Scaling up your pandas workflows with Modin

Devin Petersohn
Co-founder and CTO, Ponder
About me

- Active Duty U.S. Marine
  - Korean Crypto-Linguist
  - 3d Radio BN (2008-2012)
- BS - University of Missouri (2016)
- MS - UC Berkeley (2018)
- PhD - UC Berkeley (2021)
  - NSF Graduate Research Fellow
  - Chancellor’s Fellow
  - Modin started as my PhD project
- Currently: Cofounder and CTO of Ponder
A dataframe built from first principles
What problems do data science teams face?
Data Science has a scalability problem
Data Science has a scalability problem
Data Science Teams* have a scalability problem
Data Science Teams* have a scalability problem

MORE DATA SCIENTISTS != MORE INSIGHTS
Data Science Teams* have a scalability problem

MORE DATA SCIENTISTS != MORE INSIGHTS

MORE DATA SCIENTISTS != MORE PRODUCTION MODELS/JOBS
Many organizations look like this

New Data Source

Prototyping

Exploring

New spec

New requirements

Laptop/Workstation

pandas
Many organizations look like this

- New Data Source
- New spec
- New requirements

- Prototyping
- Exploring
- Testing
- Rewrite

- Big Data Tool

Laptop/Workstation

Small Cluster
Many organizations look like this

- New Data Source
- New spec
- New requirements

- Laptop/Workstation
- Prototyping
- Exploring

- Big Data Tool

- Small Cluster
- Testing

- Big Data Tool

- Large Cluster
- Production
Many organizations look like this

- Laptop/Workstation
- Prototyping
- Exploring
- Big Data Tool
- Small Cluster
- Testing
- Big Data Tool
- Large Cluster
- Production
- Feedback
- New Data Source
- New spec
- New requirements
- Rewrite

Rewrite
FINALLY FINISHED MY PANDAS NOTEBOOK ON THIS SAMPLE DATA

NOW TO REWRITE IT FOR THE CLOUD
Data Science scalability is human scalability

There is no service that can spin up more Data Scientists, so we must treat them like the finite resource they are
Data Science scalability is a human scalability issue. There is no service that can spin up more Data Scientists, so we must treat them like the finite resource they are.
Shifting the focus from the machine to the user

Data Scientists shouldn’t have to work for their tools

Tools should work for data scientists
Ponder’s work:

Transparencyly scale existing tools

Abstract away all of the components of the system that data scientists don’t care about, only expose details they do care about.
Ponder’s work:

Transparently scale existing tools

Abstract away all of the components of the system that data scientists don’t care about, only expose details they do care about.
Ease of use
expressiveness, flexibility, agility

Performance
scalability, robustness, efficiency

pandas

PONDER
(powered by Modin)

big data frameworks
Let’s start from first principles!
But the Pandas API doesn’t scale!

The API is simply an expression of what to do.
Solving dataframes from first principles

- Dataframe Data Model
- Dataframe Algebra
- Parallelism / Decomposition Rules
- Type System
- Operator Semantics
- Implementation
Towards Scalable Dataframe Systems

Devin Petersohn, William Ma, Doris Lee, Stephen Macke, Doris Xin, Xiangxi Mo
Joseph E. Gonzalez, Anthony D. Joseph, Joseph M. Hellerstein, Aditya Parameswaran
UC Berkeley

{devin.petersohn, williamma, dorislee, smacke, dorx, jxzno, jagonzal, adj, hellenstein, adityapp} @berkeley.edu

ABSTRACT

Dataframes are a popular and convenient abstraction to represent, structure, clean, and analyze data during exploratory data analysis. Despite the success of dataframe libraries in R and Python (pandas), dataframes face performance issues even on moderately large datasets. In this vision paper, we take the first steps towards formally defining dataframes, characterizing their properties, and outlining a research agenda towards making dataframes more interactive at scale. We draw on tools and techniques from the database community, and describe ways they may be adapted to serve dataframe systems, as well as the new challenges therein. We also describe our current progress toward a scalable dataframe system, MODIN, which is already up to 30x faster than pandas in preliminary case studies, while enabling unmodified pandas code to run as-is. In its first 18 months, MODIN is already used by over 60 downstream projects, has over 250 forks, and 3,900 stars on GitHub, indicating the pressing need for pursuing this agenda.

Characteristics such as these have helped dataframes become incredibly popular for EDA; for instance, the dataframe abstraction provided by pandas within Python (pandas.pydata.org), has, as of 2019, been downloaded over 200 million times, served as a dependency for over 160,000 repositories in GitHub, and starred on GitHub more 22,000 times. Python’s own popularity has been attributed to the success of pandas for data exploration and data science [7, 9]. Due to its ubiquity, we focus on pandas for concreteness.

Pandas has been developed from the ground up via open-source contributions; pandas’ DataFrame API operators and their implementations have been provided by dozens of contributors to satisfy immediate or ad-hoc needs, spanning capabilities that mimic relational algebra, linear algebra, and spreadsheet formula computation. To date, the pandas DataFrame API has ballooned to over 200 distinct operators [14]. R, which is both more mature and more carefully curated than pandas, has only 70 operators—but this is still far more than, say, relational and linear algebra combined [15].

While this rich API is sometimes cited as a reason for pandas’ attractiveness, the set of operators has significant redundancies, often with different performance implications. These redundancies place considerable burden on users, exacting the surprised costs of learning the pandas API and maintaining its documentation.

INTRODUCTION

...
ROASTED 'HOGiterranean' "MEATLOAF" THIEVIN' STUFF

La Ratte Potato Puree, Crispy Opalini Onions, Smoked Tomato Claze, Watercress Leaves and Whole Grain Mustard Gravy

"GOICERE"

Andante Dairy "Bufo" and Preserved Australian Black Winter Truffle "Fondue"
Dataframes: A New Data Model and API

- df.groupby(..)
- df.drop(..)
- pd.merge(..)
- df.describe(..)
- pd.concat(..)
- df.pivot(..)
- df.explode(..)

600+ functions to clean, reshape, explore, and summarize data spanning relational, linear, and spreadsheet algebra.
Convenience

Entire query at once

Flexible

Strict schema

Versatility

SFW or bust

Incremental + inspection

Mixed types, R/C and data/metadata equiv.

600+ functions
Everybody loves pandas!

What's the future of the pandas library?

pandas is a powerful, open source Python library for data analysis, manipulation, and visualization. I've been teaching data scientists to use pandas since 2014, and in the years since, it has grown in popularity to an estimated 5 to 10 million users and become a "must-use" tool in the Python data science toolkit.

Python explosion blamed on pandas

Data science fad just won't die

Not content to bait developers by declaring that Python is the fastest-growing major programming language, coding community site Stack Overflow has revealed the reason for its metastasis.

Meet the man behind the most important tool in data science
Inherited Data Model

Small group development

- 1990
- 1995
- 2000 (stable)

~70 operators

Large community development

- 2008
- >200 operators

pandas
Let $\Sigma^*$ be the finite set of characters from alphabet $\Sigma$.

Let $\text{Dom}$ be a finite set of domains $\{\text{dom}_1, \text{dom}_2, \ldots\}$.

Let each $\text{dom}_i \in \text{Dom}$ have a mapping $p_i : \Sigma^* \rightarrow \text{dom}_i$.

A dataframe is a tuple $(A_{mn}, R_m, C_n, D_n)$, where $A_{mn}$ is an arrangement of entries in columns and rows from the domain $\Sigma^*$, $R_m$ is a vector of row labels from $\Sigma^*$, $C_n$ is a vector of column labels from $\Sigma^*$, and $D_n$ is a vector of $n$ domains from some finite set of domains $\text{Dom}$, one per column, each of which can also be left unspecified. We call $D_n$ the schema of the dataframe. If any of the $n$ entries within $D_n$ is left unspecified, then that domain can be induced by applying a schema induction function $S(\cdot)$ to the corresponding column of $A_{mn}$. The schema induction function $S : \Sigma^* \rightarrow \text{Dom}$, assigns an arrangement of $m$ strings to a domain in $\text{Dom}$.
Dataframe data model

- Ordered, but not necessarily sorted
  - Rows and columns
- No predefined schema necessary
  - Types can be induced at runtime
- Typed Row/column labels
  - Labels can become data
- Indexing by label or by row/column numeric index
  - “Named notation” or “Positional notation”
Dataframes from two perspectives

From a **relational algebra** perspective, dataframes contain:

- An ordered table
- Named rows of arbitrary type
- A lazily-induced schema
- Column names of arbitrary type
- Column and row symmetry
- Support for linear algebra operators (e.g. matrix multiply)
Dataframes from two perspectives

From a **linear algebra** perspective:
- Heterogeneous matrix-like data structure
- Both numeric and non-numeric types
- Explicit row and column labels
- Indexing by label in addition to position
- Support for relational algebra operators (e.g. join)
- Dataframe Data Model ✔
- Dataframe Algebra
- Parallelism / Decomposition Rules
- Type System
- Operator Semantics
- Implementation
Next: What can a dataframe do?

First Principles next steps:
define an algebra
What can Pandas Do?

- **pd.DataFrame**: 280+ methods
- **pd.Series**: 280+ methods
- **Convenience methods (e.g. concat)**: 40+ APIs

**Total: 600+ operators in pandas**
<table>
<thead>
<tr>
<th>Example Pandas APIs</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>SELECTION</td>
</tr>
<tr>
<td>add</td>
<td>PROJECTION</td>
</tr>
<tr>
<td>all</td>
<td>UNION</td>
</tr>
<tr>
<td>corr</td>
<td>DIFFERENCE</td>
</tr>
<tr>
<td>dropna</td>
<td>CROSS-PRODUCT / JOIN</td>
</tr>
<tr>
<td>get</td>
<td>DROP DUPLICATES</td>
</tr>
<tr>
<td>groupby</td>
<td>GROUPBY</td>
</tr>
<tr>
<td>head</td>
<td>SORT</td>
</tr>
<tr>
<td>join</td>
<td>RENAME</td>
</tr>
<tr>
<td>median</td>
<td>WINDOW</td>
</tr>
<tr>
<td>pivot</td>
<td>TRANSPOSE</td>
</tr>
<tr>
<td>reindex</td>
<td>MAP</td>
</tr>
<tr>
<td>query</td>
<td>TOLABELS</td>
</tr>
<tr>
<td>rolling</td>
<td>FROMLABELS</td>
</tr>
<tr>
<td>tail</td>
<td></td>
</tr>
<tr>
<td>where</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example R APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>t()</td>
</tr>
<tr>
<td>melt()</td>
</tr>
<tr>
<td>merge()</td>
</tr>
<tr>
<td>head()</td>
</tr>
<tr>
<td>slice()</td>
</tr>
<tr>
<td>distinct()</td>
</tr>
<tr>
<td>arrange()</td>
</tr>
<tr>
<td>rename()</td>
</tr>
<tr>
<td>summary()</td>
</tr>
<tr>
<td>group_by()</td>
</tr>
<tr>
<td>aggregate()</td>
</tr>
<tr>
<td>union()</td>
</tr>
<tr>
<td>with()</td>
</tr>
<tr>
<td>subset()</td>
</tr>
<tr>
<td>match()</td>
</tr>
<tr>
<td>sample_n()</td>
</tr>
<tr>
<td>row_labels()</td>
</tr>
</tbody>
</table>
**Proof by exhaustion that all pandas APIs are covered**

<table>
<thead>
<tr>
<th>Operator/Pattern</th>
<th>Module Syntax</th>
<th>Exhaustive list of pandas public API</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applying a user-defined function uniformly element-wise, column-wise, or row-wise</td>
<td><code>map(df, axis, f)</code></td>
<td><code>abs, asarray, clip, eval, isnan, isnull, notnull, fillna, isnan, apply, applymap, transform, fillna, rank, round, unary version of (add, subtract, etc.), string manipulations (str), datetime manipulation (dt)</code></td>
<td>42</td>
</tr>
<tr>
<td>Binary function between two dataframes</td>
<td><code>map(join(df1, df2, f))</code></td>
<td><code>add, subtract, multiply, divide, eq, isin, xor, where, update, corr, corrwith, combine_first, cov, dot, ge, gt, lt, pow, radd, rsub, rtrunc, round, cuml, rpow, radd, rtrunc, um, trunc, div, true</code></td>
<td>28</td>
</tr>
<tr>
<td>Indexing/Querying on the row labels</td>
<td><code>mask(df, row_indices=query)</code></td>
<td><code>as_freq, asof, at, at_time, between_time, drop, level, drop, first, first_valid_index, loc, last, last_valid_index, loc, mask, pop, get, head, tail, ist, lookup, resample, sample, select, take, truncate</code></td>
<td>10</td>
</tr>
<tr>
<td>Reshaping, transposing, pivoting</td>
<td><code>map(grouper(y) (transpose(df), f))</code></td>
<td><code>pivot, pivot_table, stack, unstack, transpose, T</code></td>
<td>6</td>
</tr>
<tr>
<td>One-hot (dummy) encoding</td>
<td><code>transpose(t1_labels (transpose(map(df, f, axis=1)))</code></td>
<td><code>get_dummies</code></td>
<td>1</td>
</tr>
<tr>
<td>User-defined aggregation of values per-column</td>
<td><code>reduction(df, f)</code></td>
<td><code>all, any, count, agg, aggregate, compound, describe, duplicated, equals, idem, isin, isnull, max, mean, median, min, mode, memory_usage, prod, product, unique, quantile, sem, skew, std, sum, var</code></td>
<td>28</td>
</tr>
<tr>
<td>Aligning and joining two dataframes on row or column labels</td>
<td><code>join_from_labels(df1, from_labels(df2))</code></td>
<td><code>align, concat, join, reindex, reindex_axis, reindex_like</code></td>
<td>6</td>
</tr>
<tr>
<td>Window user-defined functions (window size &lt; length of dataframe)</td>
<td><code>window(df, axis, f, size)</code></td>
<td><code>bfill, cumsum, cumprod, cumsum, cummin, diff, ewm, expanding, fillna, fillna, interpolate, klargest, nsmallest, pct_change, rolling</code></td>
<td>15</td>
</tr>
<tr>
<td>Conditional filter</td>
<td><code>filter(df, f)</code></td>
<td><code>dropna, drop_duplicates, filter, query</code></td>
<td>4</td>
</tr>
<tr>
<td>Core to the schema of the dataframe</td>
<td><code>filter_obj_type(df, types)</code></td>
<td><code>select, dtypes</code></td>
<td>1</td>
</tr>
<tr>
<td>Type Inference and Reduction</td>
<td><code>infer, types(df, columns)</code></td>
<td><code>infer_objects, convert_objects</code></td>
<td>2</td>
</tr>
<tr>
<td>Columnwise insertion and assignment, appending columns/rows</td>
<td><code>concat(df1, df2)</code></td>
<td><code>append, assign, concat, insert</code></td>
<td>4</td>
</tr>
<tr>
<td>Groupby with a user-defined aggregation or function</td>
<td><code>groupby(df, by, f)</code></td>
<td><code>groupby</code></td>
<td>2</td>
</tr>
<tr>
<td>Join on an attribute</td>
<td><code>join(df1, df2, condition, how)</code></td>
<td><code>merge, merge_asof</code></td>
<td>2</td>
</tr>
<tr>
<td>Sorting on labels or column values</td>
<td><code>sort_by(df)</code></td>
<td><code>sort_index, sort_values</code></td>
<td>2</td>
</tr>
<tr>
<td>Expand the number of rows or columns</td>
<td><code>explode(df, f)</code></td>
<td><code>explode</code></td>
<td>1</td>
</tr>
</tbody>
</table>
- Dataframe Data Model ✔
- Dataframe Algebra ✔
- Parallelism / Decomposition Rules
- Type System
- Operator Semantics
- Implementation
Decomposition Rules -> Formalize parallelism

**Cell wise:** An operator can be applied to a “unit dataframe” independently

**Row-wise:** An operator can be applied to each row independently

**Column-wise:** An operator can be applied to each column independently
- Dataframe Data Model
- Dataframe Algebra
- Parallelism / Decomposition Rules
- Type System
- Operator Semantics
- Implementation
Formalization of other components of the dataframe

**Type System**

Unspecified → Any
- String
- Number
- Category
- Datetime
- **User-Defined Type**

- Integer
- Boolean
- Floating Point

**Order Semantics**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Input Order</th>
<th>Position</th>
<th>Output Order &amp; Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>mask</td>
<td>N</td>
<td>Y*</td>
<td>Parameter-Dependent</td>
</tr>
<tr>
<td>filter_by_types</td>
<td>N</td>
<td>N</td>
<td>Inherited I Updated</td>
</tr>
<tr>
<td>map</td>
<td>N</td>
<td>Y0</td>
<td>Inherited from Inputs</td>
</tr>
<tr>
<td>filter</td>
<td>N</td>
<td>Y0</td>
<td>Inherited I Updated</td>
</tr>
<tr>
<td>explode</td>
<td>N</td>
<td>Y0</td>
<td>Inherited I Updated</td>
</tr>
<tr>
<td>reduce</td>
<td>N</td>
<td>Y0</td>
<td>Inherited</td>
</tr>
<tr>
<td>window</td>
<td>Y</td>
<td>N</td>
<td>Inherited</td>
</tr>
<tr>
<td>groupby</td>
<td>N</td>
<td>N</td>
<td>Data-dependent</td>
</tr>
<tr>
<td>infer_types</td>
<td>N</td>
<td>N</td>
<td>Inherited</td>
</tr>
<tr>
<td>join</td>
<td>N</td>
<td>N</td>
<td>Inherited*</td>
</tr>
<tr>
<td>concat</td>
<td>N</td>
<td>N</td>
<td>Inherited*</td>
</tr>
<tr>
<td>transpose</td>
<td>N</td>
<td>N</td>
<td>Inherited</td>
</tr>
<tr>
<td>to_labels</td>
<td>N</td>
<td>N</td>
<td>Inherited</td>
</tr>
<tr>
<td>from_labels</td>
<td>N</td>
<td>Y</td>
<td>Inherited</td>
</tr>
<tr>
<td>sort</td>
<td>N</td>
<td>N</td>
<td>Data-dependent</td>
</tr>
<tr>
<td>rename</td>
<td>N</td>
<td>N</td>
<td>Inherited</td>
</tr>
</tbody>
</table>
- Dataframe Data Model ✓
- Dataframe Algebra ✓
- Parallelism / Decomposition Rules ✓
- Type System ✓
- Operator Semantics ✓
- Implementation
So now we know:

- What a dataframe is (formally)
- What operators a dataframe supports
- How these operators map back to pandas
- How to handle dataframe types
- How to decouple logical and physical order
- How to maximally parallelize each operator
A dataframe built from first principles
Modin
Accelerate your pandas workloads by changing one line of code

To use Modin:

```python
import pandas as pd
```

To install Modin:

```
pip install modin
```
What can we do with the formalism?

- Data model
  - Expose the “feel” of pandas without the baggage
- Dataframe Algebra
  - Smaller surface to implement
  - Map operators to other systems
Pandas on everything with Ponder

Databases
- PostGresQL
- snowflake
- amazon REDSHIFT
- Google BigQuery

Your DB Here!

Compute Engine
- RAY
- DASK
- Apache Spark

Infrastructure
- Cloud
- AWS
- Azure
- Intel
Many organizations look like this
How Modin is being used in practice:

- Prototyping
- Exploring
- Testing
- Production

New Data Source
New spec
New requirements

Laptop/Workstation
Small Cluster
Large Cluster

Feedback
Modin Open Source Impact

Used by 10% Fortune 100 companies & more!

- Bosch
- VMware
- Apple
- Bristol Myers Squibb
- lululemon
- LogMeIn
- Intel
- Cisco
- GSK
- Tesla
- Allianz
- VOXMEDIA
- Nokia
- Atlassian
- Azure
- Berkeley Lab
- Kevala

- Downloads to date: 2.6M+
- Github stars: 7k
- Open Source Contributors: 90+
E-Commerce Case Study

Current State

Pricing + Recommendation Pipelines in Production

150M Events Streamed Daily

50k Records Pandas Limit

With Ponder

Modin

1000X More Data with Modin

< 20 Sec. Faster with Modin
Finance Case Study

Current State

Regulatory Reporting
Pipelines Migration

10k+ Lines
Of Code to change est.

2k+ Hours
Human time to rewrite est.

With Ponder

Modin

100X
Reduction in code reduction w/ Modin

100X
Reduction in human time with Modin
Bring your database!

Databases:
- PostGresQL
- snowflake
- amazon REDSHIFT
- Google BigQuery

Your DB Here!

Compute Engine:
- RAY
- DASK
- Apache Spark

Infrastructure:
- Ponder
- AWS
- Azure
- Intel
In summary...

- Formal dataframe data model & algebra
  - Dataframes are newly defined structures with a lot of open problems!
  - Dataframe algebra can express all of pandas
- Reference implementation: Modin
  - High impact -> problems are real and pressing need
- Deep technical problems still exist!
  - It is an exciting time to be working on dataframes
Thank you!

Devin Petersohn
devin@ponder.io
Current state of affairs
Current state of affairs

- Coverage of pandas API
  - pandas.DataFrame - 83% (93% based on usage)
  - pandas.Series - 77% (86% based on usage)
  - pandas.read_* - 42% (>90% based on usage)
API Vision - 5 years

- More complete pandas API - 95%
- All common interactive data processing modalities
  - Spreadsheet API
  - SQL API
  - New Modin API
- Hooks for SQL systems to implement parts of the API
  - A partial pandas API for relational systems
- Preliminary numpy API - more implementation proof that dataframes can act as matrices
  - Plug in to sklearn
Engineering Vision - 5 years

- Query planning/optimization
  - Optimize for user’s time
  - Some research involved here (more later)
  - Extending Calcite?
- GPU integration
- MPI, other compute engines
- Serverless
Case study: Pivot

<table>
<thead>
<tr>
<th>Year</th>
<th>Month</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001</td>
<td>Jan</td>
<td>100</td>
</tr>
<tr>
<td>2001</td>
<td>Feb</td>
<td>110</td>
</tr>
<tr>
<td>2001</td>
<td>Mar</td>
<td>120</td>
</tr>
<tr>
<td>2002</td>
<td>Jan</td>
<td>150</td>
</tr>
<tr>
<td>2002</td>
<td>Feb</td>
<td>200</td>
</tr>
<tr>
<td>2002</td>
<td>Mar</td>
<td>250</td>
</tr>
<tr>
<td>2003</td>
<td>Jan</td>
<td>300</td>
</tr>
<tr>
<td>2003</td>
<td>Feb</td>
<td>310</td>
</tr>
</tbody>
</table>

Wide Table of MONTHS

<table>
<thead>
<tr>
<th>Month</th>
<th>2001</th>
<th>2002</th>
<th>2003</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan</td>
<td>100</td>
<td>150</td>
<td>300</td>
</tr>
<tr>
<td>Feb</td>
<td>110</td>
<td>200</td>
<td>310</td>
</tr>
<tr>
<td>Mar</td>
<td>120</td>
<td>250</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Pivot ➝

Unpivot ➞

Wide Table of YEARS

<table>
<thead>
<tr>
<th>Year</th>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001</td>
<td>100</td>
<td>110</td>
<td>120</td>
</tr>
<tr>
<td>2002</td>
<td>150</td>
<td>200</td>
<td>250</td>
</tr>
<tr>
<td>2003</td>
<td>300</td>
<td>310</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Current Machine Learning Lifecycle

New Data Source → Prototyping → Testing → Production → Feedback

- New spec
- New requirements

Prototyping
Exploring

Laptop/Workstation
Small Cluster
Large Cluster

Rewrite

pandas
\[ y_{i} = \beta_{1}x_{i1} + \varepsilon_{i} \]
One API, all scales (think SQL)
### Data Science Landscape: Today

<table>
<thead>
<tr>
<th>Tools efficient for $O(1\text{MB})$</th>
<th>Tools efficient for $O(100\text{s GB}+) $</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usable but not scalable</td>
<td>Scalable but not usable</td>
</tr>
</tbody>
</table>

- **usable but not scalable:**
  - pandas

- **scalable but not usable:**
  - Apache Spark

**Equation:**

$$y_{it} = \beta' x_{it} + \mu_t + \epsilon_{it}$$
## Data Science Landscape: Today

<table>
<thead>
<tr>
<th>Tools efficient for O(1MB)</th>
<th>Tools efficient for O(100s GB+)</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Follow typical programming styles</td>
<td>● Difficult to debug</td>
</tr>
<tr>
<td>● Tools are widely used and understood - in production</td>
<td>Required knowledge related to distributed computing</td>
</tr>
<tr>
<td>● The majority of college graduates will already know these tools</td>
<td>○ Must understand partitioning</td>
</tr>
<tr>
<td>● No scalability</td>
<td>○ Lazy evaluation hated</td>
</tr>
<tr>
<td></td>
<td>● Designed by systems people for systems people</td>
</tr>
<tr>
<td></td>
<td>○ New APIs that do the same thing</td>
</tr>
</tbody>
</table>

- Follow typical programming styles
- Tools are widely used and understood - in production
- The majority of college graduates will already know these tools
- No scalability

- Difficult to debug
- Required knowledge related to distributed computing
  - Must understand partitioning
  - Lazy evaluation hated
- Designed by systems people for systems people
  - New APIs that do the same thing
Data Science Landscape: Today

Tools efficient for $O(1MB)$

<table>
<thead>
<tr>
<th>Usable and scalable</th>
</tr>
</thead>
<tbody>
<tr>
<td>pandas ++</td>
</tr>
</tbody>
</table>

Tools efficient for $O(100s \text{ GB}+) $

<table>
<thead>
<tr>
<th>Scalable and usable</th>
</tr>
</thead>
<tbody>
<tr>
<td>pandas</td>
</tr>
</tbody>
</table>

\[ y_{it} = \beta' x_{it} + \mu_t + \epsilon_{it} \]
Dataframe Algebra

Columns become rows/rows become columns

**Operator**
- SELECTION
- PROJECTION
- UNION
- DIFFERENCE
- CROSS-PRODUCT / JOIN
- DROP DUPLICATES
- GROUPBY
- SORT
- RENAME
- WINDOW
- **TRANSPOSE**
- MAP
- TOLABELS
- FROMLABELS

$\text{df.T}$
DataFrame Algebra

**Operator**

- SELECTION
- PROJECTION
- UNION
- DIFFERENCE
- CROSS-PRODUCT / JOIN
- DROP DUPLICATES
- GROUPBY
- SORT
- RENAME
- WINDOW
- TRANPOSE
- MAP
- TO LABELS
- FROM LABELS

User defined function across all rows:

```
df.apply(f, axis=1)
```
## Dataframe Algebra

### Operator

<table>
<thead>
<tr>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECTION</td>
</tr>
<tr>
<td>PROJECTION</td>
</tr>
<tr>
<td>UNION</td>
</tr>
<tr>
<td>DIFFERENCE</td>
</tr>
<tr>
<td>CROSS-PRODUCT / JOIN</td>
</tr>
<tr>
<td>DROP DUPLICATES</td>
</tr>
<tr>
<td>GROUPBY</td>
</tr>
<tr>
<td>SORT</td>
</tr>
<tr>
<td>RENAME</td>
</tr>
<tr>
<td>WINDOW</td>
</tr>
<tr>
<td>TRANSPOSE</td>
</tr>
<tr>
<td>MAP</td>
</tr>
<tr>
<td>TO LABELS</td>
</tr>
<tr>
<td>FROM LABELS</td>
</tr>
</tbody>
</table>

### Dataframe Algebra Diagram

- **Row Labels** $R_m$
- **Column Domains** $D_n$
- **Column Labels** $C_n$
- **Array of Data** $A_{mn}$

### Code Examples

```python
df.set_index(df["col"])  
df.reset_index(drop=False)
```

Elevating data into metadata, or moving metadata into the data.
Working with tabular data

A (very) brief history
Relational Databases

- Invented in the 1970’s by Edgar F. Codd
  - Defined a data model for **structured data**
  - Schema must be known before input into DBMS
  - Decoupling of the physical representation from logical
- Popularized in the 1980’s
  - Data model has stood the test of time, still in use!
  - SQL
Census - the original “Big Data”

- 1880’s - Seaton Device, manual intervention and counting - took almost 10 years
- 1890’s - Herman Hollerith’s machine, punch cards
- 1940’s - First use of electronic computers
Dataframes

- Emerged from a real-world need at the time
  - No way of handling unstructured or semi-structured data
  - Matrices and Tables did not fit their need, something new needed
- Not formalized!
- Dataframes have an origin in S
Focus: pandas
pandas success

- 23 million installs/month
- Over 300 million total installs
- Used by over 209k projects in GitHub
- 24.6k GitHub stars
Pandas API

- pd.DataFrame: 280+ methods
- pd.Series: 280+ methods
- Convenience methods (e.g. concat): 40+ APIs
pandas API is huge and expressive

- **pd.DataFrame**: 280+ methods
- **pd.Series**: 280+ methods
- **Convenience methods (e.g. concat)**: 40+ APIs
What do people use within pandas API?

Top 30 most used pandas APIs by count -- Kaggle notebooks dataset

https://github.com/modin-project/study_kaggle_usage
Definition: Dataframes

$A_{mn}$

Array of Data
Definition: Dataframes

Array of Data

$A_{mn}$

$R_m$

Row Labels
Definition: Dataframes

$R_m$ Row Labels

$C_n$ Column Labels

$A_{mn}$ Array of Data
Modin architecture

APIs

- pandas
- SQLite (Experimental)
- MODIN API (Coming Soon™)

MODIN Query Compiler

MODIN DataFrame

RAY
DASK
python

??? Bring your Distributed DataFrame

??? Bring your backend
Modin architecture

Query Compiler

pandas

SQLite (Experimental)

MODIN API (Coming Soon™)

MODIN Query Compiler

MODIN DataFrame

RAY

DASK

python

??? Bring your backend

??? Bring your Distributed DataFrame
## Modin Architecture

<table>
<thead>
<tr>
<th>pandas</th>
<th>SQLite (Experimental)</th>
<th>MODIN API (Coming Soon™)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MODIN Query Compiler</td>
<td></td>
</tr>
</tbody>
</table>

### Middle Layer

**Modin**

DataFrame

- **RAY**
- **DASK**
- **Python**

??? Bring your backend

??? Bring your Distributed DataFrame
Modin architecture

- **pandas**
- **MODIN API** (Coming Soon™)
- **MODIN Query Compiler**
- **MODIN DataFrame**

Execution:
- **RAY**
- **DASK**
- **python**

Bring your Distributed DataFrame

Bring your backend
Dataframes emerged from a need to hybridize Matrix, Relational Table, and Spreadsheet.
Dataframes emerged from a need to hybridize Matrix, Relational Table, and Spreadsheet.

Matrix:

\[
\begin{pmatrix}
0 & 3 & 1 & 0 & 2 & 3 & 8 & 1 & 1 & 3 \\
1 & 1 & 0 & 0 & 7 & 1 & 2 & 2 & 3 & 3 \\
1 & 2 & 2 & 0 & 0 & 6 & 7 & 1 & 2 & 2 \\
1 & 2 & 3 & 10 & 0 & 4 & 6 & 1 & 0 & 5 \\
3 & 2 & 2 & 1 & 4 & 3 & 2 & 1 & 6 & 0 \\
7 & 4 & 4 & 5 & 3 & 9 & 6 & 1 & 6 & 1 \\
7 & 1 & 1 & 5 & 2 & 8 & 9 & 1 & 3 & 6 \\
5 & 0 & 1 & 6 & 2 & 0 & 0 & 0 & 1 & 5 \\
1 & 6 & 3 & 3 & 4 & 6 & 2 & 0 & 1 & 1 \\
1 & 2 & 2 & 4 & 1 & 1 & 3 & 0 & 8 & 2
\end{pmatrix}
\]

Relational Table:

<table>
<thead>
<tr>
<th>Name</th>
<th>FName</th>
<th>City</th>
<th>Age</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>John</td>
<td>3</td>
<td>35</td>
<td>$280</td>
</tr>
<tr>
<td>Doe</td>
<td>Jane</td>
<td>1</td>
<td>28</td>
<td>$325</td>
</tr>
<tr>
<td>Brown</td>
<td>Scott</td>
<td>3</td>
<td>41</td>
<td>$265</td>
</tr>
<tr>
<td>Howard</td>
<td>Shemp</td>
<td>4</td>
<td>48</td>
<td>$359</td>
</tr>
<tr>
<td>Taylor</td>
<td>Tom</td>
<td>2</td>
<td>22</td>
<td>$250</td>
</tr>
</tbody>
</table>

Spreadsheet:

Dataframes
## Dataframe Algebra

<table>
<thead>
<tr>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECTION</td>
</tr>
<tr>
<td>PROJECTION</td>
</tr>
<tr>
<td>UNION</td>
</tr>
<tr>
<td>DIFFERENCE</td>
</tr>
<tr>
<td>CROSS-PRODUCT / JOIN</td>
</tr>
<tr>
<td>DROP DUPLICATES</td>
</tr>
<tr>
<td>GROUPBY</td>
</tr>
<tr>
<td>SORT</td>
</tr>
<tr>
<td>RENAME</td>
</tr>
<tr>
<td>WINDOW</td>
</tr>
<tr>
<td>TRANSPOSE</td>
</tr>
<tr>
<td>MAP</td>
</tr>
<tr>
<td>TOLABELS</td>
</tr>
<tr>
<td>FROMLABELS</td>
</tr>
</tbody>
</table>

“Position notation”  
or  “Named notation”  

```python
df.iloc[row_pos, col_pos]
df.loc[row_lab, col_lab]
```
## Dataframe Algebra

<table>
<thead>
<tr>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECTION</td>
</tr>
<tr>
<td>PROJECTION</td>
</tr>
<tr>
<td>UNION</td>
</tr>
<tr>
<td>DIFFERENCE</td>
</tr>
<tr>
<td>CROSS-PRODUCT / JOIN</td>
</tr>
<tr>
<td>DROP DUPLICATES</td>
</tr>
<tr>
<td>GROUPBY</td>
</tr>
<tr>
<td>SORT</td>
</tr>
<tr>
<td>RENAME</td>
</tr>
<tr>
<td>WINDOW</td>
</tr>
<tr>
<td>TRANSPOSE</td>
</tr>
<tr>
<td>MAP</td>
</tr>
<tr>
<td>TOLABELS</td>
</tr>
<tr>
<td>FROMLABELS</td>
</tr>
</tbody>
</table>

Ordered union

df.append(df2)
# Dataframe Algebra

<table>
<thead>
<tr>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECTION</td>
</tr>
<tr>
<td>PROJECTION</td>
</tr>
<tr>
<td>UNION</td>
</tr>
<tr>
<td>DIFFERENCE</td>
</tr>
<tr>
<td>CROSS-PRODUCT / JOIN</td>
</tr>
<tr>
<td>DROP DUPLICATES</td>
</tr>
<tr>
<td>GROUPBY</td>
</tr>
<tr>
<td>SORT</td>
</tr>
<tr>
<td>RENAME</td>
</tr>
<tr>
<td>WINDOW</td>
</tr>
<tr>
<td>TRANSPOSE</td>
</tr>
<tr>
<td>MAP</td>
</tr>
<tr>
<td>TOLABELS</td>
</tr>
<tr>
<td>FROMLABELS</td>
</tr>
</tbody>
</table>

Ordered Joins bring fundamentally new challenges.

```python
df.merge(df2, how="inner")
df2.merge(df, how="inner")
```
Dataframe Algebra

This is a generalized definition of groupby

df.count()

df.groupby(df.columns).count()
Dataframe Algebra

Conceptually a rolling function, can output same table shape or smaller (groupby)

<table>
<thead>
<tr>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECTION</td>
</tr>
<tr>
<td>PROJECTION</td>
</tr>
<tr>
<td>UNION</td>
</tr>
<tr>
<td>DIFFERENCE</td>
</tr>
<tr>
<td>CROSS-PRODUCT / JOIN</td>
</tr>
<tr>
<td>DROP DUPLICATES</td>
</tr>
<tr>
<td>GROUPBY</td>
</tr>
<tr>
<td>SORT</td>
</tr>
<tr>
<td>RENAME</td>
</tr>
<tr>
<td>WINDOW</td>
</tr>
<tr>
<td>TRANSPOSE</td>
</tr>
<tr>
<td>MAP</td>
</tr>
<tr>
<td>TOLABELS</td>
</tr>
<tr>
<td>FROMLABELS</td>
</tr>
</tbody>
</table>

df.cumsum()
df.rolling
## Dataframe Algebra

<table>
<thead>
<tr>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECTION</td>
</tr>
<tr>
<td>PROJECTION</td>
</tr>
<tr>
<td>UNION</td>
</tr>
<tr>
<td>DIFFERENCE</td>
</tr>
<tr>
<td>CROSS-PRODUCT / JOIN</td>
</tr>
<tr>
<td>DROP DUPLICATES</td>
</tr>
<tr>
<td>GROUPBY</td>
</tr>
<tr>
<td>SORT</td>
</tr>
<tr>
<td>RENAME</td>
</tr>
<tr>
<td>WINDOW</td>
</tr>
<tr>
<td>TRANSPOSE</td>
</tr>
<tr>
<td>MAP</td>
</tr>
<tr>
<td>TOLABELS</td>
</tr>
<tr>
<td>FROMLABELS</td>
</tr>
</tbody>
</table>

**df.reset_index(drop=False)**

Insert the row labels into the data and reset the row labels to the positional notation.
Modin architecture

APIs

Query Compiler

Middle Layer

Execution

pandas

MySQL

SQLite

MODIN API

(Experimental)

MODIN API

(Coming Soon™)

MODIN Query Compiler

MODIN

DataFrame

RAY

DASK

python

???

Bring your Distributed DataFrame

???

Bring your backend
Data Science Landscape: Today

Tools efficient for O(1MB)

Usable but not scalable

pandas

Tools efficient for O(100s GB+)

Scalable but not usable
Data Science Landscape: Today

Tools efficient for O(1MB)

Usable and scalable

pandas

\[ y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it} \]

Tools efficient for O(100s GB+)

Scalable and usable

pandas

\[ y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it} \]
Dataframe origin
Chapter 3: Data for Models

“Dataframes 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.”
Modin partitioning - logical column partitioning
Modin partitioning - logical row partitioning
Modin architecture

• Highly flexible
• Layered architecture
  ○ New optimizations can be implemented as they are developed
• Support for dataframe algebra
• Partitioning approach lends itself to allowing us to use optimizations from multiple domains
The focus of this dissertation (EDA)

- **Dirty Data**
  - Iterative
  - **Unstructured** data model
  - Schema is lazily induced
  - No decoupled physical representation from logical

- **Exploration**
- **Cleaning**
- **Testing**
- **Database**
  - Static
  - Defined a data model for **structured data**
  - Schema must be known before input into DBMS
  - Decoupling of the physical representation from logical

*Important, but not the focus (Production)*
Data Science LifeCycle

The focus of this dissertation (EDA)

- **Iterative**
- **Unstructured** data model
- Schema is lazily induced
- No decoupled physical representation from logical

**Important, but not the focus (Production)**

- Defined a data model for structured data
- Schema must be known before input into DBMS
- Decoupling of the physical representation from logical

Dirty Data

Exploration

Testing

Dataframes

Databases