almost no rewriting

As a result… many different ways of doing the same thing

1700x difference!

A Beginner's Guide to Optimizing Pandas Code for Speed

Sofia Heisler Follow
Aug 2, 2017 · 9 min read

Methodology	Average single run time	Marginal performance improvement
Crude looping	645 ms	
Looping with iterrows()	166 ms	3.9x
Looping with apply()	90.6 ms	1.8x
Vectorization with Pandas series	1.62 ms	55.9x
Vectorization with NumPy arrays	0.37 ms	4.4x

1600 rows

Onus on the user to manually optimize

Other challenges with optimization

- Dataframes are entirely main-memory resident, so no datasets that go beyond main-memory can be processed
- Users construct queries piecemeal
  - Could perform join first, and then additional filters later (which could be pushed down)
    - This can overwhelm main-memory and/or be very slow
  - If you are joining k tables, you could join them in poor orders, e.g., joining largest tables first
    - This could overwhelm main-memory even though the desired result fits in main-memory
  - No pipelining of intermediate results
- Dataframes are stored the exact same way they are displayed, so the user has to determine if specific orders may make certain operations faster
  - Limits the kinds of optimizations you can perform
  - E.g., can't use hash-join because the result will not be sorted
    - Or will need to sort after a join.

Q: How do we apply database optimization techniques manually to large dataframes?

- Which techniques can we apply?
- Think query optimization (rewriting, join ordering), indexing, materialization, pipelining….

Rules of thumb for processing large dataframes

- Apply relational algebra rules: reduce the size of intermediate results
  - Push down predicates
  - Project out columns that won't be used
  - Avoid cross-dataframe operations as much as possible
    - If you must use them, "join" smaller tables first, avoid large table joins.
    - Do mental math to estimate the size of intermediate results.
      - If the operation is hitting >1M rows, consider taking a sample first before you perform it on the entire dataset
- Consider resorting the dataframe to make it cheaper to conduct future operations: like a sort prior to a merge!

Note When writing performance-sensitive code, there is a good reason to spend some time becoming a reindexing ninja: **many operations are faster on pre-aligned data**. Adding two unaligned DataFrames internally triggers a reindexing step. For exploratory analysis you will hardly notice the difference (because `reindex` has been heavily optimized), but when CPU cycles matter sprinkling a few explicit `reindex` calls here and there can have an impact.

- If you have spent a lot of time constructing an intermediate result, consider applying materialization to save the results in a CSV

Ordered Relational Algebra in Dataframes

- Demo: Dataframes support an ordered version of relational algebra… we'll go through each type of operation …

Mapping to Relations

- Dataframes can be easily mapped to relations:
  - The "index" can be treated as a special column of labels
    - The order can be captured as a separate column

- Ordered relational operations are simply relational operations, followed by a sorting by the order column(s)

Classical Database Assumption III

- *Data systems should manage data in relations that are unordered, with a predefined and rarely-modified schema, with queries that operate on relations as a whole, and primarily leverage relational algebra.*

- Dataframes are ordered,
  - With a flexible schema (and flexible types per cell)
- Queries include ordered relational algebra, but also
  - Matrix and point-wise (cell-oriented)

Main Takeaways

- Dataframes are a convenient abstraction for exploratory data analysis for small-to-medium sized datasets, supporting a rich collection of primitives
- Dataframes are simply ordered relations, with
  - Lazy typing
  - Convenient row labeling (a special column)
- Dataframes encompass ordered relational algebra, linear algebra, and cell-structured algebra
  - But its expressiveness means that the onus is on the user to perform manual optimization
- Due to its incremental, piece-meal construction, and virtually no optimization, dataframes could have performance issues on large datasets
  - Care needs to be taken to ensure that intermediate result sizes don't overwhelm main-memory
  - Apply database optimization principles: reorganizing, materializing, reducing intermediate results, defer expensive operations

Research Plug

- Bringing database query processing & optimization ideas to dataframes
  - Identifying a kernel of dataframe operations and rewriting all pandas operations using that kernel
- Starting point: applying parallel processing to scale up dataframe execution
- More at: modin.org